Cognex MVS-8000 Series

CVL User's Guide

CVL 8.0

June 2016

Chapters

Chapters

Contents

Contents

Contents

Contents

Contents

Preface

This manual contains information about using the Cognex Vision Library (CVL) to develop machine vision applications. It includes information on the following topics:

- How to write a CVL application
- How to acquire and display images
- How to work with images and coordinate systems
- How to work with shapes
- How to use shape models with CVL vision tools
- How to use CVL to do multithreading, persistence, and I/O

Style Conventions Used in This Manual

This manual uses the style conventions described in this section for text and software diagrams.

Text Style Conventions

This manual uses the following style conventions for text:

boldface	Used for C/C++ keywords, function names, class names, structures, enumerations, types, and macros. Also used for user interface elements such as button names, dialog box names, and menu choices.
italic	Used for names of variables, data members, arguments, enumerations, constants, program names, file names. Used for names of books, chapters, and sections. Occasionally used for emphasis.
`courier`	Used for C/C++ code examples and for examples of program output.
`bold courier`	Used in illustrations of command sessions to show the commands that you would type.
<italic>	When enclosed in angle brackets, used to indicate keyboard keys such as *<Tab>* or *<Enter>*.

Microsoft Windows Support

Cognex CVL software runs on several Microsoft Windows operating systems. In this documentation set, *Windows* refers to all supported operating systems, unless there is a feature specific to one of the variants. Consult the *Getting Started* manual for your CVL release for details on the operating systems, hardware, and software supported by that release.

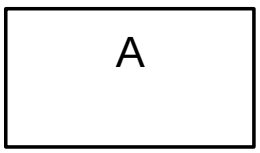
Software Diagramming Conventions

This manual uses the following symbols in class diagrams:

- **Classes** are shown as a box with the class name centered inside the box. For example, a class A with the C++ declaration

  ```
  class A{};
  ```

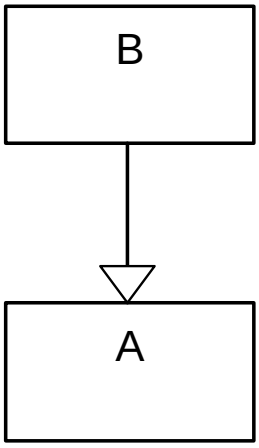
  is shown graphically as follows:



- **Inheritance** relationships between classes are shown using solid-line arrows from the derived class to the base class with a large, hollow triangle pointing toward the base class. For example, a class B that inherits from a class A with the declaration

  ```
  class B : public A {};
  ```

  is shown graphically as follows:



These symbols are based on the Unified Modeling Language (UML), a standard graphical notation for object-oriented analysis and design. See the latest *OMG Unified Modeling Language Specification* (available from the Object Management Group at http://www.omg.org) for more information.

Cognex Offices

Cognex Corporation serves its customers from the following locations:

Corporate Headquarters	Cognex Corporation Corporate Headquarters One Vision Drive Natick, MA 01760-2059 (508) 650-3000
Web Site	http://www.cognex.com

CVL Programming Overview

1

This chapter contains information you need to know to build and run applications built with the Cognex Vision Library (CVL).

Development Environment Requirements describes the development environment you need to write vision applications with CVL. This section also describes the project settings required in the Microsoft Visual C++ development environment to build CVL applications for MVS-81x0 series frame grabbers.

Files Delivered with CVL describes the directories that CVL installs on your hard disk and their contents.

Programming Examples describes the sample programs and the sample code included with CVL. The sample programs are intended to help you get started writing your own CVL applications.

CVL Programming Conventions describes programming conventions used in CVL such as the use of parameterized classes, pointer handles, naming conventions, localization and Unicode, and CVL built-in types.

Development Environment Requirements

To create vision applications with the Cognex Vision Library, your development PC must meet the minimum requirements stated in the *Getting Started* document for your release of CVL.

Configuring your Development Environment

To compile CVL programs, you must set up Microsoft Visual C++ with the correct settings and the appropriate paths to header files and libraries.

The fastest way to set up your Visual C++ project file is to copy one of the sample project files provided with CVL, renaming your copy of the file as appropriate for your application. The sample Visual C++ project files provided are shown in Table 1:

For this compiler	Copy this file	From directory
Visual C++ .NET 2012	*cvlproj11.vcxproj*	*%VISION_ROOT%\sample\cvl\cvlproj11*
Visual C++ .NET 2013	*cvlproj12.vcxproj*	*%VISION_ROOT%\sample\cvl\cvlproj12*
Visual C++ .NET 2015	*cvlproj14.vcxproj*	*%VISION_ROOT%\sample\cvl\cvlproj14*

Table 1. Project files containing CVL project settings

See *Programming Examples* on page 41 for more information on the sample programs.

If you start with a new, blank project file within Visual C++, you can configure the project settings manually as described in *Required Settings for Visual C++ .NET* on page 29.

Setup Requirements for All Users

The CVL sample code presumes the presence of the *VISION_ROOT* environment variable. The CVL installation program sets this variable for you.

VISION_ROOT is set to the path of the top-level CVL directory specified during installation. The usual setting is as follows:

```
VISION_ROOT=C:\Program Files\Cognex\CVL
```

When changing this environment variable manually, use the Control Panel's System Properties dialog so that the variable is visible to Microsoft Visual C++. It is not enough to set the variable at a command prompt.

Environment variables are case-sensitive. If you set the variable "VISION_ROOT" as shown, reference that variable with the same case letters. For example, within Visual C++, the value of this environment variable is referenced with the following syntax:

```
$(VISION_ROOT)
```

For most uses within Windows, the value of this same variable is referenced with this syntax:

```
%VISION_ROOT%
```

Once the *VISION_ROOT* variable is set, users of MVS-8000 frame grabbers should be able to build and run the CVL sample code without further changes.

Tuning System Configuration for Best Performance

Refer to the *PC Configuration Guide* supplied as part of the CVL documentation for information on selecting and configuring your PC system for optimum performance.

Development Environment for Cognex Frame Grabbers

This section contains information on setting up the Visual C++ development environment for users of Cognex frame grabbers.

Creating CVL Projects for Frame Grabbers

CVL projects built for Cognex MVS-8000 frame grabbers are linked as a standard executable, using the linker supplied with Visual C++, and linked against import libraries in *%VISION_ROOT%\lib\win32\cvl*. You can create a new project in two ways:

- Copy the project file for the appropriate development environment to a new file, rename it, then open and modify your renamed copy.
- Start a new Visual C++ project file of type "Win32 Application," and then manually specify the required project settings. In this case, use the settings described in Table 2 on page 30 for Visual C++ .NET projects.

Required Settings for Visual C++ .NET

Table 2 shows all required project settings for MVS-8000 frame grabber projects when using the Microsoft Visual C++ .NET 2012, 2013, and 2015 development environments. All other project settings should be Visual C++ defaults or can be customized for your project.

To generate a new project, click the **New Project** button on the **Start** page. In the **New Project** dialog box, select **Visual C++ Project** for the project type, and select **Win32 Project** as your project's template. Specify a name and location for your project, then click **OK**. In the **Win32 Application Wizard** dialog box, click **Finish**.

To display the **Property Page** for your project, highlight your project's name in the **Solution Explorer**, **Resource View**, or **Class View**, then select **Project** -> **Properties** (or press *<Shift+F4>*).

In the **Property Page**, modify the new project's settings as shown in Table 2.

Configuration	Folder	Category	Item	Settings for Frame Grabber Projects
All Configurations	General	Project Defaults	Use of MFC	• If your application uses **ccDisplayConsole**, select: **Use MFC in a Shared DLL** • If your application uses **ccWin32Display** and you do not want to link against MFC libraries, select: **Use Standard Windows Libraries**
	C/C++	General	Additional Include Directories	**$(VISION_ROOT)\defs**
			Detect 64-bit Portability Issues	**No**
		Preprocessor	Preprocessor Definitions	• Verify that these macros are defined: **WIN32, _WINDOWS** • Add for all projects: **NOMINMAX** • If your application uses **ccWin32Display** and you do not want to link against MFC libraries, add the macro: **cmNoMFCDependency**
		Code Generation	Enable C++ Exceptions	**Yes**
		Language	Treat wchar_t as Built-in Type	**Yes**
			Enable Run-Time Type Info	**Yes**
	Linker	General	Additional Library Directories	**$(VISION_ROOT)\lib\win32\cvl**

Table 2. Project settings for frame grabber projects in Visual C++ .NET

Configuration	Folder	Category	Item	Settings for Frame Grabber Projects
Debug	General		Output Directory	**.\Debug**
			Intermediate Directory	**.\Debug**
	C/C++	Preprocessor	Preprocessor Definitions	Verify that this macro is defined: **_DEBUG**
		Code Generation	Runtime Library	**Multi-threaded Debug DLL**
Release	General		Output Directory	**.\Release**
			Intermediate Directory	**.\Release**
	C/C++	Preprocessor	Preprocessor Definitions	Verify that this macro is defined: **NDEBUG**
		Code Generation	Runtime Library	**Multi-threaded DLL**
(Debug or Release) Unicode	C/C++	Preprocessor	Preprocessor Definitions	Verify that this macro is defined: **UNICODE**

Table 2. Project settings for frame grabber projects in Visual C++ .NET

Using CVL in DLLs

You can build dynamic link libraries (DLLs) that use CVL routines. To create a standard C++ DLL, perform the following steps. These steps are not intended for use in "Mixed Mode" DLLs, that is, DLLs that mix .NET managed code with unmanaged C++ code.

When using your CVL-based DLLs, it is important that the main routine of your application links against *cogstds.lib* as described in *Static Libraries* on page 39. There are two cases to consider:

CASE A – Your main application (exe) uses CVL and its DLLs use CVL as well. Because your main application uses CVL directly, it will already include *ch_cvl/defs.h*. You can create your own DLL using these two steps:

1. Define the symbol *cmBuildingDLLs* in your DLL project settings. To do this, display the **Property Page** for your project. Select the **C/C++** folder, then the **Preprocessor** category. Add this symbol to the **Preprocessor Definitions** field.
2. If a DLL will instantiate a **ccWin32Display** or **ccDisplayConsole** object, then your calling executable should run **cfInitializeDisplayResources()** in its startup code, as described further in *Calling CVL Display API from Your Own DLL* on page 192.

CASE B – Your main application (exe) does not use CVL but its DLLs do use CVL:

1. Define the symbol *cmBuildingDLLs* in your DLL project settings. To do this, display the **Property Page** for your project. Select the **C/C++** folder, then the **Preprocessor** category. Add this symbol to the **Preprocessor Definitions** field.
2. Link applications that use your DLL with the appropriate version of *cogstds.lib*, which contains CVL initialization code. You can do this in the following ways:
   - Simple method:
     Include *ch_cvl/defs.h* in the main application. This is for the linker, not the compiler. Note that this includes all the lines from the header file. Your application will not need most of these lines, but they should be harmless. If you do not want to include the entire header, use the method below.
   - Other method:
     Add the following preprocessor directives to your main application (Visual C++ .NET 2012 32-bit shown):

```
#ifndef cmBuildingDLLs
#pragma comment(linker, "/include:_cgLoadHardwareSupport")
#pragma comment(linker, "/include:_cgYankCVTShutdown")
#ifdef _DEBUG
#ifdef _UNICODE
#pragma comment(lib, "cogstds11ud.lib")
#else
#pragma comment(lib, "cogstds11d.lib")
#endif
```

```
#else
#ifdef _UNICODE
#pragma comment(lib, "cogstds11u.lib")
#else
#pragma comment(lib, "cogstds11.lib")
#endif
#endif
#endif
```

If you are using Visual C++ .NET, 2013, append “12” instead of “11” to the `cogstds` basename for each of the library names, calling the following four libraries instead:

```
cogstds12ud.lib
cogstds12d.lib
cogstds12u.lib
cogstds12.lib
```

For Visual C++ .NET, 2015, append “14”.

If you are using Visual C++ .NET, 2012 64-bit, call the following four libraries:

```
cogstds11_x64ud.lib
cogstds11_x64d.lib
cogstds11_x64u.lib
cogstds11_x64.lib
```

See *Static Libraries* on page 39 for a discussion of the purpose of the *cogstds.lib* library.

3. If a DLL will instantiate a **ccWin32Display** or **ccDisplayConsole** object, then your calling executable should run **cfInitializeDisplayResources()** in its startup code, as described further in *Calling CVL Display API from Your Own DLL* on page 192.

Upgrading Visual C++ 6.0 Projects to .NET

Starting with release 6.7, CVL no longer supports Visual C++ 6.0. This section describes what you need to do to upgrade a Visual C++ 6.0 project to one of the supported development environments.

For additional information, see the section *Required Settings for Visual C++ .NET* on page 29, *CVL Libraries for Visual C++ .NET 2012, 2013, and 2015* on page 38, and *Calling MFC Functions from Non-MFC Threads* on page 440.

Using Iterators and Pointers

The behavior of iterators has changed in Visual C++ .NET. There is no longer any implicit conversion from iterators to pointers in Visual C++ .NET. This is not a CVL issue, but is a behavioral change introduced in the Visual C++ compiler between version 6.0 and .NET.

When porting code from Visual C++ 6.0 to .NET, change your code to use method (b) or (c) rather than (a) as shown below to get a pointer to the first element in the vector:

```
void func(vector<ccFoo>& vect)
{
  // Visual C++ .NET does not accept:
  // (a)
  ccFoo* ptr = vect.begin();

  // Both Visual C++ 6.0 and .NET will accept either:
  // (b)
  ccFoo* ptr = &vect[0];
  // or (c)
  ccFoo *ptr = &*vect.begin();
}
```

The advantages of (c) **&*vect.begin()** over (b) **&vect[0]** in the above example are that it:

- Is closer to the original code,
- Works as a replacement whenever any iterator, not just **begin()**, is assigned to a pointer
- Works where **operator[]** is not available, as in lists.

Files Delivered with CVL

This section describes the files that make up the Cognex Vision Library and where you can find them in your installation directory.

Directory Layout

When you install CVL, you can select the drive and subdirectory where you want the *\vision* directory installed. By default, the installer places the *\vision* directory in the top level of your system drive. In this manual, file locations and directory paths are shown without the drive letter.

The CVL installer places the CVL software in the following subdirectories within the directory you select during installation as the *VISION_ROOT* directory (*C:\Vision* by default).

Directory	Contents
bin	DLLs for the CVL software and vision tools Camera Configuration Files (CCFs)
CalPlates	Feature correspondence calibration plate graphics
defs	C++ header files
Doc	CVL online documentation
drivers	Drivers and driver installation programs for supported frame grabbers
lib	C++ libraries and import libraries
sample	Source code and resources for sample programs

Table 3. CVL directory layout

Header Files

The header files that CVL uses are located in the following subdirectories in the *%VISION_ROOT%\defs* directory.

Directory	Description
ch_cog	Contains Cognex header files not intended for general use. Although some Cognex tools use these headers, avoid using them in your own programs. The interfaces in these files may change in future releases.
ch_cvl	Contains most of the public header files used in CVL. Most of the files you need are in this directory.
ch_err	Contains header files that list exception error numbers.
ch_host	Contains header files for operating system and compiler related definitions.

Table 4. Locations of CVL header files

Drivers

Driver installation programs and drivers used by frame grabbers that work with CVL are located in the *%VISION_ROOT%\drivers* directory. If necessary, the CVL installation program adds the appropriate Windows registry entries.

Libraries

For users of all Cognex hardware, the CVL installation program places CVL libraries in %VISION_ROOT%*\lib\win32\cvl* (%VISION_ROOT%*\lib\win64\cvl* in the case of a 64-bit system). These libraries are primarily import libraries for CVL DLLs (see *Dynamic Link Libraries* on page 37), and are used when building CVL applications for all MVS-8000 frame grabbers.

CVL header files include **#pragma** directives that tell the linker which library file to link against. For example, the *ch_cvl/display.h* header file includes the pragma **cmLibrarycogdisp**. This specifies the name of the libraries, *cogdisp*.dll* or *cogdisp*.lib*, in which the display code resides. Similar pragmas, located in every *ch_cvl* header file, are used to help control linking. Thus, with few exceptions, you do not need to specify in your project settings or project property page which library files to use.

Dynamic Link Libraries

CVL uses dynamic link libraries (DLLs) for the vision tools and utility functions of frame grabber vision processing applications. These DLLs are located in the directory *%VISION_ROOT%\bin\win32\cvl* (%VISION_ROOT%*bin\win64\cvl* in the case of a 64-bit system). When you install CVL, the installer adds this directory to the Windows *PATH* environment variable.

For deployment of your vision processing application, these DLLs need to be placed in the search path of the deployment computer. Refer to *Deployment Installation of CVL* on page 508 for more information.

Release and Debug Libraries

CVL provides release and debugging versions of all libraries. The debugging libraries were created with the debugging versions of the Microsoft Visual C++ standard libraries. The release versions of the libraries were created with the Release version of the Visual C++ libraries.

Note Mixing debugging libraries with release libraries may cause your vision processing application to crash. Use one set or the other, but not both.

See Table 6 on page 38 for examples of release and debug library file names.

ANSI and Unicode Libraries

You can use CVL in either ANSI or Unicode mode and all libraries are provided in both ANSI and Unicode versions. The Unicode versions of CVL libraries use the Unicode versions of MFC. For more information about using Unicode with CVL see *Localization Issues* on page 54.

CVL library names use the suffix scheme shown in Table 5 to denote the purpose of the library.

Appended letter	Library purpose
none	ANSI 32-bit release
_x64	64-bit release
d	ANSI debugging
u	Unicode release
ud	Unicode debugging

Table 5. CVL library naming scheme

Table 6 on page 38 shows example library names to clarify this naming scheme.

The libraries actually installed may vary with different CVL releases and with options specified during installation.

CVL Libraries for Visual C++ .NET 2012, 2013, and 2015

CVL libraries intended for use with Visual C++ .NET 2012 have "11" appended to their names, in addition to the Unicode and debugging letters described in the previous section.

In the same way, CVL libraries intended for use with Visual C++ .NET 2013 have "12" appended to their names and CVL libraries intended for use with Visual C++ .NET 2015 have "14" appended to their names, in addition to the Unicode and debugging letters.

For example, the DLL that supports image acquisition with Visual C++ .NET 2013 is named *cogacq**12**.dll* and the one for Visual C++ .NET 2015 is named *cogacq**14**.dll* (*cogacq**12**_x64.dll* and *cogacq**14**_x64.dll* in the case of a 64-bit system, respectively).

Table 6 (and Table 7 for 64-bit) shows the names of the ANSI, Unicode, debug, and release versions of DLLs that support image acquisition for Visual C++ .NET 2012, Visual C++ .NET 2013, and Visual C++ .NET 2015. Other CVL DLLs follow the same naming convention.

Executables and DLLs for both development environments are located in the *%VISION_ROOT%\bin\win32\cvl* directory (and in the *%VISION_ROOT%\bin\win64\cvl* directory in the case of a 64-bit system).

Libraries are located in the *%VISION_ROOT%\lib\win32\cvl* directory (and in the *%VISION_ROOT%\lib\win64\cvl* directory in the case of a 64-bit system).

DLL Type	Visual C++.NET 2012	Visual C++.NET 2013	Visual C++.NET 2015
ANSI release version	*cogacq11.dll*	*cogacq12.dll*	*cogacq14.dll*
ANSI debug version	*cogacq11d.dll*	*cogacq12d.dll*	*cogacq14d.dll*
Unicode release version	*cogacq11u.dll*	*cogacq12u.dll*	*cogacq14u.dll*
Unicode debug version	*cogacq11ud.dll*	*cogacq12ud.dll*	*cogacq14ud.dll*

Table 6. Example file names of CVL DLLs (32-bit)

DLL Type	Visual C++.NET 2012	Visual C++.NET 2013	Visual C++.NET 2015
ANSI release version	*cogacq11_x64.dll*	*cogacq12_x64.dll*	*cogacq14_x64.dll*
ANSI debug version	*cogacq11_x64d.dll*	*cogacq12_x64d.dll*	*cogacq14_x64d.dll*
Unicode release version	*cogacq11_x64u.dll*	*cogacq12_x64u.dll*	*cogacq14_x64u.dll*
Unicode debug version	*cogacq11_x64ud.dll*	*cogacq12_x64ud.dll*	*cogacq14_x64ud.dll*

Table 7. Example file names of CVL DLLs (64-bit)

See also *Upgrading Visual C++ 6.0 Projects to .NET* on page 33.

Static Libraries

Most of CVL is implemented in DLL format, but there is a small block of startup and shutdown initialization code implemented as the static library *cogstds.lib*. CVL ships with twelve 32-bit versions of *cogstds.lib*, covering the ANSI, Unicode, release, and debug versions for the three supported compilers (and twelve 64-bit versions), as described in the previous section.

The CVL initializations performed by *cogstds.lib* must take place before any (CVL) DLLs are loaded, and is therefore linked statically to all CVL applications. This linking takes place automatically by means of a **#pragma** statement in *ch_cvl\defs.h*, or with code like that shown in *Using CVL in DLLs* on page 32.

The CVL initialization code added to your application by *cogstds.lib* is small but essential, and must be present in your CVL-based main application.

Samples

The *%VISION_ROOT%\sample\cvl* directory contains sample code you can use to learn CVL. See the section *Programming Examples* on page 41 to learn how to use these programs.

The sample code provided with CVL has four build modes: Win32 Debug, x64 Debug, Win32 Release, and x64 Release. For all sample projects, Debug configuration will build using multibyte character set, while Release configuration will build using Unicode character set. To be able to build these sample projects, Microsoft Visual C++ must be installed. To build Debug configuration using Visual Studio 2013, the Visual C++ “MFC” multibyte character set support must be installed.

If “MFC” multibyte support is not present, you can build the samples in one of the Unicode modes (Win32 Release or x64 Release). To change the build mode of the solution, use the **Configuration Manager**.

See also *Unicode and ANSI Mode Interoperability* on page 57.

Programming Examples

CVL provides both sample application projects and single file code samples to help you learn how to use CVL.

- *Sample application projects* are small standalone vision applications that illustrate CVL programming issues. You can use these projects as the basis for your own vision application.
- *Single file code samples* are code examples contained in a single *.cpp* file, each designed to illustrate a particular feature of CVL. These are used, one at a time, with the project framework provided by *cvlmain.cpp*, as described in *Using the Single File Code Samples* on page 43.

  The complete list of single file code samples shipped with your CVL release is found in the *Getting Started* manual.

Using the Sample Application Projects

The following are some of the sample applications included with CVL. Some samples may not be shipped with CVL versions that do not support platforms required to run those samples.

- Image acquisition sample programs
  - Acquire images from an externally triggered camera
  - Acquire RGB images from a color camera
  - Perform master-slave acquisition using multiple analog cameras
  - Create a live-display loop application
- Display sample programs:
  - Color live display sample
  - Console display sample
  - Win32 display sample

Image Acquisition Sample Projects

The image acquisition sample programs are small standalone applications each built with its own project file. These sample programs are found in the *%VISION_ROOT%\sample\cvl\acquire* directory.

There are multiple versions of each sample program: one for Visual C++.NET 2012, one for Visual C++.NET 2013, and one for Visual C++.NET 2015. The samples for Visual C++.NET 2012 have names ending in “11”, the samples for Visual C++.NET 2013 have names ending in “12”, and the samples for Visual C++.NET 2015 have names ending in “14”.

These sample programs include:

- A collection of helper classes and subroutines that you can reuse to enhance the acquisition functionality of your own applications (in the files *acqutil.cpp* and *acqutil.h*).
- An auto trigger application that demonstrates how to avoid hanging a CVL application in the absence of triggers or other acquisition-related problems. Start with the Visual C++ project file *autotrig.dsp*.
- A live display application that demonstrates how to write a small live display loop using a custom trigger model. Start with the Visual C++ project file *livedisp.dsp*.
- A general application that demonstrates how to perform acquisition in a synchronized FIFO configuration. This sample also demonstrates how to share a single strobe light between multiple cameras. Start with the Visual C++ project file *ms_acq.dsp*.
- A color display application that demonstrates how to create three acquisition FIFOs in a master-slave-slave relationship in order to acquire the RGB planes of a color camera. It shows a simple implementation for combining the three color planes into a single displayable color image. This application was designed for the MVS-8504.

CVL also supplies examples of image acquisition code in the form of single file code samples. The sample files that illustrate image acquisition include the following. (See the *Getting Started* manual for your CVL release for a list of that release’s single file code samples.)

- The *acqbasic.cpp* file demonstrates the canonical, recommended way to perform image acquisition in CVL.

Display Sample Projects

Sample applications that demonstrate image display are in separate directories under the *%VISION_ROOT%\sample\cvl\display* directory.

There are multiple versions of each sample program, one for Visual C++.NET 2012, one for Visual C++.NET 2013, and one for Visual C++.NET 2015. The samples for Visual C++.NET 2012 have names ending in “11”, the samples for Visual C++.NET 2013 have names ending in “12”, and the samples for Visual C++.NET 2015 have names ending in “14”.

Color Live Display Sample

The color live display sample illustrates how to perform live color display and grey-scale image acquisition through a single color camera. This sample is designed to work with color GigE Vision cameras. The color live display sample is located in the *%VISION_ROOT%\sample\cvl\display\clrlive* directory. Start with the Visual C++ project file *clrliveNN.sln* (where *NN* corresponds to the Visual Studio version number).

Console Display Sample

The console display sample combines all the basic elements of a typical vision processing application: it acquires images, uses a vision tool to analyze the image, and displays results. This sample application works on all MVS-8000 frame grabbers. The console display sample is located in the *%VISION_ROOT%\sample\cvl\display\ConsDisplay* directory. Start with the Visual C++ project file *sampleappNN.sln* (where *NN* corresponds to the Visual Studio version number).

Win32 Display Sample

This sample demonstrates how to write Win32 display code in CVL, and specifically, how to use **ccWin32Display** in an MFC-created window. This sample is located in the *%VISION_ROOT%\sample\cvl\display\w32samp* directory. Start with the Visual C++ project file *w32sampNN.sln* (where *NN* corresponds to the Visual Studio version number).

Using the Single File Code Samples

The single file code samples consist of various files that illustrate how to work with different parts of CVL. All of the sample code files use the same framework. That is, the code in each sample file is defined as the function **cfSampleMain()**. The wrapper file *cvlmain.cpp* sets up the frame grabber hardware, then calls **cfSampleMain()**. Thus, each individual sample is built in a project containing exactly two source files: *cvlmain.cpp*, and *one* of the sample code files listed in the *Getting Started* manual.

The wrapper file *cvlmain.cpp* is part of a VIsual C++ project that builds applications for MVS-8000 frame grabbers. The projects are described in Table 8:

Directory in %VISION_ROOT%\sample\cvl	Start by opening
cvlproj11	Project file *cvlproj11.vcxproj* in Visual C++ .NET 2012
cvlproj12	Project file *cvlproj12.vcxproj* in Visual C++ .NET 2013
cvlproj14	Project file *cvlproj14.vcxproj* in Visual C++ .NET 2015

Table 8. Sample code wrapper projects

First Build of the Single File Sample Code

Follow this set of instructions to build and run the single file sample code the first time after installing CVL.

1. Make sure your Visual C++ environment is set up according to the instructions in *Configuring your Development Environment* on page 27.
2. Choose one of the following options depending on which development environment you are using.
   - To build one of the sample code files with Visual C++ .NET 2012, open the project file *%VISION_ROOT%\sample\cvl\cvlproj11\cvlproj11.vcxproj*.
   - To build one of the sample code files with Visual C++ .NET 2013, open the project file *%VISION_ROOT%\sample\cvl\cvlproj12\cvlproj12.vcxproj*.
   - To build one of the sample code files with Visual C++ .NET 2015, open the project file *%VISION_ROOT%\sample\cvl\cvlproj14\cvlproj14.vcxproj*.
3. As shipped, the sample projects contain the source files *cvlmain.cpp* and *disp.cpp*.
4. Build the sample project for the first time by selecting **Build->Build Solution**.
5. Run the sample project by selecting **Debug**->**Start Without Debugging**.
6. Save the project.

Subsequent Builds of Single File Sample Code

Follow the instructions below to build further samples by replacing individual sample code files in the project.

1. Open your saved project.
2. In the Solution Explorer, expand the **Source Files** folder.

   In the **Source Files** list, select the sample file you used when you last saved the project. (The first time you build the sample project, this file will be *disp.cpp*.) Delete your last-used file from the project.
3. Add one of the sample code files listed in the *Getting Started* manual for your release of CVL to the project. Be sure to add *only one* of the sample code files at a time. For example, add *blob.cpp*.
4. Rebuild the project.
5. Run this sample code project.
6. Repeat steps 2 through 5 to build the next project with the next sample code file you wish to use.

Single File Sample Code Shipped with CVL

The sample code files can be found in *%VISION_ROOT%\sample\cvl*. A list of the single file sample files shipped with the current edition of CVL can be found in that edition's *Getting Started* manual.

CVL Programming Conventions

This section describes CVL programming conventions including the use of parameterized classes, pointer handles, the Standard Template Library, CVL naming conventions, and Cognex built-in types and constants. This section also covers localization issues and the use of Unicode.

Parameterized Classes and Functions

Although many CVL classes are parameterized (template classes), most of them are not designed so that you can make your own instantiations. CVL provides instantiations of its own parameterized classes for the types that it supports.

For example, the **ccPelBuffer** class is a parameterized class, so that while it is theoretically possible for you to specialize **ccPelBuffer** with your own type, in practice you can use only those instantiations provided in CVL such as **ccPelBuffer<c_UInt8>**.

You can make your own instantiations of any Cognex class that includes a *.cpp* file in the *%VISION_ROOT%\defs\ch_cvl* directory, such as *handle.cpp* and *pair.cpp*. Of course, you can always make your own instantiations for classes defined in the Standard Template Library and your own parameterized classes. See also *Standard Template Library Usage* on page 52.

Pointer Handles

The most useful parameterized classes in CVL are the pointer handle classes **ccPtrHandle** and **ccPtrHandle_const**.

Pointer handles are reference-counted pointers to heap-allocated objects. In general, they work the same way as pointers do in C++, but have the additional benefit that the object pointed to maintains a count of how many pointer handles are pointing to it. The pointer handle classes use this count, called the *reference count*, to manage the heap memory occupied by the object. When the reference count reaches zero, the object is automatically deleted and its memory returned to the heap.

Several CVL classes use pointer handles rather than pointers to objects to ensure that the underlying objects are deleted properly. For example, the acquisition subsystem uses pointer handles to acquisition FIFOs to ensure that FIFOs are deleted when they are no longer needed by any acquisition routines.

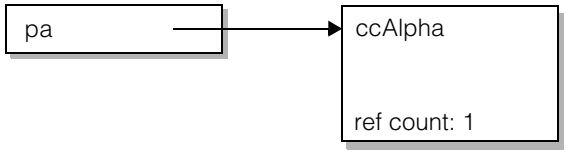
Note Pointer handles are not thread safe. At any given time, only one thread may use a given handled object or any pointer handle to the object.

Overview of Pointer Handles

The following example illustrates how pointer handles work. Assume that **ccAlphaPtrh** is a pointer handle class that points to objects of class **ccAlpha**.

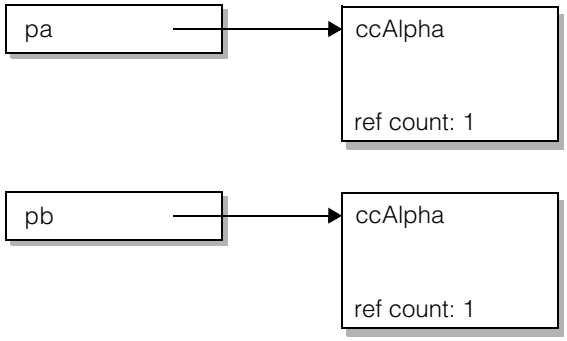
First, the pointer handle *pa* points to a newly created object of class **ccAlpha**. Pointer handles can point only to objects created on the heap:

```
►ccAlphaPtrh pa = new ccAlpha;
```



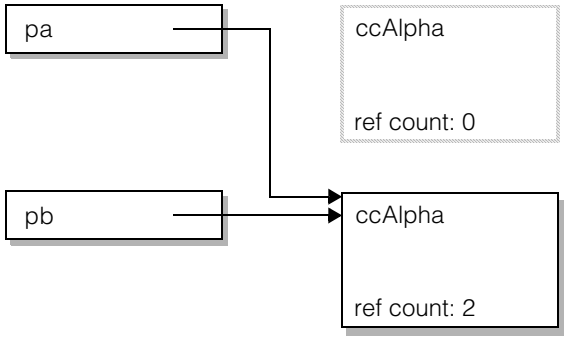
Next, the pointer handle *pb* points to another **ccAlpha** object:

```
  ccAlphaPtrh pa = new ccAlpha;
►ccAlphaPtrh pb = new ccAlpha;
```



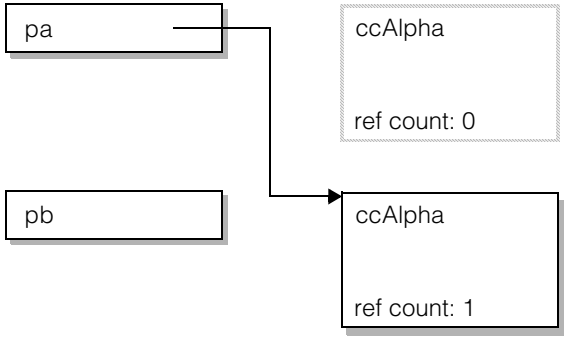
Now, the pointer handle *pa* now points to the same **ccAlpha** object as *pb*. The reference count of the object that *pb* points to is incremented to two; the reference count of the object that *pa* used to point to is decremented to zero, so it is automatically deleted.

```
  ccAlphaPtrh pa = new ccAlpha;
  ccAlphaPtrh pb = new ccAlpha;
▶ pa = pb;
```



Setting *pb* to zero (or NULL) decrements the object's reference count. *pa* is still pointing to the object so its reference count is one, not zero.

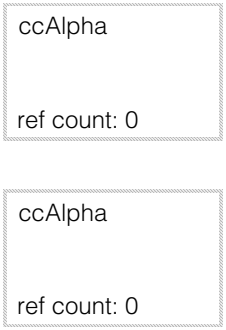
```
  ccAlphaPtrh pa = new ccAlpha;
  ccAlphaPtrh pb = new ccAlpha;
  pa = pb;
▶ pb = 0;
```



Finally, when *pa* goes out of scope, the reference count of the object that it points to is automatically decremented. If no other pointer handle is pointing to the object, it is automatically deleted as well. In other words, if the reference count goes to zero, the object is deleted.

```
 {
  ccAlphaPtrh pa = new ccAlpha;
  ccAlphaPtrh pb = new ccAlpha;

  pa = pb;
  pb = 0;
►}
```



As you can see, one of the main advantages of using pointer handles over regular handles is that pointer handles ensure that the memory occupied by the object is deleted automatically when it is no longer referenced.

Using Pointer Handles

If you are using Cognex-supplied pointer handles, all you need to do is declare a variable of the appropriate pointer handle type when a function returns a pointer handle. For example, **ccStdVideoFormat::newAcqFifo()** returns a pointer handle to a FIFO:

```
const ccStdVideoFormat& fmt =
        ccStdVideoFormat::getFormat(cmT("Sony XC75 640x480"))
ccStdGreyAcqFifoPtrh fifo = fmt.newAcqFifo(fg);
```

Both the pointer handle class (**ccStdGreyAcqFifoPtrh**) and the class it points to (**ccStdGreyAcqFifo**) have been defined for the pointer handle system to work correctly.

You can use pointer handles in much the same way you use regular pointers. You can dereference data members or member functions as you would with a regular pointer:

```
fifo->properties().timeout(10.0);
```

However, unlike regular pointers, the following operations do not work with pointer handles:

- Pointer arithmetic

  If *ph* is a pointer handle, the following expression is illegal:

  ```
  *(ph + 10) = ...;
  ```

- Array operators

  If *ph* is a pointer handle, the following expression is illegal:

  ```
  ph[10] = ...;
  ```

If you need to use either of these constructs, use the **rep()** (representation) function to obtain the regular pointer first. For example, if *p* is a pointer to the same class that *ph* points to, you could write the following:

```
p = ph.rep();
*(p + 10) = ...;
```

Note that the object returned by **rep()** is still controlled by the pointer handle classes. Never delete the object returned by **rep()**.

Defining Pointer Handle Classes

You can use the CVL pointer handle classes to create pointer handles to your own application-defined classes. Your pointer handles will have the same reference-counting benefits as the Cognex-defined pointer handle classes.

Simple Pointer Handle Classes

The following procedure shows you how to define pointer handle classes to your own classes.

1. Define the rep class.

   The class that the pointer handle points to is called the *rep class*. Define your rep class as you would any other class, but make **ccRepBase** one of its base classes:

   ```
   class ccAlpha : public virtual ccRepBase {
     ...
   };
   ```

2. Define the pointer handle.

   To define a pointer handle to your rep class, use the **ccPtrHandle** and **ccPtrHandle_const** parameterized classes:

```
ccPtrHandle<ccAlpha> pFoo;
ccPtrHandle_const<ccAlpha> pcFoo;
```

These pointer handles behave as if they had been declared

```
ccAlpha *pFoo;
const ccAlpha *pcFoo;
```

3. Allocate the rep object on the heap.

   Remember that the rep object (the object that the pointer handle points to) must be allocated on the heap. You cannot use pointer handles to point to stack based objects:

```
ccPtrHandle<ccAlpha> pFoo = new Foo; // OK

ccAlpha myFoo;
ccPtrHandle<ccAlpha> xFoo = &myFoo; // Illegal
```

Pointer Handle Classes for Derived Classes

If your rep class has as its base class a class derived from **ccRepBase**, use this procedure to define pointer handle classes. This is the procedure used for most of the Cognex-supplied pointer handle classes.

Assume that class **ccAlpha** is defined as in the preceding example and that class **ccBeta** is derived from **ccAlpha**:

```
class ccBeta : public ccAlpha {
  ...
};
```

1. Define **const** and **non-const** pointer handle types for the superclass **ccAlpha**.

```
typedef ccPtrHandle<ccAlpha> ccAlphaPtrh;
typedef ccPtrHandle_const<ccAlpha> ccAlphaPtrh_const;
```

2. Use the **cmDerivedPtrHdlDcl** macro to declare the pointer handle types for **ccBeta**.

```
cmDerivedPtrHdlDcl(ccBetaPtrh, ccBeta,
                   ccAlphaPtrh, ccAlphaPtrh_const);
```

   The macro defines the pointer handle types **ccBetaPtrh** and **ccBetaPtrh_const** that point the rep class **ccBeta**.

Standard Template Library Usage

CVL makes use of the C++ Standard Template Library (STL). In particular, it makes extensive use of the vector, list, map, and string template classes. To learn more about the C++ Standard Template Library, consult one of the following books:

- *C++ Programmer's Guide to the Standard Template Library* by Mark Nelson (IDG Books Worldwide, Inc.) ISBN 1-56884-314-3
- *STL Tutorial and Reference Guide* by David R Musser and Atul Saini (Addison-Wesley) ISBN 0-201-63398-1
- C*++ Programming Language, Third Edition* by Bjarne Stroustrup (Addison-Wesley) ISBN 0-201-88954-4

CVL Naming Conventions

The first two characters of most CVL identifiers help you determine their purpose. CVL uses the following naming conventions:

Example	Description
cfFunctionName	A global function
ccClassName	A class name
cmMacroName	A macro name
ckConstantName **k**ConstantName	A constant defined with **const**
ceEnumeration **e**Enumeration	An enumerator (enumeration constant) defined within an **enum**

Table 9. CVL identifier naming conventions

CVL uses case in identifier names as follows:

Example	Description
Upper	An enumerated type defined with **enum**
lowerUpper	A member function

Table 10. Use of case in CVL identifier names

CVL reserves all identifiers beginning with c*xU* where *xU* is any lower case letter followed by an upper case letter or underscore.

Cognex Integer Types

CVL uses the following integer types (defined in *<ch_cvl/inttypes.h>*) rather than the built-in C++ types:

Type	Description	Minimum Value Maximum Value
c_UInt8	8-bit unsigned integer (unsigned char)	**ckMinUInt8** (0) **ckMaxUInt8** (255)
c_UInt16	16-bit unsigned integer (unsigned short)	**ckMinUInt16** (0) **ckMaxUInt16** (65,535)
c_UInt32	32-bit unsigned integer (unsigned long)	**ckMinUInt32** (0) **ckMaxUInt32** (4,294,967,295U)
c_UInt64	64-bit unsigned integer (unsigned __int64)	**ckMinUInt64** (0) **ckMaxUInt64** (18,446,744,073,709,551,615U)
c_Int8	8-bit signed integer (signed char)	**ckMinInt8** (-128) **ckMaxInt8** (127)
c_Int16	16-bit signed integer (short)	**ckMinInt16** (-32,768) **ckMaxInt16** (32,767)
c_Int32	32-bit signed integer (long)	**ckMinInt32** (-2,147,483,648) **ckMaxInt32** (2,147,483,647)
c_Int64	64-bit signed integer (__int64)	**ckMinInt64** (-9,223,372,036,854,775,808) **ckMaxInt64** (9,223,372,036,854,775,807)
c_Bool	integer	**ceFalse** (0) **ceTrue** (1)
c_FFloat	float	

Table 11. CVL integer types

Cognex-Defined Constants

CVL uses the following constants (defined in *<ch_cvl/math.h>*):

Constant	Meaning
ckPI	π
ck2PI	Two times π
ckPI_2	π divided by two
ckPI_4	π divided by four
ck1_PI	One divided by π
ck2_PI	Two divided by π
ck2_SQRTPI	Two divided by the square root of π
ckE	e
ckLN2	The natural logarithm of 2
ckLN10	The natural logarithm of 10
ckLOG2E	The base 2 logarithm of e
ckLOG10E	The base 10 logarithm of e
ckSQRT2	The square root of 2
ckSQRT1_2	The square root of one half.

Table 12. CVL constants

Localization Issues

CVL defines several string and character types, macros, and functions for compatibility with both ANSI (8-bit) characters and Unicode (16-bit) characters. CVL functions use and return the CVL-defined types. Your own program can use either CVL-defined types or specific ANSI or Unicode character types.

Your program must be either in ANSI mode (and use the ANSI libraries) or in Unicode mode (and use the Unicode libraries). You cannot mix modes.

Generic String and Character Types

To ensure that your code works correctly in either ANSI or Unicode mode, use the Cognex-defined generic character types.

Use the `cmT()` macro for strings that are passed to any function. This macro ensures that strings are the correct width (ANSI or Unicode). For example:

```
MessageBox(NULL, cmT("This string will always work."));
MessageBox(NULL, "This will not work if Unicode is on.");
MessageBox(NULL, L"This will not work if Unicode is off.");
```

Table 13 lists the Cognex-defined string and character types and their ANSI and Unicode equivalents.

Cognex Generic Type	ANSI Type	Unicode Type
TCHAR	char	wchar_t
ccCvlChar	char	wchar_t
ccTString	std::string	std::wstring
ccCvlString	std::string	std::wstring
ccCvlOStream	std::ostream	std::wostream
ccCvlIStream	std::istream	std::wistream
ccCvlOFStream	std::ofstream	std::wofstream
ccCvlIFStream	std::ifstream	std::wifstream
ccCvlFStream	std::fstream	std::wfstream
ccCvlOstrStream	std::ostrstream	std::ccWOStrStream
ccCvlIStrStream	std::istrstream	std::ccWIStrStream
ccCvlStrStream	std::strstream	std::ccWStrStream
ccCvlOStringStream	std::ostringstream	std::wostringstream
ccCvlIStringStream	std::istringstream	std::wistringstream

Table 13. CVL string and character types

Setting Up Visual C++ Projects for Unicode

The libraries and DLLs for ANSI and Unicode development have different names but reside in the same directory. You do not need to change any search paths to use one mode or the other.

Most of the sample projects included in CVL are set to build only in ANSI mode. To use Unicode, you will need to modify the sample project files as well as your own project files as follows. The steps shown are for Visual C++ 6.0; the procedure is analogous for Visual C++ .NET.

1. First, create a new configuration for Unicode:
   a. In Visual C++, select **Build->Configurations**.
   b. Click the **Add** button.
   c. Type a name for a new Unicode configuration and click **OK**.
   d. In the Configurations dialog box, click **Close**.
   e. Select **Build->Set Active Configuration**.
   f. Select the new Unicode configuration and click **OK**.
2. Next, define the `UNICODE` and `_UNICODE` preprocessor symbols:
   a. Select **Project->Settings**.
   b. Click the **C/C++** tab.
   c. Add the following symbols to the **Preprocessor definitions** text box:

      ```
      UNICODE, _UNICODE
      ```

      When these symbols are defined, Visual C++ uses the Unicode versions of the CVL DLLs. These DLLs have the letter "u" appended to their names, as discussed in *ANSI and Unicode Libraries* on page 37.
3. Next, choose the entry point for Unicode:
   a. While still in the **Project Settings** dialog box, click the **Link** tab.
   b. From the **Category** drop-down list, select **Output**.
   c. In the **Entry-point symbol** field, type the appropriate entry point for Unicode:
      - For a GUI Unicode application, the entry point is `wWinMainCRTStartup`.
      - For a console application, the entry point is `wmainCRTStartup`.
   d. Click **OK**.

4. Finally, set the debugger to display Unicode strings:
   a. Select **Tools->Options**.
   b. Click the **Debug** tab.
   c. Make sure the **Display unicode strings** check box is checked.
   d. Click **OK**.

Unicode and ANSI Mode Interoperability

CVL archives and CDB files may be written to and read by Unicode-mode and ANSI-mode applications. For information on how Unicode and ANSI strings are handled by the persistence system, see the section *Persisting Strings* on page 474.

Localizing CVL Resources

Some software components of CVL (such as display console menus, the text console, and PatMax diagnostics) use text that is visible to end users of your applications. CVL loads the text for these components from resource files. You can edit these resources to localize them for a particular language.

Table 14 lists resource files and the localizable resources they contain.

Resource file	Localizable resources
cogacqENU.dll	Console and image window menus for use during acquisition **Note:** This file contains no resources that are used or supported, and therefore does not need to be localized.
cogdiagENU.dll	Diagnostics Console menus, dialogs, and strings
cogdispENU.dll	Display Console menus, dialogs, and strings
cogptmxENU.dll	PatMax diagnostic strings
cogstdENU.dll	Cognex standard output (*cogOut*) console strings and menu items

Table 14. Localizable CVL resource files

In each file name, the letters *ENU* represent the two-letter ISO Standard 639 language code for English (*EN*) plus an additional sublanguage code for US English (*U*).

Note Cognex supplies only US English versions of localizable DLLs. Also note that these DLLs do not contain the error messages associated with CVL exceptions.

How to Localize the Resources

To localize resource files, you must copy each Cognex-supplied English-language DLL to a file with a suffix that identifies the locale. For example, to localize resources for Japanese, the resource file names would end in *JPN*; to localize them for Mexican Spanish, the file names would end in *ESM*.

1. Optionally, you can begin by listing the localizable resources. The resource files are in the directory *%VISION_ROOT%\bin\win32\cvl*. For example, you might use the **dir** command with an argument of **ENU.dll* to get a list of all localizable resources in this directory.
2. Copy the localizable files, naming them appropriately for the target language. For example, to localize resources for standard Japanese, replace the three-letter code ENU with JPN, the code for Japanese, in the target file names:

   ```
   copy cogacqENU.dll cogacqJPN.dll
   copy cogdiagENU.dll cogdiagJPN.dll
   copy cogdispENU.dll cogdispJPN.dll
   copy cogptmxENU.dll cogptmxJPN.dll
   copy cogstdENU.dll cogstdJPN.dll
   ```

   You can use the Windows function **GetLocaleInfo()** (using *LOCALE_SABBREVLANGUAGE* as the value of the *LCType* parameter) to determine the three-letter language identifier for any supported language.
3. Next, edit the dialogs, menus, strings, and other resources in the new DLLs. The steps below are shown are for Visual C++ 6.0; the procedure is analogous for Visual C++ .NET.
   a. From the **File** menu in Microsoft Visual C++, select **Open**.
   b. In the **Files of type** drop-down list, select **Executable Files (.exe;.dll;.ocx)**.
   c. In the **Open As** drop-down list, select **Resources**.
   d. Select a resource file and click **Open**.
   e. Double-click each dialog to open an editor.
   f. Double-click each item in the editor to open a dialog in which you can localize the item.

Note There must be a complete set of resource files for each language. For example, even if you want to localize only *cogstdENU.dll* to Japanese (*cogstdJPN.dll*), you must make copies of the other resource files and name them with the JPN suffix even if you do not localize them. If the function that changes languages, **cfSetLanguage()**, cannot find a complete set, it will throw an error.

How to Use the Localized Resources

In your application, call the CVL function **cfSetLanguage()** to specify which resource DLLs should be loaded. To load the Japanese resources, for example, write the following:

```
cfSetLanguage(MAKELANGID(LANG_JAPANESE, SUBLANG_NEUTRAL));
```

You can call this function any time to change languages while your program is running. This function throws a *BadParams* exception if it cannot find the localized version of all of the DLLs. CVL uses the default US English DLLs until you call **cfSetLanguage()**.

Your application must be built using the Unicode versions of the CVL libraries as described earlier in this section. If the language you are using requires a different font, you must provide it. To use a font with the CVL display console, you must specify the font with **ccWin32Display::fontTable()**.

For more information about language codes and localization, see *National Language Support* in the *Windows Base Services* section of Microsoft's *Platform SDK* documentation and the Microsoft Win32 documentation for the **MAKELANGID()** macro.

Header Files Suppress Warnings 4786 and 4251

The header file *<ch_cvl/defs.h>* includes two **#pragma** directives that disable Visual C++ compiler warnings 4251 and 4786, both of which can be safely ignored. All other CVL header files include *defs.h* as the first **#include**, so these warnings are disabled when you include any CVL header file.

You can suppress these compiler warnings in your own code as follows:

- Include *<ch_cvl/defs.h>* as your first #include.
- Include any CVL headers before your own headers.
- Use **#pragma** directives to disable the warnings in your *.h* or *.cpp* files.

Acquiring Images: Basics

2

This chapter describes how to use the Cognex Vision Library to acquire images.

This chapter has the following major sections:

Some Useful Definitions defines terms you will encounter as you read this chapter.

Overview of Image Acquisition presents and explains a short sample program that performs an image acquisition.

Acquiring Images in CVL describes in detail the steps necessary to acquire grey-scale images.

Simultaneous (Master-Slave) Acquisition describes how to use synchronous configurations to acquire images simultaneously in a master-slave configuration.

Using Callback Functions describes how to register a function you write to be called automatically by the acquisition engine at certain points in the image acquisition cycle.

General Recommendations gives some general advice about working with acquisition FIFOs.

Useful Techniques shows you how to determine whether a frame grabber is present, how to determine whether a video format is supported, how to determine which properties a FIFO supports, and how to create CVL pel buffers from non-CVL image data.

Some Useful Definitions

The following definitions may be useful while reading this chapter.

acq image An object that holds an acquired image of any image format. Acq image methods can construct pel buffers containing the acquired image in a specific format.

acquisition FIFO A CVL object that maintains a FIFO of acquisitions. Typically, you create one acquisition FIFO for each camera connected to a frame grabber.

acquisition properties A set of parameters that control the way a camera works or the way that a camera interacts with its acquisition FIFO.

automatic triggering A method that begins image acquisition when a pulse is detected on a trigger line. Also called hardware triggering.

Camera Configuration File (CCF) A text file that contains information used by the acquisition system to create a video format object for a particular camera. Not all frame grabbers and CVMs use CCFs. For CVL versions earlier that 6.0, CCFs are located in a directory named in the environment variable *COGNEX_CCF_PATH*. For CVL 6.0 and later, CCF files are automatically located by the acquisition system.

Cognex Video Module (CVM) A circuit board, part of Cognex vision processors and some frame grabbers, that provides the interface to different kinds of cameras. CVMs can be built into the baseboard or factory-installed as daughter cards. CVMs are specified by a number, such as CVM4.

display console A window used to display an image stored in a pel buffer.

encoder An positional encoding device that communicates the progress of a subject past a line scan camera to a Cognex Video Module (CVM).

FIFO A first-in first-out list. Items are removed from a FIFO in the same order in which they were added.

frame grabber Hardware that digitizes images, and makes them available to software. In CVL, the objects that represent frame grabbers describe the capabilities of the hardware and the video formats the hardware supports.

image format Describes how image data is formatted in memory. For example, grey scale (8 bits/pixel) and 3-plane RGB. CVL-supported image formats are listed in Table 22 on page 94 and are enumerated in the **ccAcqImage** reference page.

line scan camera A camera that acquires an image a single row of pixels at a time.

manual triggering	A method that begins image acquisition when a function is invoked. Also called software triggering.
pel buffer	An array of pixel values that represent an image. In CVL, these are objects of type **ccPelBuffer**. Pel buffers are templated to hold various image formats.
singleton objects	Singleton objects are objects where the creation and destruction of instances are managed by the object itself, and are usually limited to a single instance of the object. For example, if you have two MVS-8514s installed in your system, each will be represented by a singleton object instantiated from the **cc8504** class.
trigger event	A signal on a trigger line or a function invocation that starts an image acquisition.
trigger line	A dedicated hardware input line on the frame grabber used to signal the acquisition software to start an image acquisition in response to an external event.
video format	A description of camera type, image size, and image depth. In CVL, the objects that represent video formats are used to create acquisition FIFOs.

Overview of Image Acquisition

Regardless of the image acquisition hardware base, CVL uses a common image acquisition algorithm. Understanding how CVL acquires images is an important part of understanding how to use the acquisition API to obtain the best acquisition performance. Figure 1 provides an overview of how an acquired image is moved from a camera to a frame grabber, and then to a pel buffer in system memory.

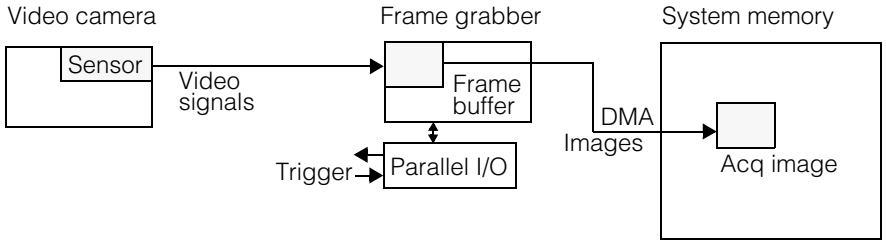


Figure 1. Acquiring an image, frame grabber example

An image is captured by the sensor of a video camera. Each sensor pixel stores an analog voltage proportional to the light falling on the sensor at that location. Most cameras transmit the sensor array pixel-by-pixel serially as an analog signal in a raster scan format to a Cognex-supported frame grabber. When an image is completely acquired in the frame buffer, the image is transferred to a pel buffer in system memory through the processor DMA channel. Once you setup the acquisition, from the time an acquisition is started until the image is in system memory no intervention is required from your application program.

Digital cameras perform the analog-to-digital conversion inside the camera and output a digital signal that can be sent directly to system memory through an adapter. This method is described by the GigE specification. Figure 2 shows a GigE configuration.

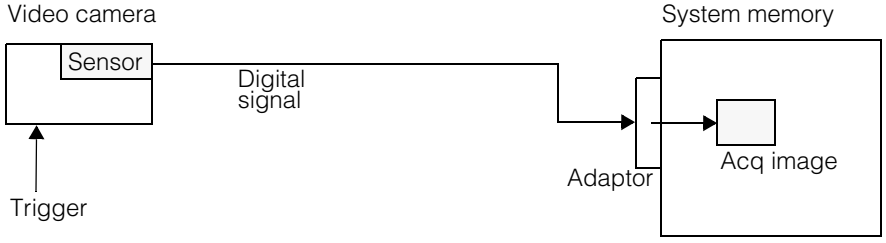


Figure 2. Acquiring an image, GigE example

To acquire images as described above, you must create a FIFO object in your program. For example, you create an object of type **ccAcqFifo**. You configure the FIFO for the camera and format you are using and acquire images by calling its **start()** method. You also call the FIFO **completeAcq()** method to initiate processing the acquired image.

start() and **completeAcq()** should always be issued in pairs, one pair for each acquired image. Multiple starts can be queued within the FIFO. Only one FIFO is required to process a stream of acquisitions from the same camera. For a multiple camera application you will generally create one FIFO for each camera.

A queued **start()** invokes an acquisition engine thread that manages image acquisition. Each acquisition FIFO is associated with its own engine thread and each engine thread executes the following steps for each acquisition:

ACQ0: Wait for engine queue requests
ACQ1: Gain access to hardware.
ACQ2: Set up hardware for next acquisition.
ACQ3: Wait for trigger (hardware trigger only) and acquire image.
ACQ4: Release hardware for use by other FIFOs.

Figure 3 is a simplified model of the image acquisition engine subsystem.

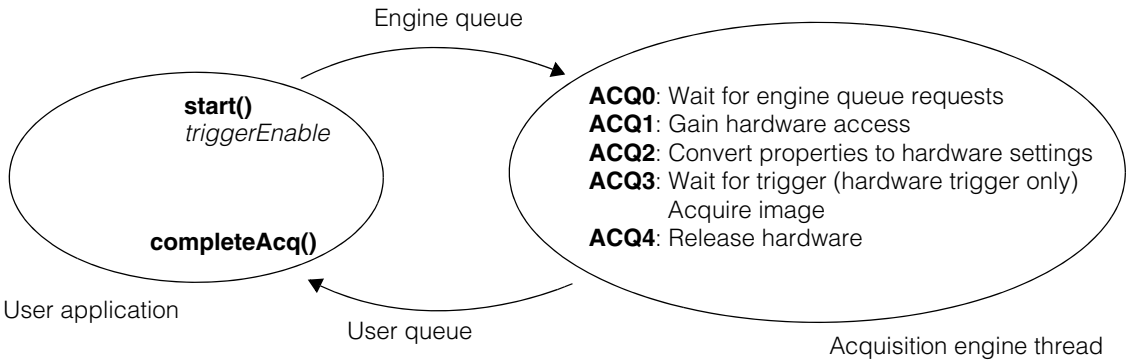


Figure 3. CVL acquisition subsystem

Each FIFO contains two internal queues, the *engine queue* and the *user queue*. The engine queue contains acquisition requests for which images have not been acquired. The user queue contains acquisition requests for which image acquisition is completed, but the **completeAcq()** method has not been called.

The following paragraphs provide a short description of each engine thread state.

ACQ0
Requests in this state are ready for acquisition and are held in the engine queue. In manual trigger mode and semiautomatic trigger mode, *triggerEnable* plus **start()** places a request in the engine queue. In automatic trigger mode, *triggerEnable* alone places the request in the engine queue.

ACQ1
In this state, the request waits to gain exclusive control of non-shared hardware resources in the frame grabber. If the current FIFO request uses the same resources as

the previous request, no hardware allocation wait is required. If the current FIFO request uses different resources, it will block in **ACQ1** waiting for the previous request to clear from the FIFO engine so that it can take control of the frame grabber hardware resources. This hardware allocation state allows acquisition hardware to be shared among multiple acquisition FIFOs.

ACQ2
In this state, the FIFO converts properties into hardware settings. The time required to perform this step depends greatly on the previous state of the hardware. The first acquisition is typically the worst case, where the FIFO must wait for the camera and frame grabber hardware to synchronize video timing. If the timing is already synchronized, and there were no property changes since the last acquisition, **ACQ2** can take very little time.

ACQ3
In this state, the image is actually acquired. For manual triggers, acquisition begins immediately. For auto and semi auto triggers, the thread waits for an external trigger signal to begin the acquisition.

If external triggering is used, and a trigger occurs before the FIFO is ready to transition to **ACQ3**, an error is generated.

ACQ4
After the acquisition is completed, **ACQ4** releases the FIFO's claim to the hardware, allowing the hardware to be used by other FIFOs. For performance reasons, the FIFO may elect not to release the hardware during **ACQ4**. This speeds up **ACQ1**, but more importantly precludes other FIFOs from claiming or making changes to hardware that would slow down **ACQ2**.

Note The FIFO provides a *triggerEnable* property that enables or disables the engine thread. When *triggerEnable* is set to false, the engine thread immediately aborts any acquisition in progress and terminates. Additional requests will not be processed until the engine thread is restarted by setting *triggerEnable* to true. For more information on *triggerEnable*, see *Enabling Triggers* on page 84.

Acquisition Example

There are seven basic steps, and one optional step, to acquire images using CVL.

1. Get a reference to the frame grabber object.
2. Select a video format for the camera connected to the frame grabber.
3. Create an acquisition FIFO using the video format and frame grabber information.
4. Disable triggers.
5. Optionally, set any required properties of the acquisition FIFO.

6. Set the trigger model and enable triggers. See also *Trigger Models* on page 101.
7. Start the acquisition manually by invoking a function (**start()** only) or automatically using an external input line (trigger).
8. Complete the acquisition and get a pel buffer that contains the acquired image.

The following code illustrates these steps. The sections that follow discuss each of these steps in more detail.

Note

The following example is designed to illustrate several techniques in a small amount of space. In an actual vision application, use only those techniques that are appropriate for your application. See *General Recommendations* on page 117.

Starting with CVL 6.0, you must #include *acq.h*, *prop.h*, and *vidfmt.h* explicitly.

```
#include <ch_cvl/vp8100.h>
#include <ch_cvl/callback.h>
#include <ch_cvl/acq.h>
#include <ch_cvl/prop.h>
#include <ch_cvl/vidfmt.h>
#include <ch_cvl/constrea.h>

class MyPartMover: public ccCallback
{
  virtual void operator()(); // override
};

// Define your callback function as an override of operator()
// of your callback class.
void MyPartMover::operator()()
{
  // Place code here to signal your part mover thread that it
  // is ok to move the part. Do not perform lengthy operations
  // or call CVL functions in a callback function.
}

int cfSampleMain(int, TCHAR** const)
{
  // Step 1: Get a frame grabber (see also Figure 4 on page 68)
  cc8100m& fg = cc8100m::get(0);

  // Step 2: Select a video format
  const ccStdVideoFormat& fmt =
       ccStdVideoFormat::getFormat(cmT("Sony XC-ST50 640x480"));
```

```
// Step 3: Create an acquisition FIFO
ccAcqFifoPtrh fifo = fmt.newAcqFifoEx(fg);

// Step 4: Disable triggers
fifo->triggerEnable(false);

// Step 5: Set the FIFO properties. (See the Using
// Callback Functions section of this chapter for
// an alternate way to register your callback function.)
fifo->properties().exposure(20e-3);
fifo->properties().movePartCallback(new MyPartMover);

// Step 6: Set the trigger model and enable triggers.
// cfManualTrigger() is the default trigger model setting.
// Thus, the next line is included for illustration.
fifo->triggerModel(cfManualTrigger());
fifo->triggerEnable(true);

// Step 7: Acquire images.
//         The initial start() gets the acquisition
//         engine's queue started, which speeds up overall
//         acquisition.
fifo->start();
for (int i = 0; i < 10; ++i)
{
  ccAcquireInfo info;

  // Step 8: Get a pel buffer with the image.
  ccAcqImagePtrh img = fifo->completeAcq(
     ccAcqFifo::CompleteArgs().acquireInfo(&info));
  ccPelBuffer<c_Uint8> pb = img->getGrey8PelBuffer();

  // Start the next acquisition.
  fifo->start();

  if (info.failure())
  {
    cogOut << cmT("Acquisition failed. ")
           << (info.failure().isMissed()      ?
               cmT("Reason: missed trigger.")  :
               info.failure().isAbnormal()    ?
               cmT("Reason: abnormal failure."):
               cmT("Unknown reason."))
           << std::endl;
    continue;
  }
```

```
    // Process this image.
    cogOut << cmT("Acquired image size: ")
           << pb.width() << cmT("x") << pb.height() << std::endl;

    // pb goes out of scope here, freeing the pel buffer and the
    // acquired image.
  }

  // FIFO goes out of scope here, destroying the ccAcqFifo
  // object, terminating any remaining outstanding acquisitions.
  return 0;
}
```

Classes representing Cognex hardware all derive from the **ccBoard**, **ccFrameGrabber**, and **ccParallelIO** classes. (Remember that for GigE applications, the camera is the frame grabber.)

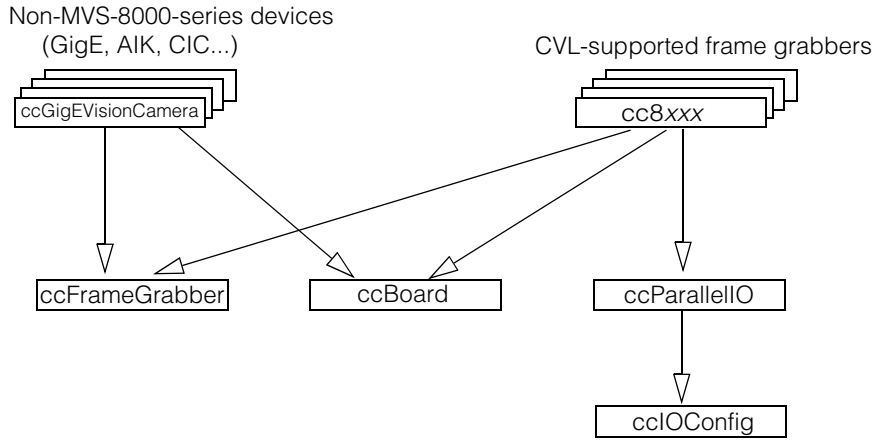


Figure 4. Class diagram of MVS-8000 series and other hardware classes

Step 1 of the preceding program addresses a specific type of board (an MVS-8100M) by instantiating one of the derived frame grabber classes (**cc8100m**) and getting a reference to the MVS-8100M frame grabber with **cc8100m::get()**. You can make your application more generic so that it runs with any frame grabber by instantiating a **ccFrameGrabber** instead of a class representing a specific type of board, and getting a reference to this generic frame grabber with **ccFrameGrabber::get()**.

Acquisition Throughput

This section describes acquisition throughput performance. Keep in mind the following basic guidelines:

- The rate at which you can acquire images is ultimately limited to the rate at which you can process the acquired images. The time required to process an acquired image includes the time required to perform any image processing, apply any vision tools, perform any computations, and take any required actions.
- Displaying images takes time. This time must be included in the total image processing time.

In addition to these guidelines, there is also an ultimate limit to the rate at which the acquisition hardware and software can process acquisition triggers.

Always verify the status of each image acquisition. You do so by supplying a **ccAcquireInfo** object when you call **completeAcq()**.

Your application can experience two main types of throughput-related acquisition failures:

- Failures caused by the trigger rate exceeding the amount of time required to process acquired images.

  If the rate at which acquisition triggers are received by your application exceeds the amount of time required to process images, eventually your system will exhaust image storage. When this happens, the error reported is platform dependent, as follows:

Platform	Error
MVS-8600	No error reported. These platforms cannot detect a missed trigger.
MVS-8500	**ccAcqFailure::isMissed()**

- Failures due to a trigger rate higher than the ability of the hardware to process triggers.

  When this happens, the **ccAcqFailure** or **ccAcquireInfo::failure()** will indicate an **isMissed()** error on most frame grabbers.

For more information on the **ccAcqFailure** object and **ccAcquireInfo::failure()**, see the pages for these classes in the *CVL Class Reference* as well as *Determining Why an Acquisition Failed* on page 98.

Acquisition FIFO and Image Throughput

The acquisition FIFO resides in host PC memory and different Cognex frame grabbers provide different amounts of onboard frame buffer memory. Most Cognex frame grabbers use the frame buffer memory as a temporary buffer for each acquired image on its way to the acquisition FIFO. These frame grabbers do not need to read the entire image from the camera before transferring it to the host PC. After exposure is complete, the camera sends the image line by line to the frame grabber which starts transferring lines to the acquisition FIFO as soon as they are received. For example, line 1 can be transferred from the frame grabber to the PC while line 2 is on its way from the camera to the frame grabber. The frame grabber continues transferring lines in this manner until the entire image is transferred to the acquisition FIFO.

When each image arrives in the FIFO, the vision application is notified that an image is available. If the application is not ready for the image, the image remains in the FIFO until the application removes it by calling **completeAcq()**.

For all acquisition FIFOs, the acquisition engine does not queue up more than 32 images per FIFO, regardless of image size, pel pool size, frame grabber image buffer size, or the amount of available host PC memory. (This number is set as **ccAcqFifo::kMaxOutstanding**.) Once there are 32 images in the acquisition FIFO, your application must deal with the first image in the FIFO before a 33rd image can be acquired. Your application need not process the image, but the image must be removed from the FIFO by calling **completeAcq()** and stored in host PC memory or discarded, for image acquisition to continue. Your application can call **ccAcqFifo::completedAcqs()** to determine how many completed acquisitions are currently being held by the FIFO.

As long as your application can continue shuffling images into and out of the FIFO in this fashion, the limit on the number of images acquirable depends only on the amount of available host PC memory. However, once host PC physical memory is exhausted, the overall acquisition speed can no longer run at the full frame rate of the camera-frame grabber combination.

For most applications, if you find the acquisition FIFO's 32 image buffering capacity is consistently exhausted, your application must either slow down the trigger rate or increase the rate at which images are retrieved and processed.

The exception is an application that acquires a short burst of images at high speed, as might be done for motion analysis. For this kind of application, images can be acquired into a pel buffer array, as shown in this example:

```
enum { kBuffers=150 };
ccPelBuffer<c_UInt8> pb[kBuffers];
for (imgNum=0; !stopRequested() && imgNum < kBuffers;
      ++imgNum)
{
  // Leave the image where it currently is.
  ccAcqImagePtrh img = fifo->completeAcq();
  if (img)
    pb[imgNum] = img->getGrey8PelBuffer();
}
```

Acquiring Images in CVL

This section describes in detail each of the steps involved in acquiring images.

Getting a Frame Grabber Reference

Before your application can acquire images, it needs to know what kind of hardware you are using. CVL uses the term *frame grabber* for any hardware that can be used to digitize images. CVL defines a class for each frame grabber it supports and automatically creates an instance of these frame grabber classes for each frame grabber it detects in the system. You obtain a frame grabber object to work with by calling **get()**.

You can first call the static **count()** function for the frame grabber you wish to use to make sure one is installed in your system. For example,

```
c_Int32 num8100m = cc8100m::count();
```

If *num8100m* = 0, no MVS-8100M frame grabbers are installed in your system. If there are one or more, reference them by index number starting with 0. The following code obtains a frame grabber object for the first (index = 0) MVS-8100M frame grabber in your system.

```
cc8100m& fg = cc8100m::get(0);
```

You can then use this frame grabber reference in conjunction with a video format to create an acquisition FIFO, as shown in the following section and in the example program on page 66.

Generally frame grabber boards are assigned index numbers according to the motherboard slot in which they are installed. Index 0 is assigned the lowest numbered slot, index 1 to the next higher numbered slot, and so on. However, the numbering of PCI slots varies with motherboard manufacturers and can be BIOS dependent. Consult your system hardware documentation to verify the correct frame grabber index numbers to use when more than one frame grabber is installed in your system. If you have only one frame grabber, it will always be index 0.

For another technique for getting a frame grabber reference, see *Testing for a Frame Grabber* on page 119.

Selecting a Video Format

The next step is to select the *video format* that describes the camera connected to your frame grabber and its characteristics, including the size of the image to acquire. The function **ccStdVideoFormat::getFormat()** returns a reference to the video format whose name you provide.

This is how the example program gets a video format for a Sony XC-75 camera:

```
const ccStdVideoFormat& fmt =
   ccStdVideoFormat::getFormat(cmT("Sony XC-ST50 640x480"))
```

CVL creates a single instance of a **ccStdVideoFormat** for each supported video format. If the name of the video format name that you passed to **getFormat()** is not a valid name, it throws *ccVideoFormat::NotFound*.

Video formats are either static formats built in to the CVL code or are available though Camera Configuration Files (CCFs). Some frame grabbers can use only built-in video formats, while others use CCF-based video formats. For CVL versions prior to 6.0, CCFs are located in the directory specified by the environment variable *COGNEX_CCF_PATH*. For CVL 6.0 and later versions, CCFs are automatically located by the acquisition system (usually in the directory containing *cogacq.dll*).

Included with your CVL documentation is a complete list of the video formats supported by each frame grabber for your release of CVL. CCF-based formats have names that end in "CCF," while built-in static formats do not.

Note Earlier revisions of CVL used global functions, such as **cfXc75_640x480()**, to generate references to video formats. In order to ensure compatibility with older programs, those functions are still available for older cameras. However, in new programs, or when you update older programs, use **ccStdVideoFormat::getFormat()** to get references to video formats.

Creating an Acquisition FIFO

Once you have obtained a video format, use its **newAcqFifoEx()** member function and the reference to the frame grabber to create an acquisition FIFO:

```
const ccStdVideoFormat& fmt =
     ccStdVideoFormat::getFormat(cmT("Sony XC-ST50 640x480"))

ccAcqFifoPtrh fifo = fmt.newAcqFifoEx(fg);
```

This code creates an acquisition FIFO suitable for acquiring images that are 640 pixels wide by 480 pixels high with a Sony XC-ST50 camera.

CVL uses acquisition FIFOs (**ccAcqFifo**), to manage image acquisitions. For some frame grabbers you can select the image format using an argument to **newAcqFifoEx()**. Each acquisition FIFO is associated with a particular frame grabber. The acquisition FIFO queues requests to acquire images. Depending on the trigger model you specify, the acquisition can begin as soon as all the hardware components are ready, or the acquisition may be deferred until the frame grabber receives a signal on an input line. See *Trigger Models* on page 101 for more information on triggers.

The **newAcqFifoEx()** member function of the video format object returns a specific class of acquisition FIFO specialized for that format. If a frame grabber does not support a particular video format, **newAcqFifoEx()** throws *ccVideoFormat::NotSupported*.

Use Pointer Handle Classes

In the code example starting on page 65 and the code fragment at the beginning of this section, you will notice that the variable *fifo* is of type **ccAcqFifoPtrh** rather than **ccAcqFifo***.

Pointer handles are reference-counted pointers to objects; they have the same semantics as regular pointers in your code. Since pointer handles are reference counted, it is not necessary to explicitly delete the acquisition FIFO when you use pointer handles. More than one pointer handle can point to the same acquisition FIFO. When the last pointer handle pointing to an acquisition FIFO is deleted or goes out of scope, the reference count for the FIFO becomes zero, which forces the acquisition FIFO to be automatically deleted. That is why it is not necessary to explicitly delete acquisition FIFOs when using pointer handles.

For more information, see *Pointer Handles* on page 46.

To release a FIFO pointed to by a pointer handle, use the following:

```
fifo = 0;
```

Cognex **strongly** recommends that you use the pointer handle classes instead of pointers to the FIFO for all acquisition FIFOs.

Warning *Use of raw pointers for acquisition FIFOs may result in undefined behavior in certain situations. Always use pointer handles for acquisition FIFOs.*

For convenience, you can combine two steps, getting a video format and creating the acquisition FIFO, into one:

```
ccAcqFifoPtrh fifo(ccStdVideoFormat::getFormat(
                  cmT("Sony XC-ST50 640x480")).newAcqFifoEx(fg));
```

Note We recommend that you do not use pointer handles as static objects. Doing so typically leads to problems with static destruction. If you must use a static pointer handle, be sure to set it to 0 before program termination. For example:

```
#include <ch_cvl/acq.h>
ccAcqFifoPtrh theFifo;
ccAcqImagePtrh img;
main ()
{
    theFifo = fmt.newAcqFifoEx(fg);
    // use fifo
    img = theFifo.completeAcq();
    img = 0;
    theFifo = 0;
}
```

Setting FIFO Properties

Set FIFO properties by first obtaining a handle to a properties object using the **ccAcqFifo::properties()** method, and then calling the setter for that property. For example:

```
fifo->properties().cameraPort(0);
fifo->properties().exposure(20e-3);
fifo->properties().contrastBrightness(0.5, 0.5);
```

The first line selects the frame grabber port where the camera is connected that will be used to acquire images. Port numbers are numbered sequentially starting with 0.

The second line sets the camera exposure time to 20 msec.

The third line sets the frame grabber contrast and brightness. This controls how light values are mapped to pixel values as the image passes through the frame grabber.

Each property class defines pairs of member functions that let you set or get its values. Some properties, such as the trigger property, define additional member functions that let you specify finer aspects of the property.

Table 15 provides a description of each of the acquisition FIFO property classes.

Property Class	Description
cc8BitInputLutProp	Lets you specify a lookup table (LUT) that maps raw input pixel values to your own 8-bit values.
ccCameraPortProp	Controls which camera port the acquisition FIFO takes images from. Port numbering is zero-based.

Table 15. Acquisition property classes

Property Class	Description
ccCompleteCallbackProp	Allows you to register a function that will be called when the acquisition is completed. See also **ccAcqFifo::completeInfoCallback()**.
ccContrastBrightnessProp	Controls the contrast and brightness of analog cameras.
ccCustomProp	Allows you to create a collection of custom property-value pairs and custom commands. These custom commands are applied to the acquisition device before each acquisition. This class is supported for GigE and Imaging Device Adapter acquisition.
ccDigitalCameraControlProp	Allows you to control properties of digital cameras.
ccEncoderControlProp	Provides functions to control and match the specifications of a hardware position encoder connected to CVM11.
ccEncoderProp	Controls frame grabber-specific devices for interfacing to an external position encoder on CVM11.
ccExposureProp	Sets the shutter speed for shuttered cameras. For strobed applications, informs the acquisition system about the duration of the strobe flash. The default value is appropriate for strobing, and may need to change when using ambient lighting.
ccFirstPelOffsetProp	Allows you to specify the offset of the first image pixel from the value supplied by the camera's device driver. This property is used with platforms that support line scan cameras.
ccGigEVisionTransportProp	Allows you to control the transport-related properties for GigE Vision cameras.
ccMovePartCallbackProp	Allows you to register a function that will be called when the image has been integrated but before the acquisition is complete. See also **ccAcqFifo::movePartInfoCallback()**.

Table 15. Acquisition property classes

Property Class	Description
ccOverrunCallbackProp	Allows you to register a function that will be called if the acquisition failed because it was unable to start the acquisition in a timely fashion even though the acquisition system was able to obtain the required resources. See also **ccAcqFifo::overrunInfoCallback()**.
ccPelRootPoolProp	Allows you to specify how root image buffers are allocated.
ccRoiProp	Allows you to specify a region of interest (ROI) when acquiring images from certain cameras. For line scan cameras, the ROI property lets you refine the height and width of the acquired image.
ccSampleProp	Specifies the subsampling applied to the ROI of acquired images. Image x-direction and y-direction subsampling are performed independently.
ccStrobeDelayProp	Specifies the strobe delay associated with an acquisition FIFO. Strobe delay is the delay in seconds between the shutter pulse and the strobe firing.
ccStrobeProp	Controls whether a signal is sent on the strobe output line associated with the current camera port, when acquisition starts.
ccTimeoutProp	**Note**: This property is deprecated and is maintained for backward compatibility only.
ccTriggerFilterProp	Specifies the input trigger properties: trigger width, trigger period, and trigger delay. Can be used to tailor the frame grabber's response to trigger signals so that false trigger-like signals are rejected.
ccTriggerProp	Controls whether acquisitions are triggered manually with a function call or by an electrical signal on the trigger input line.

Table 15. Acquisition property classes

Some properties apply to FIFOs for all frame grabbers, while others apply only to specific frame grabbers. Table 16 shows a matrix of properties and frame grabbers. The dots in the matrix indicate that the property is supported by that frame grabber hardware.

Although this table shows properties supported by frame grabbers, some cameras used with a given frame grabber might not support every specified property.

	8500	8600	GigE Vision	Imaging Device
cc8BitInputLutProp	•	•		
ccCameraPortProp	•	•		
ccCompleteCallbackProp	•	•	•	•
ccContrastBrightnessProp	•		•	•
ccCustomProp			•	•
ccEncoderProp		•		
ccExposureProp	•	•	•	•
ccGigEVisionTransportProp			•	
ccMovePartCallbackProp	•	•	•	•
ccOverrunCallbackProp	•			
ccPelRootPoolProp	•	•		
ccRoiProp	•	•	•	•
ccSampleProp	•			
ccStrobeDelayProp	•	•		
ccStrobeProp	•	•		

Table 16. Acquisition properties

	8500	8600	GigE Vision	Imaging Device
ccTimeoutProp[1]	•	•	•	•
ccTriggerFilterProp[2]	•	•		
ccTriggerProp	•	•	•	•

Table 16. Acquisition properties

Note 1: **ccTimeoutProp** is deprecated and is maintained for backward compatibility only.

Note 2: Only the following methods of **ccTriggerFilterProp** are supported: **triggerDelay()** on 8500 and 8600 boards and **ignoreMissedTrigger()** on 8600 boards.

You can also test a FIFO to see if it supports a particular property, as discussed in *Testing for Properties* on page 121.

If you need to change property settings during the course of your application, be sure to read *Changing Properties During Acquisition* on page 103.

To make working with properties easier, CVL provides the **ccAcqProps** class, which inherits from all other FIFO property classes. This allows you to set or get any FIFO property by accessing a single object. Figure 5 shows the properties class hierarchy.

Figure 5. Class diagram of acquisition FIFO property classes

Note Earlier versions of CVL used two classes, **ccAcqProps** and **ccAnalogAcqProps**, to define FIFO properties. All of the acquisition properties are now contained in **ccAcqProps**, and **ccAnalogAcqProps** has been deprecated.

The **properties()** member function for **ccStdGreyAcqFifo**, **ccStdRGB16AcqFifo**, and **ccStdRGB32AcqFifo** returns a **ccAcqProps** object that enables you to work conveniently with properties that apply to acquired images.

Lookup Tables and Clipping

Several FIFO properties allow you to change the pixel values of acquired images in ways that may benefit your application. These properties include the **cc8BitInputLutProp**. Each frame grabber supports only one of these properties. Table 17, a subset of Table 16 on page 78, summarizes the frame grabber support for the **cc8BitInputLutProp** property.

	8500	8600
cc8BitInputLutProp	•	•

Table 17. Clipping and LUT properties (subset of Table 16)

This property allows you to adjust and constrain pixel values in the acquired image to a mid-range by clipping (or clamping) high and low pixel values at some threshold. Pixel values less than the lower threshold are set to the lower threshold, and pixel values higher than the high threshold are set to the high threshold. This property also allows you to apply an offset to all pixels before any clipping is done. The resulting image is then passed on to the pel buffer. This property used for this purpose is discussed in the following paragraphs.

Figure 6 shows graphically how the acquired image pixel values are changed if both the offset and clamp are applied.

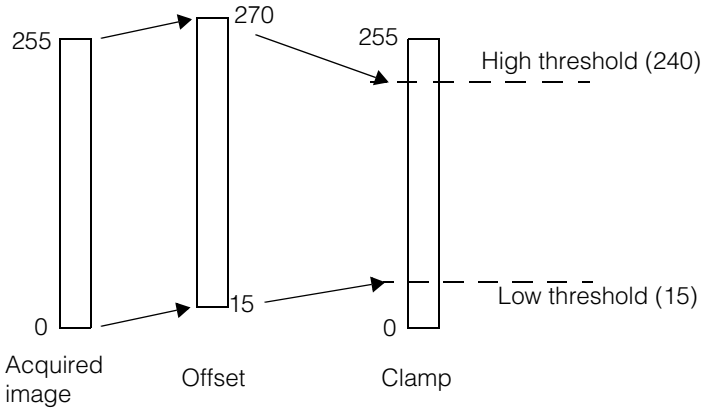


Figure 6. Offset and clamp

After the offset is applied, all pixel values are increased by 15. There are no pixel values in the range 0 through 14. After the clamp, all pixels are in the range 15 through 240.

cc8BitInputLutProp

The **cc8BitInputLutProp** property is supported by most frame grabbers, as illustrated in Table 17 on page 81. The property provides the same offset and clipping functionality as described in Figure 6 above. The **cc8BitInputLutProp** property allows you to provide an offset or an offset plus clipping. The default mode provides clipping only. To change the mode call **cc8BitInputLutProp::inputLut(***mode***)** once you have created your FIFO object. The following values for *mode* are supported:

Mode	Description
positive8BitLut	No offset or clipping is applied. The acquired image is placed in the pel buffer unaltered.
default8BitInputLut	Perform clipping only. Pixel values below 15 are set to 15 and pixel values above 240 are set to 240. This is the default mode.
shifted8BitLut	Perform offset and clipping. An offset of 15 is added to all pixel values and then the pixel values are clipped to the range 15 through 240. Offset pixel values below 15 are set to 15 and pixel values above 240 are set to 240

Starting an Acquisition

This section discusses setting up trigger models and enabling triggers, using the **prepare()** and **start()** functions, and checking the FIFO's current status.

Trigger Models

One of the FIFO properties you set is the trigger property, which specifies how your application initiates image acquisition. With CVL, you can choose between manual triggering, automatic triggering, semi automatic triggering, slave triggering, and free run triggering. With manual triggering (also called software triggering) you call a function to place an acquisition request in the acquisition FIFO. If you use automatic triggering (also

called hardware triggering), the acquisition software automatically places an acquisition request into the acquisition FIFO in order to detect a pulse on an input trigger line. The triggering models are summarized in Table 18.

Trigger type	Also known as	Acquisition started by	Associated global function
Manual trigger	Software trigger	**ccAcqFifo::start()**	**cfManualTrigger()**
Auto trigger	Hardware trigger	External signal	**cfAutoTrigger()**
Semi trigger	n/a	**ccAcqFifo::start()**, then external signal	**cfSemiTrigger()**
Slave trigger	n/a	Same as master	**cfSlaveTrigger()**
Free run trigger	n/a	**ccAcqFifo:triggerEnable()** set to true	**cfFreeRunTrigger()**

Table 18. Trigger models

The relationship between the non-slave trigger models is summarized in Table 19.

	Needs external trigger	Does not need external trigger
Needs start()	Semi	Manual (software)
Does not need start()	Auto (hardware)	Free Run

Table 19. Relationship between trigger models

Regardless of the trigger model you select, **triggerEnable()** must be set to true in order for acquisitions to take place. The following sections discuss setting the trigger model and enabling triggers.

Setting the Trigger Model

The following line of code sets the FIFO trigger model to manual trigger mode:

```
fifo->triggerModel(cfManualTrigger());
```

The trigger model specifies how the acquisition FIFO engine queue receives acquisition requests. In this case, the value is set to **cfManualTrigger()**, meaning that acquisitions are initiated with a function call to **start()**. If your application uses an external trigger on an input line, set this value to **cfAutoTrigger()** instead. A third trigger model, **cfSemiTrigger()**, is a hybrid of automatic and manual triggering. For more information on trigger models see *Trigger Models* on page 101 and **ccTriggerModel** in the *CVL Class Reference*.

Enabling Triggers

Trigger enabling is the on/off switch for the acquisition engine thread. When triggers are enabled, the engine thread begins processing acquisition requests. When triggers are disabled, the engine thread halts any acquisitions in progress and waits for triggers to become enabled before restarting.

When triggers are disabled, external triggers are ignored, producing neither acquisitions nor **isMissed** errors. For trigger models that do not use external triggering, acquisition is also disabled when **triggerEnable()** is false. This is demonstrated by the following code:

```
fifo->triggerEnable(false);
for (i=0; i < 10; ++i)
   fifo->start();
// 10 acquisitions are queued, but will not be acquired until
// some time later, when triggers are enabled
fifo->triggerEnable(true);
```

While disabling triggers halts any acquisition in progress, it does not flush the FIFO. This preserves successfully completed acquisitions in the FIFO. However, consider the following code, where the expected number of acquisitions does not take place:

```
for (i=0; i < 10; ++i)
   fifo->start();

// After some, but not all, acquisitions have completed:
fifo->triggerEnable(false);
// We just halted one of the 10 acquisitions, now resume:
fifo->triggerEnable(true);

// Allow enough time for all acquisitions to complete
ccTimer::Sleep(10.0);

// This will probably display "9"
cogOut << fifo->completedAcqs() << std::endl;
```

When triggers are disabled in the above example, the acquisition in progress is halted and disappears from the FIFO. However, any acquisitions that have not started are left intact and ready to resume when triggers are re-enabled.

When two or more FIFOs are associated in a synchronous (master-slave) relationship, be especially watchful of trigger disabling side effects. See *Considerations When Using Master-Slave Acquisition* on page 109 for details.

The following functions make internal calls to **triggerEnable()**, disabling triggers then re-enabling them before exiting only if triggers were originally enabled.

ccAcqFifo::triggerModel()
ccTriggerProp::triggerMaster()
ccAcqFifo::flush()

The following function disables triggers and then destroys the FIFO object:

ccAcqFifo::~ccAcqFifo()

Using prepare()

The **prepare()** function serves two purposes in the acquisition system:

- To ready the hardware so that the first acquisition will not take extra time compared to subsequent acquisitions.
- To detect hardware setup problems that would prevent successful acquisitions, such as a missing camera or an incorrectly attached camera.

The **prepare()** function returns a Boolean that indicates whether the hardware could be prepared successfully:

- True means that the hardware could be successfully prepared and is ready to perform acquisition.
- False means that there is a problem that will prevent the acquisition from working correctly, such as a missing camera. Do not attempt to acquire an image. If **prepare()** returns false, calling **completeAcq()** results in a timing error or a bad image, but does not hang the application. See also *Using prepare() with Internal and External Drive Formats* on page 87.

The **prepare()** function must be able to access the hardware before it can prepare it. If the hardware is currently in use by the FIFO on which **prepare()** is called, the function returns true immediately. If the hardware is in use by another FIFO, **prepare()** waits until it can access the hardware. **prepare()** takes a single argument that specifies how long to wait to gain access to the hardware before returning false.

Note For releases prior to CVL 6.0, **prepare()** returned false if called on a CVM4 with no camera connected to the port associated with the FIFO. For CVL 6.0 and later, this behavior was expanded to cover CVM1, CVM6, and the MVS-8100D.

The **prepare()** function eliminates overhead associated with the first acquisition that occurs after a property change or FIFO creation. This overhead results from setting up the hardware for the current properties in state **ACQ2**, and from making sure that timing is synchronized between the camera and the frame grabber. After **prepare()** is called, the following steps must still occur in the FIFO before an image can be acquired:

1. The application must call **start()** (for manual and semi trigger modes only).
2. The acquisition request must be sent to the engine thread.

3. The engine thread must receive the acquisition request.
4. The engine thread must allocate hardware resources.
5. The engine thread must verify that the hardware settings are correct.

Depending on platform and processor speed, the above steps may take up to 1 ms.

Keep the following in mind when using the **prepare()** function:

- In most cases, Cognex recommends calling **prepare(0.0)** before enabling triggers for the first acquisition on a FIFO. Triggers should be disabled immediately after creating the FIFO.
- Always use a timeout value of 0.0 when calling **prepare()**. Failure to do so can cause **prepare()** to hang instead of returning an error.
- Call **prepare()** only on an idle FIFO, or on an auto trigger FIFO when triggers are disabled. Failure to follow this guideline results in **prepare()** returning true without having done anything to prepare the hardware.
- Calling **prepare()** and then **start()** never executes faster than just calling **start()** alone. However, calling **prepare()** first makes **start()** perform in a more consistent manner.
- Starting with CVL 6.0, for synchronous configurations you only need to call **prepare()** on the master FIFO. Any slave FIFOs are prepared with the master.
- If multiple FIFOs are sharing hardware such as a camera port, **prepare()** should not be used. In this situation **prepare()** can cause extraneous changes to hardware settings. Consider the following incorrect code sequence:

```
// Assume fifo1 and fifo2 use the same camera
// with different properties

fifo1->prepare(0.0); // set up fifo1 properties

fifo2->prepare(0.0); // set up fifo2 properties,
                     // but inadvertently undo
                     // fifo1->prepare()

fifo1->start(); // set up fifo1 properties,
                // but undo fifo2->prepare()

fifo2->start(); // set up fifo2 properties
```

The exception in this example is that **prepare()** *can* be used to test camera and video format compatibility before acquisition is started on either FIFO.

Note The prepare function can be defeated if another FIFO interacts with the hardware after **prepare()** is called.

Note If multiple FIFOs are sharing an interlaced (free running) camera, they may require up to three fields to switch between FIFOs, rather than the expected one field.

Using prepare() with Internal and External Drive Formats

The **prepare()** function behaves differently depending on whether the hardware it is attempting to prepare uses internal or external timing:

- **Cameras that CVL can detect directly (CVC-1000 and CDC series):** **prepare()** can detect whether a camera of this type is missing or incorrect.
- **Cameras that supply a clock to the frame grabber:** If a camera of this type is not connected, **prepare()** can detect the missing clock. If the wrong camera is connected, it is less likely that **prepare()** is able to detect a problem.
- **Cameras that supply a sync signal that the frame grabber PLL locks to:** This case is harder to detect because the PLL usually runs at the wrong rate if no camera is present. In this case, **prepare()** does a time-consuming check for the presence of the camera and returns false if it is not able to detect one. An attempt to acquire an image skips the check and does not return a timing error unless the PLL is completely stopped.
- **Cameras that receive all timing from the frame grabber:** **prepare()** cannot detect the presence of these cameras.

For internal drive formats in general, the hardware still works and **prepare()** returns true even if no camera is connected. However, **completeAcq()** returns only black images.

Alternatives to prepare()

In some cases, **prepare()** might used where it does not need to be. This section describes an alternative to using **prepare()** for a certain case.

Problem: We want to use **prepare()** to get the hardware ready in advance, hoping to avoid missing the first trigger when using the auto trigger model. We try to use code like the following.

```
fifo->triggerEnable(false);
fifo->triggerModel(cfAutoTrigger());
if (!fifo->prepare(0.0))
    panic();

// start conveyor belt

fifo->triggerEnable(true);
// Possibly miss the first trigger before the
// acq FIFO is ready to acquire.
```

There is a finite amount of time between when triggers are enabled and when the FIFO is ready to accept a trigger. This amount of time can be reduced but not eliminated by using **prepare()** to get the hardware ready. However, the code above does not protect against **triggerEnable(true)** still taking too long, long enough to miss the first trigger.

Solution: Starting in CVL 6.0, you can use the **isAcquiring()** query to determine when the FIFO is ready to accept triggers:

```
fifo->triggerEnable(false);
fifo->triggerModel(cfAutoTrigger());
fifo->triggerEnable(true);

// Wait for FIFO to be ready for triggers
ccTimer watchDog(true);
while (!fifo->isAcquiring() && (watchDog.sec() < 5.0))
    ccTimer::sleep(0.100);

if (!fifo->isAcquiring())
    panic();

// FIFO is now ready, it is safe to start conveyor belt
```

This example, instead of calling **prepare()**, just re-enables triggers after setting the trigger model, then polls until either the FIFO returns true from **isAcquiring()** or 5 seconds have elapsed.

We assume that triggers do not arrive before the conveyor belt is started. If a trigger arrives while polling **isAcquiring()**, the program could sleep through the entire acquisition and the FIFO would return **isComplete()** true, but **isAcquiring()** false. In this case, the second **isAcquiring()** call traps this condition and causes the code to time out and panic.

Using start()

The start method places an acquisition request in the acquisition engine queue. If triggers are enabled and the engine thread is idle, it immediately begins processing the request. If triggers are disabled, the request waits in the queue until triggers are enabled.

Note that there are no guarantees of the state the engine thread is in when **start()** returns. For the semi auto trigger model, this means the hardware may not be ready to accept a trigger immediately after calling **start()**. To determine when the hardware is ready to accept a trigger, use **ccAcqFifo::isAcquiring()**.

Another way to view **start()** is that it commits the FIFO to acquire an image with the current property settings.

Some trigger models such as **cfAutoTrigger()** do not require that you call **start()**. If you call **start()** and the trigger model does not require it, a **ccAcqFifo::StartNotAllowed()** exception is thrown.

Each FIFO is allowed a finite number of outstanding acquisition requests as defined by **ccAcqFifo::kMaxOutstanding** (usually set to 32 requests). If you exceed this number, an **isMissed()** error is returned when you call **completeAcq()**.

You can optionally supply an *appTag* as an argument to **start()**. The **completeAcq()** function returns this tag in its *appTag* argument. Your application can ignore *appTag* completely, or use it to match up completed acquisitions with the **start()** call that requested them. For example, if your application has multiple threads that share a FIFO, each thread can use its thread ID as the *appTag*. When **completeAcq()** is called, the application can use the returned *appTag* to match up each image with the thread that requested it.

Synchronizing start() and completeAcq() Calls

During error free operation, you will issue exactly one **completeAcq()** call for each acquisition **start()**. The acquisition FIFO processes requests in order, so that the first request in is always the first request out. There are some errors, however, that cause **completeAcq()** to return before acquisition is completed. For these cases, you will need to issue a second **completeAcq()** to retrieve the acquired image.

When you call **completeAcq()** with the newer syntax, you pass a **ccAcquireInfo** object, one of whose parameters is a *startReqStatus* pointer, as shown in this example:

```
ccAcqFifo::ceStartReqStatus status;
ccAcqImagePtrh pb = fifo->completeAcq
  (ccAcqFifo::CompleteArgs().startReqStatus(&status));
```

When **completeAcq()** returns, if *status* = *ckStartReqWasServiced*, the acquisition was completed (but may have associated errors). However, if *status* = *ckStartReqNotServiced*, then the acquisition was not completed and you should issue another **completeAcq()** to retrieve the image.

It is possible for *status* = *ckStartReqNotServiced* to occur when using an auto-trigger if you are also using the **ccAcqFifo::CompleteArgs().maxWait()** parameter as a timeout timer, and the timeout expires.

start() takes an *appTag* parameter that is carried through the acquisition engine and is returned with **completeAcq()** as a member of the **ccAcquireInfo** object returned. It is a good design practice to check these tags in your program to make sure the **start()** calls are synchronized with the **completeAcq()** calls. Note that when *status* = *ckStartReqNotServiced*, the **ccAcquireInfo::appTag()** is invalid.

Checking FIFO Status

Although you can place many requests in an acquisition FIFO, you always collect the oldest request on the list when calling **completeAcq()**. This is called the oldest outstanding request. Keep in mind that it is possible for the oldest outstanding request to be in the input engine queue, in the engine thread, or in the completed user queue. **ccAcqFifo** provides several member functions that let you check the status of the oldest outstanding request, and other request activity in the FIFO. Table 20 lists the **ccAcqFifo** functions that describe FIFO activity in general.

Function	Description
isIdle()	Returns true only if the engine queue is empty, the user queue is empty, and the engine thread is idle.
isComplete()	Returns true if at least one outstanding acquisition is complete (perhaps unsuccessfully). If the acquisition completed successfully, you can obtain the image in a pel buffer.
isValid()	Returns true if the FIFO is configured properly to perform acquisitions.
pendingAcqs()	Returns the number of acquisitions in the *pending* state. This is the number of requests in the engine queue.

Table 20. Acquisition FIFO status functions

Function	Description
completedAcqs()	Returns the number of acquisitions in the *completed* state. This is the number of requests in the user queue.

availableAcqs()	Returns an upper bound on the number of additional acquisitions that can be queued. Generally this is **ccAcqFifo::kMaxOutstanding** minus the sum of (pending requests + completed requests).
	
	This method is intended for diagnostic purposes only. Values returned can vary between CVL releases. One possible use is to snapshot the return value after you have created a FIFO and then to check the value later when the FIFO is idle. If the value has gone down, then there is a problem (such as a possible resource leak or old acquisitions remaining in the FIFO).
	
	Do not use the return value for anything other than comparative purposes, for example:
	
	`while(fifo->availableAcqs() > 0)`
	`  fifo->start();`

Table 20. Acquisition FIFO status functions

Table 21 lists functions that describe only the oldest outstanding request.

Function	Description
isWaiting()	Becomes true when the FIFO begins waiting for **start()**, and returns to false when hardware resources are allocated and all setup delays are finished.
isAcquiring()	Returns true if the current outstanding acquisition is waiting for a trigger signal or is acquiring an image. For synchronous acquisitions, it becomes true only after the master and all slaves are ready for acquisition.
isMovable()	Returns true if the object in the camera's field of view can be moved without affecting the image.

Table 21. Oldest outstanding request status functions

The timing of these states as requests move through the FIFO engine are shown in Figure 7 on page 92.

Figure 7 is a simplified acquisition engine state diagram and shows the times during the procedure where **isWaiting()**, **isAcquiring()**, and **isMoveable()** return *true*.

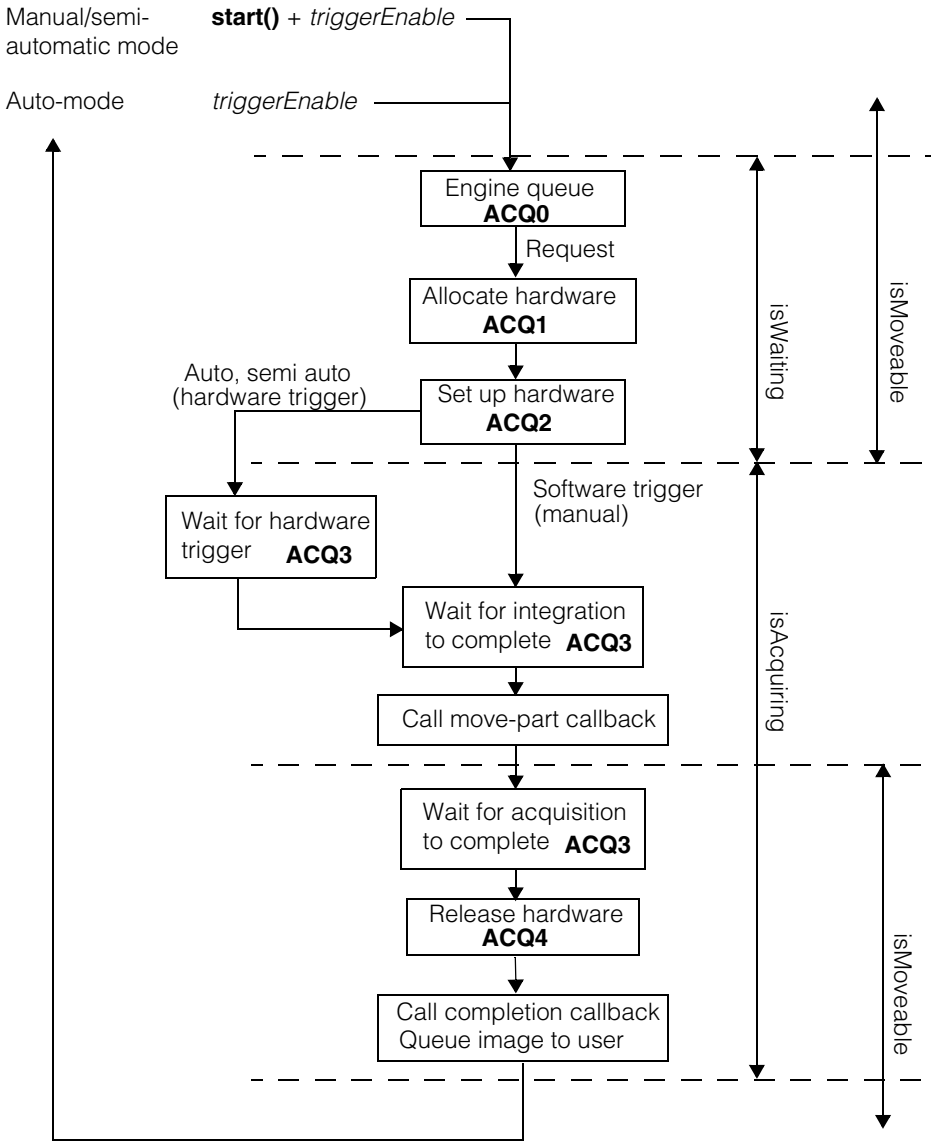


Figure 7. Acquisition engine state diagram

Completing the Acquisition

Once an image has been acquired, you use the acquisition FIFO's **completeAcq()** function to get a reference to the acquired image. The function blocks until the oldest outstanding acquisition completes. The function will always return a **ccAcqImage** but the object may be unbound (may not contain an acquired image) if there was an acquisition error. You should always check for a bound image (**ccAcqImage::isBound()**) before trying to retrieve a pel buffer from a **ccAcqImage**.

Note Earlier releases of CVL used **ccStdGreyAcqFifo::complete()** to retrieve acquired images. While this API is still supported for backward compatibility, the newer **ccAcqFifo::completeAcq()** API should be used for new code.

Creating a Pel Buffer From a ccAcqImage

When you use **ccAcqFifo::completeAcq()** to retrieve an acquired image, the function returns an **ccAcqImage** object that contains the raw image. The image can be of any image format and can be color or grey scale. You call **ccAcqImage** member functions to obtain a pel buffer with the image format you require in your application. For example, you call **ccAcqImage::getGrey8PelBuffer()** to obtain an 8-bit grey scale pel buffer from the acquired image. If the acquired image is an 8-bit grey scale image the image is passed directly in the pel buffer. If the acquired image is not an 8-bit grey scale image, the function converts the acquired image to 8-bit grey scale and returns a pel buffer containing the requested format.

Note If the function does a format conversion the original acquired image is retained in the **ccAcqImage** object and additional conversions can be done on it. However, if the acquired image was passed without conversion, the pel buffer you get contains the actual acquired image. If you modify the pel buffer there is no backup copy of the acquired image.

ccAcqImage provides four functions for obtaining pel buffers in four different image formats as follows:

Member function	Pel buffer format returned
getGrey8PelBuffer()	8-bit grey scale
getPackedRGB16PelBuffer()	16-bit packed RGB
getPackedRGB32PelBuffer()	32-bit packed RGB
get3PlanePelBuffer()	8-bit planer RGB

Only certain conversions from acquisition formats are supported. The supported conversion are specified in Table 22.

Acquired image formats		Convert to: G8	R16	R32	PIr
G8	(ceImageFormat_Grey8)	Yes	No	No	No
G10	(ceImageFormat_Grey10)	Yes	No	No	No
G12	(ceImageFormat_Grey12)	Yes	No	No	No
G16	(ceImageFormat_Grey16)	No	No	No	No
R16	(ceImageFormat_PackedRGB16)	No	Yes	No	No
R32	(ceImageFormat_PackedRGB32)	No	No	Yes	No
B8	(ceImageFormat_Bayer8)	Yes	Yes	Yes	Yes
B16	(ceImageFormat_Bayer16)	No	No	No	No
411	(ceImageFormat_UYYVYY)	Yes	Yes	Yes	Yes
422	(ceImageFormat_UYVY)	Yes	Yes	Yes	Yes
444	(ceImageFormat_UYV)	Yes	Yes	Yes	Yes
R24	(ceImageFormat_PackedRGB24)	Yes	Yes	Yes	Yes
R48	(ceImageFormat_PackedRGB48)	No	No	No	No
PIr	(ceImageFormat_PlanarRGB8)	No	Yes	Yes	Yes

Table 22. Supported image format conversions

Attempts to convert a non-supported format cause an exception error. To determine if a conversion is supported you can call **ccAcqImage::isConversionSupported()**.

Retrieving Acquisition Status

completeAcq() takes a **ccAcqFifo::CompleteArgs** object to specify its completion parameters. To retrieve the results of an image acquisition, you specify, as one of the **CompleteArgs** parameters, a **ccAcquireInfo** object to contain the results.

Default Invocation of CompleteArgs

If you use the default invocation of **completeAcq()** without specifying any parameters, a default **CompleteArgs** object is constructed.

```
ccPelBuffer<c_UInt8> pb = fifo->completeAcq();
   // uses default CompleteArgs object
```

The default parameters for **completeAcq()** are the following:

- **makeLocal()** is true (the pel buffer is copied automatically to PC memory from frame grabber memory, if applicable).
- **maxWait()** is HUGE_VAL (**completeAcq()** waits indefinitely for a completed acquisition).
- **startReqStatus**, the completion status of **ccAcqFifo::start()**, is not stored. For the meanings of the status values, see *Synchronizing start() and completeAcq() Calls* on page 89 and **ccAcqFifo::ceStartReqStatus** in the *CVL Class Reference*.
- No **ccAcquireInfo** object is specified, and thus the success or failure results of the acquisition are not stored.

Specifying CompleteArgs Parameters

There are two ways to specify the arguments to **completeAcq()** using a **CompleteArgs** object, the long form and the short form. Cognex documentation shows the short form in code examples, and recommends using it.

In the long form, you instantiate a **CompleteArgs** object and use its member functions, one at a time, to override the default-constructed value for any argument you want to change, as shown in the following example.

```
ccAcquireInfo info;
ceStartReqStatus status;

ccAcqFifo::CompleteArgs args;

args.acquireInfo(&info);
args.startReqStatus(&status);
args.maxWait(5.0);
args.makeLocal(false);

ccAcqImagePtrh img = fifo->completeArgs(args);
```

The short form is made possible because all **CompleteArgs** member functions return a reference to the **CompleteArgs** object. For this reason, you can chain each member function off the constructor, resulting in a simple list of period-separated **CompleteArgs** arguments, as shown in the following example. White space and line breaks are used for clarity, but are not required. This example accomplishes exactly the same as the preceding example.

```
ccAcquireInfo info;
ceStartReqStatus status;
ccAcqImagePtrh img = fifo->completeAcq
          (ccAcqFifo::CompleteArgs().acquireInfo(&info)
                                    .startReqStatus(&status)
                                    .maxWait(5.0)
                                    .makeLocal(false));
```

The argument chaining makes the named variable *args* unnecessary, and it can be dropped. Using the short form, all function-call arguments to **completeAcq()** are consolidated into a single statement.

As another example, if you want only to specify a **ccAcquireInfo** object, set **maxWait()** to 3.5 seconds, and leave the other arguments in their default values, use code like the following:

```
ccAcquireInfo info;
ccAcqImagePtrh img = fifo->completeAcq
          (ccAcqFifo::CompleteArgs().maxWait(3.5)
                                    .acquireInfo(&info));
```

Using the ccAcquireInfo Object

Starting with CVL 6.1, you can examine or test the results of an image acquisition by specifying a **ccAcquireInfo** object to contain the results. This object is specified as one of the **CompleteArgs** parameters, as described in the preceding section. The former method of specifying a **ccAcqFailure** instance is supported with the older syntax for **completeAcq()**.

The parameters contained in a **CompleteArgs** object are the following:

- The *appTag* of the acquisition being retrieved. The *appTag* is an arbitrary integer you specify with **start(***appTag***)** for each acquisition. You can compare the *appTag* value of each **start()** and **completeAcq()** to make sure you have matching pairs. See *Synchronizing start() and completeAcq() Calls* on page 89.
- The trigger number for this acquisition, stored internally by the acquisition engine, as described in *Using Trigger Numbers* on page 97.
- A **ccAcqFailure** object, whose functions you can use to query the reason for an acquisition's failure.

To specify a **ccAcquireInfo** object in your call to **completeAcq()**, use code like the following:

```
ccAcquireInfo info;
ccAcqImagePtrh img = fifo->completeAcq
          (ccAcqFifo::CompleteArgs().acquireInfo(&info));
```

Thereafter, use member functions of **ccAcquireInfo** to retrieve information about the acquisition. For example, to retrieve the acquisition's *appTag* value:

```
c_UInt32 tag;
tag = info.appTag();
```

Use the **failure()** function to determine the results of the acquisition:

```
if (info.failure())
// acquisition failed
```

You can use any of the **ccAcqFailure** functions with code like the following:

```
if (info.failure().isIncomplete())
// the maxWait time expired
if (info.failure().isOverrun())
// trigger received, but an image could not be acquired
```

See *Determining Why an Acquisition Failed* on page 98 for a list of the **ccAcqFailure** functions.

Using Trigger Numbers

A trigger number is a monotonically increasing integer value that counts the number of triggers processed by the acquisition FIFO. The trigger number count is maintained automatically by the image acquisition engine, to the best of its ability. You do not need to initialize or start the count of trigger numbers. The trigger number count starts over when the acquisition FIFO object is instantiated.

For some applications, using the trigger number provides an easier method of tracking missed triggers than using **failure().IsMissed()** errors. Tracking the trigger number is particularly useful if the trigger model is semi-automatic or custom, where **failure().IsMissed()** errors require extra calls to **completeAcq()** relative to the number of calls to **start()**.

Determining Acquisition Success

If your application uses automatic (hardware) triggering, there are several ways you can determine when the image is ready to be transferred to a pel buffer.

- Call **completeAcq()**
  This function waits until the oldest outstanding acquisition has completed successfully or unsuccessfully.
- Poll using **isComplete()**
  If your application needs to perform some procedures while it is waiting for the image, you can poll **isComplete()** to find out when the image is ready. This technique is simple, but not efficient.

- Register a completion callback function
  You can specify a function that the acquisition engine will call as soon as the image is ready. Your callback function can signal a semaphore to let your application know that the image has been acquired. To learn more about callback functions see *Using Callback Functions* on page 111.

When your program returns from **completeAcq()**, check to see if the acquisition was successful before attempting to use the pel buffer. The following represents the usual way to do this.

```
if (info.failure())
    {
        // Find out what failed
    }
else
    {
        // Successful acquisition, process image
    }
```

Determining Why an Acquisition Failed

There are times when an image acquisition fails, usually because of a timeout or because the acquisition rate was too fast for the hardware. You can use **ccAcqFailure** functions to determine why an acquisition failed. The functions are used as described in *Using the ccAcquireInfo Object* on page 96.

Table 23 lists the acquisition failure functions.

Function	Returns true if...
isTimeout()	The acquisition failed because the timeout period elapsed before the acquisition system was able to obtain the required resources. **Note**: This function has been deprecated and is maintained for backward compatibility only.
isMissed()	The acquisition failed because the acquisition resources were not available at the time the trigger was received. One or more triggers may have occurred between serviced triggers.
isOverrun()	The acquisition failed because even though the acquisition system was able to obtain the required resources, it was unable to start the acquisition in a timely fashion.

Table 23. Acquisition failure functions

Function	Returns true if...
isAbnormal()	The acquisition failed because of a fault in the acquisition hardware or because of some other unusual problem.
isIncomplete()	Returns true if the acquisition failed because the **ccAcqFifo::CompleteArgs.maxWait()** interval expired before the acquisition entered the complete state (**ccAcqFifo::isComplete()** still returns true).
isTooFastEncoder()	The acquisition failed because of a line overrun condition. (Line scan cameras only.)
isInvalidRoi()	The acquisition failed because the region of interest property was invalid. Typically, a region of interest that is too large will generate this error.
isOtherFifoError()	The acquisition failed because of a failure on a master or slave acquisition FIFO. For example, if a FIFO has two slaves and the first slave times out, that slave fails with **isTimeout()** while the master and second slave fails with **isOtherFifoError()**. **Note**: **isTimeout()** is now deprecated and is maintained for backward compatibility only. If you do not use **isTimeout()** in your program, you will not get this error.
isTimingError()	A grievous video timing error prevented the acquisition from occurring. This is commonly caused by attempting to acquire when the camera is not plugged in, or when the video format is not appropriate for the camera. This error will occur only for mismatches that prevent the acquisition hardware from working at all. Other mismatches can cause corrupted images but not cause this error.

Table 23. Acquisition failure functions

Guarding Against Application Deadlocks

There are some circumstances, even in a manual acquisition application, when calling **completeAcq()** can hang your thread or application. If logic errors in your application prevent a call to **start()** or if an unusual hardware error occurs, an otherwise valid call to **completeAcq()** can hang your thread or application.

Calling **completeAcq()** always returns the oldest acquisition request in the FIFO. If there is no image available, then **completeAcq()** will block. If an image never becomes available, your application or thread will hang.

If you are using automatic or semi autotriggered acquisition, then you must verify that an image is available before you call **completeAcq()**. You can do this by calling **ccAcqFifo::isComplete()** or the recommended method, by registering a completion callback, as described in *Using Callback Functions* on page 111.

If you are using manual (software triggered) acquisition, perform an acquisition by calling **start()** to request the acquisition and then call **completeAcq()** to obtain the acquired image.

Because of the possibility of deadlock, Cognex recommends that you *always* verify that an image is available before calling **completeAcq()**. You can also set a finite value for **completeAcq()**'s **maxWait()** parameter and check the result from **ccAcqFailure::isIncomplete()** or **ccAcquireInfo::failure().inIncomplete()**, as appropriate.

Although using **maxWait()** as a timeout timer is one possible design choice, the preferred approach is to leave **maxWait()** set to the default (HUGE_VAL) and the use **ccAcqFifo::isComplete()** to determine when an acquisition is completed.

A thorough application could perform even more elaborate testing for acquisition completion in a separate thread, as shown in the CVL sample code file *%VISION_ROOT%\sample\cvl\display\autotrig.cpp*.

Restarting Acquisition

Follow the restart procedure in this section if your application performs a number of image acquisitions, then disables triggers, then restarts acquisitions on the same FIFO. Follow this procedure anytime you disable triggers after one or more image acquisitions have occurred. Remember that a FIFO may have performed acquisitions if you called **start()**, or if triggers are enabled and the current trigger model does not require **start()**.

To restart disabled triggers on the same FIFO:

1. Use **fifo->triggerEnable(false)** to stop image acquisitions on the FIFO.
2. [Optional] If you wish to make use of the images still in the FIFO, call **fifo->completeAcq()** to retrieve them.
3. Use **fifo->flush()** to flush the FIFO and restore it to a known state.
4. Perform the task for which you disabled triggers.
5. Restart triggers with **fifo->triggerEnable(true)**.
6. If there were any queued **start()** calls pending at the moment you disabled triggers, you must now requeue them. (This can be done before restarting triggers, if desired.)

Trigger Models

This section describes the trigger models that are built into CVL, and describes how to create your own custom trigger model.

Manual Trigger Model

When you use manual triggering, as in the example program on page 63, you acquire images by calling the acquisition FIFO's **start()** function.

```
fifo->start();
```

Calling **start()** places an acquisition request into the engine queue. If there are no pending requests and if the hardware is set up according to the properties, the acquisition takes place immediately. Otherwise, the acquisition request remains in the queue until it can be processed.

Automatic Trigger Model

If you are using automatic triggering, the acquisition request is placed in the engine queue immediately after triggers are enabled (which is the default), and the auto trigger model is selected. When the engine thread accepts the request the hardware is immediately prepared. Following this preparation, the hardware performs an acquisition on the next external trigger. After an external trigger, the next acquisition request is generated automatically in preparation for a subsequent trigger. Each newly generated request uses the latest property values.

If there is a transition on the trigger line before the acquisition can take place, the acquisition is missed and **ccAcquireInfo->failure().isMissed()** returns true.

If you create a new FIFO, specify automatic triggering, then set other properties for the FIFO (in that order), the first few images acquired on that FIFO will be acquired using the default FIFO properties and not the properties you set after specifying automatic triggering.

You can prevent this from happening by disabling triggers after you create the FIFO and before you set the FIFO properties. Remember to enable triggers after setting the properties as the final step.

It is a good practice to call **prepare()** and verify that it returns true before attempting your first acquisition. See the following code:

```
fifo->triggerEnable(false);
fifo->triggerModel(cfAutoTrigger());

// Set properties for acquisition

if (!fifo->prepare(0.0))
throw something; // You have a serious failure.

fifo->triggerEnable(true);
```

Once you have successfully performed your first acquisition, there is no need to call **prepare()** again.

Note If you are running in auto trigger mode, do not attempt to access the camera with another FIFO. This can cause the auto trigger FIFO to miss triggers without reporting an error.

Semi and Slave Trigger Models

CVL provides two additional built-in trigger models: semi automatic and slave. Semi auto triggering requires a call to **start()**, but the acquisition does not begin until a trigger is received. Slave triggering begins an acquisition when another FIFO, the master FIFO, begins the acquisition. For more information about semi auto triggering and slave triggering see *Starting an Acquisition* on page 82, and see the description of **ccTriggerModel** in the *CVL Class Reference*.

Free Run Trigger Model

The Free Run trigger model requires no software **start()** and no external event to perform image acquisitions.

The acquisition engine starts acquiring images as soon as triggers are enabled. To obtain an acquired image, you only need to call **complete()**.

In many cases, the free run trigger model runs faster than queued manual starts, because the underlying frame grabber can be placed into a continuously acquiring state.

On the MVS-8600, using this trigger model in line scan acquisition is the only way to obtain continuous, gapless acquisitions, one frame to the next. For this combination, other trigger models may place an indeterminate number of dropped lines between frames.

Custom Trigger Models

CVL allows you to create custom trigger models in which:

- Image buffering can be disabled.
- Reporting of **isMissed()** errors can be disabled.
- The requirement to call **start()** for each acquisition can be disabled.
- The trigger source is selectable between *none* (immediate) and *hardware.*
- **ccTriggerModel::bufferHardwareTrigger()** can be enabled.

You create a custom trigger model by copying of one of the built-in trigger models and then customizing the copy using the **copyForCustomization()** function provided in the **ccTriggerModel** class.

Note The **ccTriggerModel** class and related global functions were moved from the *ch_cvl\prop.h* header file to *ch_cvl\trigmodl.h* as of CVL 6.0.1. The former includes the latter, so changes to code created with pre-6.0.1 versions of CVL are not needed.

Create your custom trigger model using the singleton design pattern as shown in the sample code files *%VISION_ROOT%\sample\cvl\acquire\acqutil.*.*

Use **copyForCustomization()** with the pointer handle typedef **ccTriggerModelPtrh**, as described in the CVL Class Reference entry for **ccTriggerModel**.

Changing Properties During Acquisition

In general, you will set your FIFO's properties soon after you create it, and they will remain unchanged for the duration of your acquisition process. In some applications, however, you may want to change property settings while your application runs. This section describes how to do this efficiently.

A property change always takes effect with the next acquisition request. When the request is made depends on the trigger model.

Changing Properties with Manual and Semi Trigger

If your application uses manual or semi triggering, calling **start()** generates an acquisition request. The servicing of the request is separate from the generation of the request. If triggers are disabled, and you call **start()** several times, the acquisition requests are queued until triggers are re-enabled.

The following code illustrates how this works. First, disable triggers. Then, change the exposure property several times, and call **start()** after each change. When triggers are re-enabled, the four acquisition requests are serviced, each with its own exposure time.

```
// disable triggers
fifo->triggerEnable(false);

ccExposureProp* exposureProp = fifo->propertyQuery();

// change the exposure property, then call start()
if (exposureProp)
  exposureProp->exposure(0.001);
fifo->start();

if (exposureProp)
  exposureProp->exposure(0.002);
fifo->start();

if (exposureProp)
  exposureProp->exposure(0.005);
fifo->start();

if (exposureProp)
  exposureProp->exposure(0.009);
fifo->start();

fifo->triggerEnable(true);
// The acquisition system now collects four images
// with exposures of 1, 2, 5, and 9 milliseconds
```

Changing Properties with Auto and Free Run Triggers

If you are using a trigger model that does not support the **start()** function, including auto trigger and free run trigger, an acquisition request is generated immediately after each satisfied request. The first acquisition request is generated when both **triggerEnable()** is true and the trigger model is set to auto trigger. As with manual triggering, the property settings used are those associated with the FIFO at the time that the acquisition request is made.

Thus, if you modify any properties for an auto-triggered FIFO while **triggerEnable()** is true, you cannot predict with which of the subsequent acquisitions the property settings will take effect.

For this reason, the recommended way of changing properties for auto-triggered or free run FIFOs is to set **triggerEnable()** to false before making changes. The following code illustrates how to do this.

```
// The following assertion makes sure that the trigger
// model is auto.
assert(!fifo->triggerModel().canStart());

// Disable triggers and flush any outstanding
// acquisition requests before changing properties.
fifo->triggerEnable(false);
fifo->flush();

ccExposureProp* exposureProp = fifo->propertyQuery();
if (exposureProp)
  exposureProp->exposure(0.001);

// enable triggers
fifo->triggerEnable(true);

// Wait for the acquisition to complete
// but don't wait longer than three seconds.
acqImage = fifo->completeAcq
          (ccAcqFifo::CompleteArgs().maxWait(3.0));

// To change the exposure, disable triggers, then
// set the exposure to five milliseconds. Any
// triggers that might occur are ignored while
// triggerEnable() is false.
fifo->triggerEnable(false)
if (exposureProp)
  exposureProp->exposure(0.005);

// Re-enable triggers to have the exposure take effect
// with the next acquisition request.
fifo->triggerEnable(true);
acqImage = fifo->completeAcq
          (ccAcqFifo::CompleteArgs().maxWait(3.0));
```

Compensating for Settling Time

When you change property settings from one acquisition to the next, CVL may need to impose a settling time delay. Some property changes do not require any settling time delay, while others may require 500 milliseconds or more. You can use the **prepare()** function to ensure that CVL performs all of the operations that require settling time before your acquisitions begin.

The **prepare()** function sets the state of the hardware according to the current property settings. If any of the changes require settling time, **prepare()** waits before it returns. One thing to keep in mind is that **prepare()** has no effect for auto-triggered FIFOs while

triggerEnable() is true, and it has no effect for the other trigger models if an acquisition request is being serviced. To make effective use of **prepare()**, call it when **triggerEnable()** is false.

The following example shows how you can deal with settling time.

```
// Assume a manual triggered fifo that supports exposure.

// Using prepare to avoid overhead is only useful if there's
// something else we could be doing instead. Assume it will
// take a while to move the part into position. First we
// start moving the part.
motionControl.BeginPartMotion();

// Next we set the properties as appropriate and call prepare
// to set up the hardware while the part is in motion.
fifo->properties().exposure(someExposure());
if (!fifo->prepare(0.0))
   throw visionFailure;

// The vision system is ready, now wait for part to get into
// position for inspection
motionControl.WaitForPartInPosition();

// Part is in position, perform the acquisition. This one
// will not have any overhead because of the property change.
// Note the 5 second timeout as a guard against hangs.
fifo->start();
ccAcquireInfo info;
acqImage = fifo->completeAcq
     ((ccAcqFifo::CompleteArgs().acquireInfo(&info)
                                .maxWait(5.0));
if (!acqImage.isBound())
   throw visionFailure;
```

Trigger Delay and Strobe Delay Interaction

The strobe delay (**ccStrobeDelayProp**) and the trigger delay (**ccTriggerFilterProp**) interact such that changing one can affect the other. The following examples describe this interaction.

The timing diagram in Figure 8 illustrates default values for trigger delay and strobe delay.

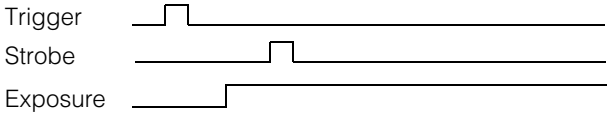


Figure 8. Default values

Increasing the strobe delay has no affect on the trigger, it simply delays when the strobe occurs relative to the exposure. See Figure 9.

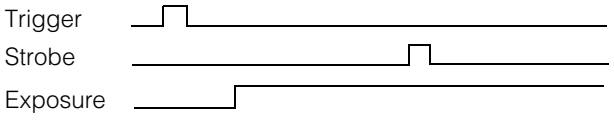


Figure 9. Increasing the strobe delay

When you increase the trigger delay using the default strobe delay, the exposure and strobe are both delayed. See Figure 10.

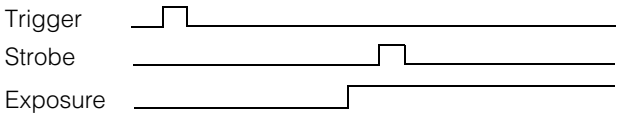


Figure 10. Increased trigger delay

If you keep the default trigger delay but specify a small negative strobe delay the strobe will occur earlier relative to the exposure. The exposure start relative to the trigger is not affected. See Figure 11.

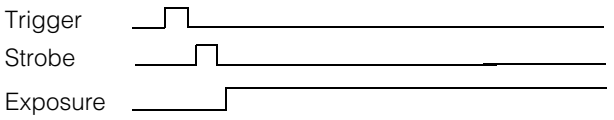


Figure 11. Small negative strobe delay

Since the strobe cannot be moved ahead of the trigger, large negative strobe delays used with the default trigger delay cause the strobe to occur immediately after the trigger, and cause the exposure start to be delayed by the strobe delay time. Effectively, the trigger delay is increased to account for the negative strobe delay. See Figure 12.

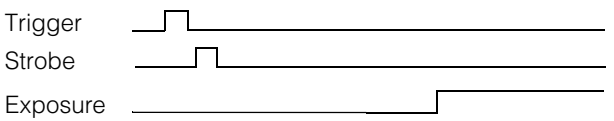


Figure 12. Large negative strobe delay

Simultaneous (Master-Slave) Acquisition

You can acquire images from two or more cameras simultaneously, with one camera designated as master, and with one or more cameras designated as slaves. The slave cameras acquire images at the same time as the master. Synchronous configurations are described in the reference page for **ccTriggerModel::cfSlaveTrigger()** in the *CVL Class Reference*.

When using a master-slave configuration, the slave camera's video timing is locked to the master's video timing. This means you can use a single strobed light source for the master and all slaved cameras.

If you use an external source of synchronization, only the master camera should be connected to the external sync source. All slaved cameras must be set for internal sync source; they will share the master camera's synchronization.

The master camera and all slaved cameras must use identical acquisition FIFO property settings for any properties that affect video timing. This implies that the master camera and all slaved cameras must be same camera model and must use the same CVL video format.

For master-slave configurations, you will see **ccAcqFifo::CompleteArgs.failure(). isMissed()** errors on the master FIFO only.

Simultaneous Acquisition with Analog Cameras

Synchronous image acquisition is supported on the Cognex hardware platforms shown in Table 24. In the Camera Ports column, arrows point to the master camera port.

Hardware Platform	Camera Ports	Notes
MVS-8511, MVS-8600	N/A	Synchronous configurations not supported.
MVS-8514	0 ← 1 0 ← (1, 2) 0 ← (1, 2, 3) 2 ← 3 0 ← 1, 2 ← 3	Supports one or two masters and one to three slaves. Ports 0 and 2 can be masters. Port 0 can have any or all of ports 1, 2, and 3 as slaves. Port 2 can have only port 3 as a slave. Any port not used as master or slave can be used normally.

Table 24. Synchronous configuration support by hardware platform

The synchronous camera port combinations stated in this section are general rules and may be disallowed in software for a particular combination of cameras. Use **ccTriggerProp::couldSlaveTo()** and **ccAcqFifo::isValid()** to determine if a camera port can be slaved to a given master port for your application using your combination of cameras.

Note Changing the trigger model on a master FIFO will flush the master FIFO and all associated slave FIFOs.

Also, deleting a slave FIFO will disable triggers for all the FIFOs in the master-slave group. Cognex recommends deleting all synchronously related FIFOs together at the same time.

Considerations When Using Master-Slave Acquisition

When two or more FIFOs are associated in a synchronous (master-slave) relationship, calling **triggerEnable()** on any related FIFO uniformly enables or disables triggers on all associated FIFOs.

When the certain functions are used in synchronous configurations, observe the following guidelines:

- **ccAcqFifo::flush()**
  Disable triggers on the master FIFO before flushing. See *Flushing FIFOs* on page 117 for more information.
- **ccAcqFifo::triggerModel()**, **ccTriggerProp::triggerMaster()**, and **ccAcqFifo::~ccAcqFifo()**
  FIFOs used in a synchronous configuration should be constructed together, used together, and then destroyed together. If it is necessary to acquire from a single camera of a synchronous group, it is easier to create another FIFO that shares the camera with the master-slave group.

  While the CVL API allows dynamically rearranging synchronous relationships, this is not a common procedure and Cognex recommends against doing so.

  When setting up a synchronous configuration, the following sequence is the most efficient. Other sequences may work, but may result in extra internal function calls that needlessly arm and disarm the video hardware:

  a. Create each FIFO individually, and immediately disable triggers on each one.

  b. On each slave FIFO set the trigger model to **cfSlaveTrigger()** and then assign **triggerMaster()** to the master FIFO.

  c. Enable triggers on the master FIFO.

When destroying FIFOs, the order of destruction is unimportant. However, once one FIFO of a group has been destroyed, the other FIFOs should no longer be used.

Using Callback Functions

The CVL acquisition engine provides a mechanism for you to register callback functions you write. The acquisition engine calls your registered function under one of three conditions:

- When a completed acquisition (or error) is placed in the user queue.
- When the current acquisition reaches the move-part state, which is when the camera's field of view can be changed without affecting the acquired image.
- If a trigger overrun condition occurs.

A CVL acquisition callback is implemented as a callback class you define, which contains your callback function. You must define your callback function as an override of **operator()()** in your callback class.

There are three steps to perform in order to use an acquisition callback:

1. Define a callback class derived from **ccCallback** or **ccCallbackAcqInfo** and override the parentheses operator in that class.
2. Write your callback function as an implementation of **operator()()** in your callback class.
3. Register an instance of your callback class with your acquisition FIFO.

Two Ways to Use Callbacks

You can register your callback class two ways:

- Method 1: by using one of three callback properties of the acquisition FIFO.
- Method 2: by using one of three **ccAcqFifo** callback registration functions. This method allows the acquisition engine to pass a **ccAcquireInfo** object to your callback function, which allows you to use that object's status reporting functions in your callback routine.

The two methods are clarified in Table 25.

Callback Condition	Method	Method of registering example callback class *xyz*	Uses ccAcquireInfo object?
acquisition complete	1	With **completeCallback()** from **ccCompleteCallbackProp** `fifo->properties().completeCallback(xyz);`	No
	2	With **completeInfoCallback()** from **ccAcqFifo** `fifo->completeInfoCallback(xyz);`	Yes
movable state reached	1	With **movePartCallback()** from **ccMovePartCallbackProp** `fifo->properties().movePartCallback(xyz);`	No
	2	With **movePartInfoCallback()** from **ccAcqFifo** `fifo->movePartInfoCallback(xyz);`	Yes
overrun condition	1	With **overunCallback()** from **ccOverrunCallbackProp** `fifo->properties().overrunCallback(xyz);`	No
	2	With **overrunInfoCallback()** from **ccAcqFifo** `fifo->overrunInfoCallback(xyz);`	Yes

Table 25. Methods of registering callback classes

Using Callbacks, Method 1

This section describes how to define a callback function using the one of the callback property classes.

Defining a Callback Class, Method 1

Derive your callback class from **ccCallback** and override the parentheses **operator()()**, as shown in this example:

```
class MyPartMover : public ccCallback
{
  virtual void operator()(); // override
};
```

Defining Your Callback Function, Method 1

Implement **operator()()** in the definition of your callback class to perform the appropriate actions for your application.

Your callback function should set flags or semaphores in your application that allow it to proceed to another state the next time it executes. Your callback function is executed internally by the acquisition engine, which cannot proceed until your callback function returns. Therefore, callback functions should be short and quick, and *must not block*, since blocking would cause the acquisition engine to pause while waiting for the callback function to complete.

The following example implements a callback function that calls a single function defined elsewhere in your application.

```
void MyPartMover::operator()()
{
  myConveyorBeltClass::advanceConveyorBelt();
}
```

Do not perform lengthy operations and do not call CVL functions in your callback function.

Registering Your Callback Class, Method 1

The last step is to create an instance of your callback class and register it with your acquisition FIFO. Registering is accomplished with the active function of one of the FIFO's callback properties, as shown in Table 26.

Callback Property	Function that registers your callback class
ccCompleteCallbackProp	**completeCallback()**
ccMovePartCallbackProp	**movePartCallback()**
ccOverrunCallbackProp	**overrunCallback()**

Table 26. Callback properties and active functions

The following example registers the callback class defined in the examples in this section with the move-part callback property of the current FIFO. Create an instance, not of **ccCallback**, but of the pointer handle typedef version, **ccCallbackPtrh**.

```
ccCallbackPtrh mpmCB = new MyPartMover;
fifo->properties().movePartCallback(mpmCB);
```

Using Callbacks, Method 2

This section describes how to define a callback function using the one of the callback-with-information functions of **ccAcqFifo**.

Defining a Callback Class, Method 2

Derive your callback class from **ccCallbackAcqInfo**. Override the parentheses **operator()()**, specifying a **ccAcquireInfo** object as the single parameter of the **operator()()** function, as shown in this example:

```
class MyTrigCount : public ccCallbackAcqInfo
{
  virtual void operator()(const ccAcquireInfo&); // override
};
```

Defining a Callback Function, Method 2

Implement **operator()()** in the definition of your callback class to perform the appropriate actions for your application.

Your callback function should set flags or semaphores in your application that allow it to proceed to another state the next time it executes. Your callback function is executed internally by the acquisition engine, which cannot proceed until your callback function returns. Therefore, callback functions should be short and quick, and *must not block*, since blocking would cause the acquisition engine to pause while waiting for the callback function to complete.

This example implements a callback function that calls a simple function defined elsewhere in your application.

```
void MyTrigCount::operator()(const ccAcquireInfo&)
{
  myAcqCompleteClass::incrementCounter();
}
```

Do not perform lengthy operations and do not call CVL functions in your callback function.

Using the ccAcquireInfo Object

Callback classes derived from **ccCallbackAcqInfo** pass a **ccAcquireInfo** object to the callback function you define. **ccAcquireInfo** objects store status information about the acquisition, as described in *Using the ccAcquireInfo Object* on page 96.

Your callback function can make use of this status information as shown in the following example:

```
class MyTrigCount : public ccCallbackAcqInfo
{
  public: void operator()(const ccAcquireInfo& info)
  {
    triggerCount_ = info.triggerNum();
    myAcqCompleteClass::incrementCounter();
  }
  private: c_UInt32 triggerCount_;
  public:  c_Uint32 triggerCount() {return triggerCount_;}
};
```

Registering Your Callback Class, Method 2

The last step is to create an instance of your callback class and register it with your acquisition FIFO. Registering is accomplished with one of the callback registration functions derived from **ccAcqFifo**, as shown in Table 27 on page 116.

This example registers the callback class defined in the examples in this section with the acquisition complete callback of the current FIFO. Create an instance, not of **ccCallbackAcqInfo**, but of the pointer handle typedef version, **ccCallbackAcqInfoPtrh**.

```
ccCallbackAcqInfoPtrh mtcCB = new MyTrigCount;
fifo->completeInfoCallback(mtcCB);
```

The following example shows how you might make use of information about the acquisition stored in the callback function's **ccAcquireInfo** object:

```
cogOut << "Number of triggers: " << mtcCB->triggerCount()
       << endl;
```

When Callback Functions Are Called

The three callback types correspond to the three points in the image acquisition cycle that a callback function might be called by the acquisition engine, as shown in Table 27.

Callback Invoked By	When Callback Function Is Called
ccCompleteCallbackProp or **ccAcqFifo:: completeInfoCallback()**	The acquisition-complete callback function is called when an acquisition is placed in the user queue, meaning that the next invocation of **completeAcq()** can retrieve the acquisition (or error) immediately. This callback function is invoked whether or not the acquisition completed successfully. **completeAcq()** may still block until the image transfer across the PCI bus completes, but this delay should be bounded.
ccMovePartCallbackProp or **ccAcqFifo:: movePartInfoCallback()**	The move-part callback is called when the camera has integrated the image and it is safe to move the object from its field of view. The image may not yet have been transferred to video memory. The move-part callback by definition occurs before the acquisition is complete. For this reason, the reason-for-failure information returned by **ccAcquireInfo::failure()** is not accurate or useful for the move-part callback.
ccOverrunCallbackProp or **ccAcqFifo:: overrunInfoCallback()**	Allows you to provide a function that will be called if the acquisition failed because the acquisition was overrun. See *Determining Why an Acquisition Failed* on page 98.

Table 27. Callback types and when each is called

See Figure 7 on page 92 for an illustration of when the move-part and acquisition complete callback functions are called.

General Recommendations

This section describes some techniques that can improve the reliability of your vision application, especially when it is running in a multi-threaded environment or when it uses synchronous configurations.

Unsupported Global Video Format Functions

Certain global video format functions (for example, **cfXcSt50_640x480()**, which supports the Sony XC-ST50 camera on the MVS-8110) are located in *ch_cog\vidfmt.h*. The *ch_cog* directory contains interfaces that are generally unsupported and subject to change. If you are using the unsupported global video format functions, you must include this header file instead of (or along with) *ch_cvl\vidfmt.h*. Cognex recommends, however, that you change your code to use the preferred **ccStdVideoFormat::getFormat()** interface together with the proper video format string. In the latter case, you would only need to include *ch_cvl\vidfmt.h* and not *ch_cog\vidfmt.h*. For the XC-ST50 camera, the strings used to support various video formats are shown in the supported camera table included with the CVL documentation.

Avoid Frequent FIFO Allocation and Deallocation

The examples in this chapter are intended to illustrate several techniques in a small amount of space. Although the example program at the beginning of this chapter may appear to imply that you should allocate an acquisition FIFO each time before acquiring an image and dispose of it immediately after acquisition, this practice is not recommended.

Allocate a FIFO once and use it repeatedly to acquire images. Dispose of the FIFO only after you have finished acquiring images. Remember that if you use pointer handles, the FIFO is deallocated automatically when all referencing handles go out of scope.

Flushing FIFOs

This section describes how to flush the acquisition FIFO using different frame grabbers.

You call **ccAcqFifo::flush()** to discard all outstanding acquisitions, leaving the FIFO in the idle state. If an acquisition is pending or is in progress, it is cancelled and discarded. Completed acquisitions in the **completeAcq()** queue are also discarded.

If **ccAcqFifo::triggerEnable()** is true, **flush()** disables triggers before discarding all acquisitions, then re-enables triggers.

Because enabling/disabling triggers affects all related masters and slaves, this function can have unpredictable results in a synchronous configuration when triggers are enabled. To avoid hangs and unexpected behavior, follow the steps below when flushing master-slave FIFOs:

1. Call **triggerEnable(false)** on the master FIFO. CVL will forward the call to all slaves.
2. Call **flush()** on all synchronously related FIFOs. The calls to flush can happen simultaneously in separate threads if desired.
3. Call **triggerEnable(true)** on the master FIFO, if desired.

Note that each of the above steps must be completed before proceeding to the next step.

Flushing a FIFO can take anywhere from 0.3 ms to 32 ms to complete. Because a flush halts any acquisitions in progress, CVL waits for the camera to finish reading out the halted image before flushing the FIFO. This ensures that the camera is fully flushed and ready to start a new acquisition. Disabling triggers will take a finite amount of time, as determined by the camera frame time. This affects all functions that call the function to disable triggers.

If you simply want to get rid of stale images, use the code shown below. This code will flush the completed acquisitions (but not pending acquisitions) for a FIFO in less than 100 microseconds per loop without affecting any acquisitions that may be in progress:

```
while(fifo->isComplete())
  fifo->completeAcq(ccAcqFifo::CompleteArgs()
                                                .makeLocal(false));
```

Useful Techniques

This section shows you how to determine whether a frame grabber is present, how to determine whether a video format is supported, how to determine which properties a FIFO supports, and how to create CVL pel buffers from non-CVL image data.

Note Many of these techniques use C++ exception handling and the **dynamic_cast** operator. If you are not familiar with these language features, learn more about them in *The C++ Programming Language, Third Edition* by Bjarne Stroustrup.

Testing for a Frame Grabber

If you are certain that your application will always be used with a particular frame grabber, you can use the specific class that describes it, as shown in *Getting a Frame Grabber Reference* on page 72. In some cases, however, you may not know exactly what hardware is available to your application. To test whether a particular board is a frame grabber, use the following:

```
ccBoard& board = ccBoard::get(i);

ccFrameGrabber* pfg = dynamic_cast<ccFrameGrabber*>(&board):
if (!pfg)
{
  // board is not a frame grabber
}
```

You can use the same technique to determine the particular type of frame grabber available to your program. For example, the following code checks for an MVS-8100M frame grabber.

```
ccBoard& board = ccBoard::get(i);

cc8100m* pfg = dynamic_cast<cc8100m*>(&board):
if (!pfg)
{
  // board is not an MVS-8100M
}
```

Note that when you use the code above to find a frame grabber board, the index (*i*) refers to a specific board. When you have more than one frame grabber in your system each board is assigned an index starting with 0. Generally frame grabber boards are assigned index numbers according to the motherboard slot into which they are installed. Index 0 is assigned the lowest numbered slot, index 1 to the next higher numbered slot, and so on. However, this can vary with motherboard manufacturers and can be BIOS

dependent. You should consult your system hardware documentation to verify the correct frame grabber index numbers to use when more than one frame grabber is installed in your system. If you have only one frame grabber, it will always be index 0.

Note also that **ccBoard** covers frame grabbers of all types as do its associated index numbers. If you specify a particular type frame grabber, each type uses its own index numbers starting with 0. For example,

```
cc8100m& fg = cc8100m::get(i);
```

The index for a **cc8100m** frame grabber board may not be the same index you would use if you acquire a **ccBoard** object for the same board. Figure 13 shows an example system configuration to illustrate this point.

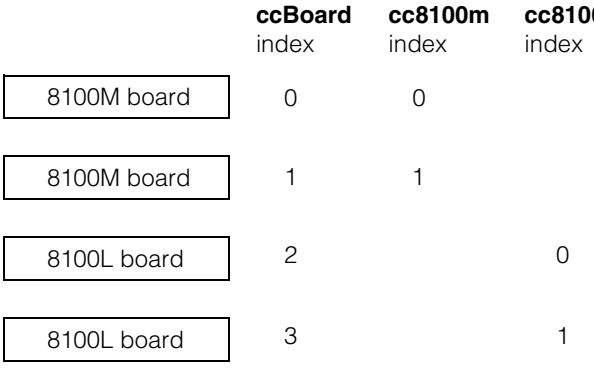


Figure 13. Frame grabber index example

Testing for Video Formats

To print out a list of the names of all video formats you can use the following code:

Note Starting with CVL 6.0 you must #include *acq.h*, *prop.h*, and *vidfmt.h* explicitly.

```
#include <ch_cvl/acq.h>
#include <ch_cvl/prop.h>
#include <ch_cvl/vidfmt.h>
#include <ch_cvl/constrea.h>

typedef cmStd vector<const ccVideoFormat*> Vfv;
```

```
void printVideoFormats()
{
  Vfv videoFormats;
  videoFormats = ccVideoFormat::fullList();

  cogOut << cmT("The full list contains ") <<
    videoFormats.size() << cmT(" elements.") << cmStd endl;

  for (Vfv::const_iterator i = videoFormats.begin();
       i != videoFormats.end(); i++)
    cogOut << (*i)->name() << cmStd endl;
}
```

To return a list of all the video formats that support the frame grabber *fg*, use the following code:

```
cmStd vector<const ccVideoFormat*>myVidFmts =
  ccVideoFormat::filterList(ccVideoFormat::fullList(), fg);
```

To test whether a frame grabber supports a particular video format, you can use something like the following:

```
ccFrameGrabber& fg = cc8100m::get(0);

const ccStdVideoFormat& fmt =
     ccStdVideoFormat::getFormat(cmT("Sony XC-ST50 640x480"))
bool fmtIsSupported = fg.isSupportedEx(fmt);
```

Testing for Properties

While some properties apply to all acquisition FIFOs, some apply only to specific kinds of FIFOs. If you are not sure that an acquisition FIFO supports a particular property, you can use the **propertyQuery()** function to test for it. For example, this is how to test to see if a FIFO supports the contrast and brightness property:

```
ccContrastBrightnessProp* cbProp = fifo->propertyQuery();
if (cbProp)
  cbProp->contrastBrightness(0.4, 0.6);
```

Or, using reference semantics, write:

```
try
{
  ccContrastBrightnessProp& cbProp = fifo->propertyQuery();
  cbProp.contrastBrightness(0.4, 0.6);
}
catch (cmStd bad_cast&)
{
  // The brightness/contrast property is not supported.
}
```

Getting the Name of a Frame Grabber

To get the name of a frame grabber associated with a particular acquisition FIFO, use the following code:

```
dynamic_cast<ccBoard&>(fifo->frameGrabber()).name();
```

Querying Video Format Strings

Several string query methods in **ccVideoFormat** allow you to query specific information contained in video format strings. Types of information you can obtain include the camera manufacturer and model, and the video format resolution, drive type, and options.

Note The format strings returned by **ccVideoFormat::name()** do not change from release to release.

Table 28 shows examples of the types of information you can obtain from video format strings using the string query methods. This information is not guaranteed to be the same from release to release.

ccVideoFormat Method	Example Return
cameraManufacturer()	“Sony”
cameraModel()	“XC-55”
videoFormatResolution()	“640x480”
videoFormatDriveType()	“IntDrv”
videoFormatOptions()	"rapid-reset, shutter-sw-EDONPISHAII"

Table 28. Video format string methods

The two methods in Table 29 return boolean values that indicate whether a video format is CCF-based or static, and whether it is provided as a static format for backwards compatibility only but internally implemented with a CCF. For example, the **cfxc55_640x480()** static format is used on the MVS-8120 with CVM1, but is internally implemented with a CCF.

ccVideoFormat Method	Return Value
formatFromCCF()	• *True* if format is created from a CCF • *False* if format is built into CVL (static)
isSupportedForLegacy()	• *True* if provided as static format for backwards compatibility only, but internally implemented with a CCF • *False* if implemented purely as a static function

Table 29. Video format information query methods

Creating Pel Buffers from Arbitrary Pixel Data

In some cases, image data may not come from a camera connected to a Cognex frame grabber. For example, you may have images stored in a database in a proprietary format, or the image may come from another device connected to your host computer. In these cases, you need to store the image data in a CVL pel buffer before you can use any of the Cognex vision tools.

To store your image data in a CVL pel buffer (**ccPelBuffer**), you must first create a root image, or pel root, (**ccPelRoot**), which will contain the pixel data of your image. Then you associate the pel root with a pel buffer, which will impose a coordinate system on the root image and allow you to use it with the vision tools.

The instructions that follow, adapted from the example program *userbuf.cpp*, show you how to do this step by step. After you read these instructions, you will find it useful to also read *Root Images* on page 228 and *Windows (Pel Buffers)* on page 229 to learn the details of root images and pel buffers.

The example program uses an 8-bit, 640x480, synthetic image to represent the image data that you might supply. In your application, of course, the image may be a different bit-depth or size.

```
const int width = 640;
const int height = 480;
unsigned char *rawImage = new unsigned char[width * height];

  // make the entire image black (0 pixel value)...
memset(rawImage, 0, width * height);
```

```
  // ...except for a small white (255 pixel value) square.
for (int y=300; y < 340; y++) {
  unsigned char *rowPtr = rawImage + y * width;
  unsigned char *pelPtr = rowPtr + 200;
  memset(pelPtr, 255, 40);
}
```

rawImage now points to your image data.

The next step is to create a root image to manage the pixel data. The parameterized class **ccPelRoot** lets you create root images from raw pixel data in several bit depths. In this example, each pixel is 8 unsigned bits, which corresponds to the Cognex type **c_UInt8**. The following code creates the root image.

```
int pitch = width;
ccPelRoot<c_UInt8>* root =
  new ccPelRoot<c_UInt8>(width, height, pitch, rawImage);
```

Note that you use **new** to allocate the root image. You cannot create a root image on the stack. The root image you create does not copy your pixel data, it points to it.

The *pitch* parameter is the difference between the first pixel of adjacent rows. In some cases, each row may have additional padding pixels beyond the width.

After creating the root image, bind it to a pel buffer. Like **ccPelRoot**, **ccPelBuffer** is a parameterized class. Your pel buffer must be of the same pixel type as your root image. The following code binds a root image to a pel buffer.

```
ccPelBuffer<c_UInt8> buf(root);
```

Once you bind the root image to a pel buffer, the pel buffer owns the root image. You must not destroy the root image yourself. It will be deleted automatically when the last pel buffer that uses it is destroyed.

At this point, you can use the pel buffer *buf* as you would any other pel buffer that contained an acquired image. For example, the sample program *userbuf.cpp* displays the contents of *buf* in a console window.

If you want to reclaim the memory used by *rawImage*, first make sure that no other pel buffer references your root image. The member function **refc()** returns the number of pel buffers that share a pel buffer's root image. When **refc()** returns 1, the root image is used by only one pel buffer. At that point, you can unbind the root image. Unbinding releases the memory occupied by the **ccPelRoot** image, but not the underlying pixel data. If you allocated it, you need to delete it yourself. The following code shows how to do this.

```
if (buf.refc() == 1) {
  buf.setUnbound();
  delete[] rawImage;
}
```

Acquiring Images: Application Notes

3

This chapter builds on the previous chapter, and describes how to use the Cognex Vision Library to acquire images for particular applications.

This chapter has the following major sections:

Acquiring with Color Cameras describes how to acquire color images.

Acquiring with Line Scan Cameras describes how to set up the acquisition software when using a line scan camera.

Acquiring with Camera Link Cameras describes how to set up the acquisition software when using a Camera Link camera on an MVS-8600 or MVS-8600e frame grabber.

Acquiring with GigE Vision Cameras describes how to set up the acquisition software when using a GigE Vision camera.

Acquiring from an Imaging Device describes how to set up the acquisition software when using third-party cameras or frame grabbers.

Persistent Camera Enumeration describes how to use Persistent Camera Enumeration, which prevents unexpected changes in camera enumeration. For example, if your application uses **ccGigEVisionCamera::get(0)** to get a reference to a particular camera, it likely expects to always get a reference to the same camera whenever the application is run. Without PCE in a multi-camera system, if the power supply for that camera fails, then **ccGigEVisionCamera::get(0)** will return a reference to a different camera. With PCE, cameras will always be enumerated at the same index regardless of the state of other cameras in the system.

Camera-Specific Usage Notes provides information on using certain cameras with CVL.

Frame Grabber Acquisition Usage Notes describes techniques and work arounds specific to a frame grabber or frame grabber family.

Acquiring with Color Cameras

You can acquire color images using a color camera and a Cognex MVS-8514 frame grabber. Note that color images can be used only for display. None of the CVL image processing or vision tools operate on color images. However, you can acquire a color image and create a grey scale pel buffer from it that you can then use with vision tools. The following examples show how you acquire color images.

Acquiring with Color Cameras on an MVS-8514

The MVS-8514 frame grabber can be used with a color camera to acquire 3-plane color images which can then be converted into packed RGB images. Use a 3-plane color format to construct a single FIFO as shown below.

```
const ccStdVideoFormat& format = ccStdVideoFormat::getFormat
      (cmT("Sony DXC-390 640x480 IntDrv (3 Plane Color) CCF"));

ccAcqFifoPtrh fifo = format.newAcqFifoEx(fg);
```

Use this FIFO to acquire an **ccAcqImage** as shown in the previous section.

This differs from an older method where a single plane format was used to construct 3 FIFOs, one for each plane, and then master/slave acquisition was used to acquire the image. The old method is still supported, but the method above is recommended for new applications.

Acquiring Packed RGB Images from Monochrome Cameras

Many Cognex frame grabbers can reformat image data into packed RGB formats and DMA them directly to the video display. This is often done to reduce CPU load.

Use code like the following when constructing **ccAcqFifo**s that will be used exclusively with **startLiveDisplay()**.

```
// Determine the acquisition image format that matches the current
// display settings.
```

```
ceImageFormat ideal =
  cfConvertDisplayFormat2ImageFormat(myDisplay.displayFormat());

// Create a fifo to acquire in the ideal format, if supported
ccAcqFifoPtrh fifo;
if (fg.isSupportedEx(vidfmt, ideal))
  fifo = vidfmt->newAcqFifoEx(fg, ideal);

// Otherwise, acquire in the format specified in the video
// format
else
  fifo = vidfmt->newAcqFifoEx(fg);

// Display a live image using the newly created fifo.
myDisplay.startLiveDisplay(fifo);
```

Color Camera Usage Notes

This section discusses frame versus field integration on the color cameras supported by CVL, and provides usage notes.

Frame Versus Field Integration

Interlaced video cameras generally operate in either frame integration or field integration mode, also known as frame or field accumulation.

Note CVL does not support strobed image acquisition on cameras operating in field integration mode.

The Sony XC-003 and DXC-390 cameras ship from the factory configured for frame integration. They can be set to use field integration with a menu setting on their onboard menu system.

Color Image Quality

Be sure to experiment with your color camera's white balance and color temperature settings to obtain the best possible images. These features are adjustable by means of switches on the back of each camera or by means of menu settings. Consult the camera's documentation for instructions.

Acquiring with Line Scan Cameras

Some Cognex vision boards and video modules, such as the MVS-8120/CVM11 (which is not supported any longer), MVS-8600, and MVS-8600e support line scan cameras. These cameras acquire a single row of pixels at a time, building a complete image as the subject passes in front of the camera on a stage or conveyor. An encoder keeps track of the position of the subject as it travels in front of the camera, and communicates this information to the video module to let it know when to start and stop acquisitions. (Detailed information on the MVS-8600 and MVS-8600e frame grabbers can be found in *Acquiring with Camera Link Cameras* on page 141).

The major difference between line scan acquisition and area scan (normal) acquisition in CVL is that you need to specify line scan-specific properties, such as the number of encoder steps that correspond to a single image line.

There are significant differences between the use of line scan cameras with MVS-8120/CVM11, and with the MVS-8600 and MVS-8600e frame grabbers, as outlined in Table 30 below.

	MVS-8120/CVM11	**MVS-8600/MVS-8600e**
Maximum image size	32 megapixels	Limited by available memory.
Maximum lines per image	8192	64000
Supports continuous line scan	No	Yes (FreeRun trigger model only)
Supports strobed acquisition	Yes (not tested)	No
Supports encoder-triggered acquisition	Yes	MVS-8601/MVS-8602: No MVS-8602e: Yes
Steps-per-line setting for test encoder	Settable	Ignored. Camera free runs at maximum rate allowed by exposure setting.
Directional test encoder	Yes	No
Supports fractional steps-per-line	Yes	No
Encoder types supported	RS-422	RS-422, LVDS or TTL (open collector)
Uses **positiveAcquireDirection()** setting	At start of acquisition.	For every line

Table 30. Line scan differences between MVS-8120/CVM11 and MVS-8600/MVS-8600e

	MVS-8120/CVM11	MVS-8600/MVS-8600e
Supports **useSingleChannel()**	No	Yes (useful when direction of motion is irrelevant)
Encoder count behavior	Continuous	Available only when acquisition is in progress
Supports **zeroCounter()**	Yes	Yes (only to reset backwards count)
Supports **zeroCounterWhenTriggered()**	Yes	No
Supports **currentEncoderCount()**	Returns instantaneous encoder counter value.	Only valid during acquisition; returns estimated value relative to start of acquire, based on currently acquiring line number.
Supports **encoderPort()**	Yes	Yes
How to offset first acquired line from trigger	**zeroCounterWhenTriggered()** **encoderOffset()**	Not available
Selectable encoder resolution	No. Fixed **encoderResolution** of 4x	Fixed **encoderResolution** of 1x for MVS-8601/MVS-8602. Selectable **encoderResolution** of 1x, 2x, or 4x for MVS-8602e
Encoder overruns	Always reported	Never reported
Ignoring backward motion between acquires	Use **zeroCounterWhenTriggered()**	Use **zeroCounter()**

Table 30. Line scan differences between MVS-8120/CVM11 and MVS-8600/MVS-8600e

- To understand how the two platforms use the **positiveAcquireDirection()** setting, see *Detecting Encoder Direction* on page 131.
- To understand line scan acquisition with MVS-8120/CVM11, see *To always ignore backward encoder motion between acquisitions, set ignoreBackwardEncoderCountsBetweenAcquires() to true (MVS-8602e only).* on page 132.
- To understand line scan acquisition with MVS-8600 and MVS-8600e, see *Line Scan Acquisition with MVS-8600 and MVS-8600e* on page 132.

Detecting Encoder Direction

This section describes the **positiveAcquireDirection()** setting and how it is used on the MVS-8600 and MVS-8600e.

Using positiveAcquireDirection()

Rotary shaft encoders always produce normal-direction signals when the encoder's shaft is spinning in a particular direction, and reverse-direction signals when the encoder's shaft is spinning in the opposite direction. CVL does not alter or control this fundamental encoder behavior.

A setting of **positiveAcquireDirection()** = true is the default; it defines the state where your encoder is working as expected, and as described in the preceding paragraph.

Your encoder may produce opposite results because of the way it is wired, or because a number of rotation wheels are spun by the encoder's shaft. Set **positiveAcquireDirection()** to false in cases where you empirically determine that your encoder rotation direction is reversed.

positiveAcquireDirection() with the MVS-8600 and MVS-8600e

When using the MVS-8600 or MVS-8600e with a line scan camera, encoder triggering is not supported. After image acquisition begins, it continues for the specified number of lines in the positive direction, as defined by **positiveAcquireDirection()**.

The MVS-8600 and MVS-8600e have hardware support for detecting and reacting to direction changes reported by the encoder. The hardware keeps track of direction stops and reversals, and only allows lines scanned in the positive direction to form the image, where “positive direction” is defined by the setting of **positiveAcquireDirection()**. Thus, the MVS-8600 and MVS-8600e effectively check the register set by **positiveAcquireDirection()** once per line scanned.

If the MVS-8600 or MVS-8600e’s encoder reports that motion direction has reversed, the hardware counts the number of lines in the reverse direction, and does not continue the current acquisition (or start the next acquisition) until the encoder reports that it has changed direction again and has moved the same number of lines in the positive direction. If you wish to override this behavior you can call **zeroCounter()** to clear the backward count and cause acquisition to occur without this delay. There are several methods that you can use to override or manage this behavior:

- To ignore backward encoder motion altogether, use the **useSingleChannel()** flag.
- To discard backward encoder motion that has happened since the last acquisition completed, call **zeroCounter().**

- To always ignore backward encoder motion between acquisitions, set **ignoreBackwardEncoderCountsBetweenAcquires()** to true (MVS-8602e only).

Line Scan Acquisition with MVS-8600 and MVS-8600e

The discussion of acquisition from line scan cameras in this section applies only when using MVS-8600 or MVS-8600e and a supported Camera Link line scan camera. For more information on the MVS-8600 and MVS-8600e frame grabbers see *Acquiring with Camera Link Cameras* on page 141.

Overview of MVS-8600 and MVS-8600e Line Scan Acquisition

The MVS-8600 and MVS-8600e do not support encoder-triggered acquisition, but do support continuous line scan and large image sizes.

See Table 30 on page 129 for a summary of the differences between MVS-8120/CVM11 line scan, and the MVS-8600 and MVS-8600e line scan.

MVS-8600 Line Scan Code Example

The following example is supplied as the single file CVL sample *acq8600linescan.cpp* with versions of CVL that support the MVS-8600. It demonstrates the free run trigger model and continuous line scan acquisition, in which images are acquired without dropping any lines between images.

The example first defines a utility function **cfStitchPelBuffer()**, then proceeds into the **cfSampleMain()** section.

```
#include <ch_cvl/windisp.h>
#include <ch_cvl/vp8600.h>
#include <ch_cvl/acq.h>
#include <ch_cvl/prop.h>
#include <ch_cvl/vidfmt.h>
#include <ch_cvl/constrea.h>
#include <ch_cvl/timer.h>

// Modify ckNumImages to acquire a larger number of
// continuous images
#define ckNumImages 5

// NOTE : This sample code is specific to the Basler L402k camera
// Change the camera type to the camera that you actually use.
```

```
using namespace std;

ccPelBuffer<c_UInt8> cfStichPelBuffer(ccPelBuffer<c_UInt8> pb[],
                                      c_Int32 size,
                                      ccPelBuffer<c_UInt8>& stitchedPb)
{
  // This function only stitches images using the
  // width and height of the provided pelbuffer

  c_Int32 totalHeight = 0;

  for (int idx = 0; idx < size; idx++)
    totalHeight += pb[idx].height();

  ccPelRoot<c_UInt8>* img = new
        ccPelRoot<c_UInt8>(pb[0].rowUpdate(),
        totalHeight, 1);
  stitchedPb = ccPelBuffer<c_UInt8>(img);

  c_Int32 currentLoc = 0;
  for (int idx = 0; idx < size; idx++)
  {
    c_UInt8* pbPtr8 = stitchedPb.pointToRow(currentLoc);
    c_UInt8* pbOrig8 = pb[idx].pointToPel(0,0);
    c_Int32 rowUpdate = pb[idx].rowUpdate();
    c_Int32 height = pb[idx].height();
    currentLoc += height;
    memcpy(pbPtr8, pbOrig8, rowUpdate * height);
  }

  // We stitched using the pelbuffer's rowupdate.
  // Change it to the width of the original pelbuffer.
  stitchedPb.window(0,0,pb[0].width(),totalHeight);
  return stitchedPb;
}

int cfSampleMain(int, TCHAR** const)
{
  // Quit if there is no 8600 board installed
  if (!cc8600::count())
  {
    cogOut << cmT("No 8600 boards installed in this system.")
           << endl;
    return 1;
  }
```

```
  // Get a frame grabber reference
  cc8600& fg = cc8600::get(0);
  cogOut << cmT("This 8600 board has ") << fg.numChannels()
         << cmT(" channels.") << endl;

  // Get a reference to the video format. Use the video format
  // name correct for your line scan camera, as shown in the
  // Supported Cameras for your release of CVL.
  const ccStdVideoFormat* format;
  try
  {
     format = &ccStdVideoFormat::getFormat(cmT
       ("Basler L402k 4080x512 CameraLink (8-bit, 2-tap,
level-controlled) CCF"));

  }
  catch (ccVideoFormat::NotFound&)
  {
    cogOut << cmT("Video format not found.") << endl;
    return 0;
  }

  // Check to make sure the format and frame grabber are
  // compatible.  If you catch exceptions when creating the
  // FIFO (shown below), this step isn't necessary.
  if (!fg.isSupported(*format))
  {
    cogOut << cmT("Frame grabber does not support video format.")
           << endl;
    return 0;
  }

  // Create the FIFO.
  // This step may generate an exception because:
  //   The frame grabber could not be initialized
  //   The video format is not supported
  //   Required CVL files are missing
  ccGreyAcqFifoPtrh fifo;
  try
  {
    fifo = format->newAcqFifo(fg);
  }
  catch (ccException& exc)
  {
    cogOut << cmT("Fifo creation failed due to exception: ")
```

```
            << exc.message() << endl;
      return 0;
    }

    // Set up the FIFO properties
    ccAcqProps& props = fifo->properties();

    // Disable triggers while setting properties
    props.triggerEnable(false);

    props.exposure(20e-6);
    props.useTestEncoder(true);

  // useTestEncoder(true) causes the MVS-8600 to free run
  // The steps-per-line setting has no effect when using
    // the test Encoder with an MVS-8600.
    // If you are using a real encoder, set useTestEncoder(false)
    // and then supply a valid steps-per-line value corresponding
    // your encoder's input.
    props.stepsPerLine(10);

    // Set the trigger model to free run to support continuous
    // line scan acquisition.
    props.triggerModel(cfFreeRunTrigger());

    if (!fifo->prepare(0.0))
    {
      cogOut << cmT("FIFO prepare failed, check camera connection.")
            << endl;
      return 0;
    }
    ccAcquireInfo info;
    ccPelBuffer<c_UInt8> pb[ckNumImages];
    ccPelBuffer<c_UInt8> pbDummy;

    // Create the display console
    ccDisplayConsole* disp = new ccDisplayConsole(ccIPair(
          800, 600), format->name());

    // Enable Triggers
    props.triggerEnable(true);

    // Acquire some images with no gaps between them.
    for (int i = 0; i < ckNumImages; ++i)
    {
      // In free run trigger model, there is no need to explicitly
```

```
 // call fifo->start()
  ccTimer timer(true);
  // Get a pel buffer with the image.
  pb[i] = fifo->complete(
          ccAcqFifo::CompleteArgs().acquireInfo(&info));
  timer.stop();
  if (info.failure())
  {
    cogOut << cmT("Acquisition failed. Reason: ")
           << (info.failure().isMissed() ?
               cmT("missed trigger") :
               info.failure().isAbnormal() ?
               cmT("abnormal failure") :
               info.failure().isTooFastEncoder() ?
               cmT("too fast") :
               info.failure().isInvalidRoi() ?
               cmT("bad ROI") :
               cmT("unknown failure."))
               << std::endl;
    continue;
  }

  // Process this image.
  cogOut  << cmT("Acquired image size: ")
          << pb[i].width() << cmT("x") << pb[i].height()
          << cmT(" in ") << timer.msec() << cmT(" ms.")
          << std::endl;
}

// Stop acquiring
props.triggerEnable(false);

// Display the individual images first
for (int idx = 0; idx < ckNumImages; idx++)
{
  disp->image(pb[idx]);
  disp->fit();
  bool ok = cfWaitForContinue();
}

// Stitch the five images together into a single pelbuffer
ccPelBuffer<c_UInt8> stitchedBuf;
ccTimer timer(true);
cfStichPelBuffer(pb, ckNumImages, stitchedBuf);
timer.stop();
```

```
  // Display the stitched image
  if (stitchedBuf.isBound())
  {
    disp->image(stitchedBuf);
    disp->fit();
    cogOut  << cmT("Stitched image : ")
            << stitchedBuf.width() << cmT("x")
            << stitchedBuf.height()
            << cmT(" in ") << timer.msec() << cmT(" ms.")
            << std::endl;
  }

  bool ok = cfWaitForContinue();
  delete disp;

  return 0;
}
```

Getting a Frame Grabber

Use the techniques discussed in *Getting a Frame Grabber Reference* on page 72 to get a reference to a frame grabber.

Selecting a Video Format

The next step is to select the *video format*, which describes the camera connected to your frame grabber and the size of the image to acquire. Unlike area scan cameras, the line scan cameras describe the default size of the image you can acquire. You can make the actual size of the image larger or smaller with the ROI property, as described later.

The line scan example for the MVS-8600 specifies a Basler line scan camera:

```
     format = &ccStdVideoFormat::getFormat(cmT
       ("Basler L402k 4080x512 CameraLink (8-bit, 2-tap,
level-controlled) CCF"));
```

This camera has a sensor that is 4080 pixels wide. The default image size for this camera is 4080 pixels x 512 pixels.

The *Getting Started* manual for your version of CVL lists the video formats supported by various Cognex boards.

Creating an Acquisition FIFO

Line scan cameras are grey-scale cameras. Use the same techniques discussed on page 73 to create an acquisition FIFO for a line scan camera.

Setting FIFO Properties

There are several properties that apply only to line scan cameras. These are properties that let you refine the image size and specify the relationship between the encoder and the acquisition software.

Setting the Region of Interest

If you do not want to use the video format's default image size, use the ROI property to set the actual size of the image you will acquire. For example, to set the image size to 1536 pixels wide by 512 pixels deep instead of the default 4080 by 512 pixels:

```
props.roi(ccPelRect(0, 0, 1536, 512));
```

Note

You can use the ROI property to set an image size larger than the video format's default setting, as long as your image size does not exceed the maximum image size for your platform. The platform maximum for the MVS-8600 is tested up to 8K by 16K.

For example, to specify a 4K by 4K image size, use this example:

```
props.roi(ccPelRect(0, 0, 4080, 4096));
```

Setting Encoder Properties

The next few properties specify the relationship between a position encoder and the acquisition software:

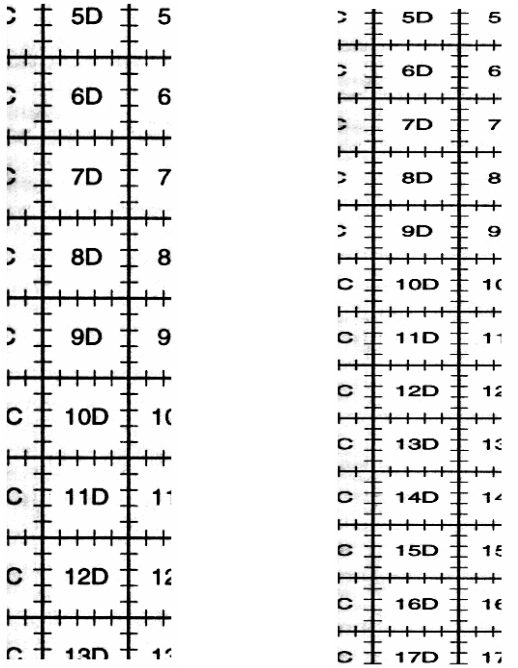
```
props.stepsPerLine(10);
props.positiveAcquireDirection(true);
```

The **stepsPerLine()** property specifies how many encoder counts are needed to make up one row of pixels. The *CVL Class Reference* entry for **ccEncoderProp** contains more information about setting the number of steps per line. You can use the following formula to determine the minimum steps per line, but you may need to refine this value for your application.

$$minStepsPerLine = (encoderCountsPerSecond \times effectiveSensorScanTime) + 1$$

For line scan cameras that support shuttering (where you set the exposure property, **ccExposureProp::exposure()**), *effectiveSensorScanTime* is the greater of the exposure time or *sensorScanTime*. The *sensorScanTime* is the time required to readout the entire line scan sensor.

Figure 14 shows the effect of different values of steps per line. Depending on your application, the vertical compression of the image may or may not be a desirable feature.



stepsPerLine(8) stepsPerLine(12)

Figure 14. Different stepsPerLine() settings.

The **positiveAcquireDirection()** setting is discussed in *Using positiveAcquireDirection()* on page 131.

Using a Test Encoder

CVL provides a test encoder that you can use to test your application without a physical encoder. To use the test encoder, set **useTestEncoder()** to true.

When using the test encoder with the MVS-8600, the hardware free runs and acquires lines as fast as possible without using an encoder input. The **stepsPerLine()** setting is ignored.

See The *CVL Class Reference* entry for **ccEncoderProp** for more information about using the test encoder.

Enabling Triggers

The example program sets up the free run trigger model and then enables triggers to allow acquisition to begin.

```
props.triggerModel(cfFreeRunTrigger());
. . .
props.triggerEnable(true);
```

See *Free Run Trigger Model* on page 102 for more information about free run triggering.

Starting and Completing the Acquisition

Once you have set up the special properties for line scan acquisition, you acquire images the same way as when you use an area scan camera.

The code example on page 132 finishes by demonstrating the stitching of several acquired images into a single large image.

Acquiring Continuous Images

To acquire continuous line scan image acquisitions, with no gaps between frames, use the free run trigger model, as described in *Free Run Trigger Model* on page 102. When using other trigger models, you may get dropped lines between acquisition frames.

For best results with the free run trigger model, limit the height of acquired images.

Acquiring with Camera Link Cameras

CVL supports the Cognex MVS-8600 and MVS-8600e series frame grabbers that support the industry-standard Camera Link protocol. The series includes the following models:

PCI bus models:

- MVS-8601, with support for one base configuration Camera Link camera
- MVS-8602, with support for one or two base configuration Camera Link cameras

Note: The term MVS-8600 refers to both models.

PCI Express bus model:

- MVS-8602e, with support for one or two base configuration Camera Link cameras, or one medium configuration camera

Note: The term MVS-8600e refers to both models.

CVL Support for MVS-8600 and MVS-8600e Frame Grabbers

CVL provides the following support for MVS-8600 and MVS-8600e frame grabbers.

Class Support for the MVS-8600 and MVS-8600e

You can adapt CVL sample code and your own existing vision processing application to use the MVS-8600 and MVS-8600e by getting a reference to a **cc8600** object.

```
cc8600& fg = cc8600::get(0);
```

Documentation for the **cc8600** class can be found in the *CVL Class Reference* which is installed as part of your CVL release.

Using I/O Devices with the MVS-8600 and MVS-8600e

The MVS-8600 and MVS-8600e support a full complement of I/O connections per camera, including support for triggers, strobes, encoders, and other parallel I/O devices. Connect wires from trigger, strobe, and encoder devices to the 8600's I/O connection module, Cognex part number 800-5885-1.

The MVS-8600 and MVS-8600e support loadable software I/O configurations that determine the number and type of I/O connections available. The cable you use to connect the frame grabber board to the I/O connection module corresponds to the software I/O configuration you plan to use. Table 31 summarizes the I/O configuration and cable options. The connection of I/O devices to the MVS-8600 and MVS-8600e is discussed in Chapter 1 of the *MVS-8600 and MVS-8600e Hardware Manual*.

Software I/O option	Cable to I/O connection module	Camera configuration	Encoder type	MVS-8602, MVS-8602e only
1 (default)	300-0539	One area scan	n/a	
		Two area scan	n/a	yes
		One line scan	LVDS	
		One line scan, one area scan	LVDS	yes
		Two line scan (shared encoder)	LVDS	Yes
2	300-0540	One area scan	n/a	
		Two area scan	n/a	yes
		One line scan	TTL	
		Two line scan	TTL x 2	yes
		One line scan, one area scan	TTL	yes
3	300-0538	Two line scan	LVDS x 2	yes

I/O options 1, 2, and 3 are associated with the following classes:
1 - ccIO8600LVDS
2 - ccIO8600TTL
3 - ccIO8600DualLVDS

Table 31. Software I/O configuration options

Documentation for the CVL classes that determine the current I/O configuration for the MVS-8600 and MVS-8600e is available in the *CVL Class Reference* which is installed as part of your CVL release. See **ccIOConfig**, **ccIO8600LVDS**, **ccIO8600TTL**, and **ccIO8600DualLVDS**.

To set up an I/O configuration, use code like the following example:

```
ccParallelIO* pio - dynamic_cast<ccParallelIO> (&fg);
pio->setIOConfig(ccIO8600TTL);
```

I/O configuration 1 is the default setting for the MVS-8600 and MVS-8600e, and that configuration is in effect if you do not specify a configuration.

Note If you use an I/O configuration that supports TTL trigger input lines, and you are not using the Cognex I/O connection module to break out the trigger lines from the MVS-8600 or MVS-8600e, then do not leave the input line floating. Connect either ground or a +5 V signal to the input line while the MVS-8600 or MVS-8600e is powered on to prevent the generation of spurious trigger signals.

Using Line Scan Cameras

The MVS-8600 and MVS-8600e support Camera Link line scan cameras. Information about using line scan cameras with the MVS-8600 and MVS-8600e is provided in the section *Acquiring with Line Scan Cameras* on page 129.

Support for Camera Link Cameras

CVL supports a select list of Camera Link base configuration and medium configuration cameras, both area scan and line scan, for use with the MVS-8600 and MVS-8600e. Supported cameras are listed in the *CVL Cameras* document, which is installed as part of your CVL release. *CVL Cameras* shows the Cognex camera cable part numbers and the video format name to use for each supported camera.

Correcting Basler L402k Images

The Basler L402k line scan camera can exhibit striping in the right half of the image. You can calibrate out the striping by following the procedures given in Basler's *L400k User's Manual*. See "Shading Correction" in section 3.6 of the Basler manual.

Apply both DSNU and PRNU shading correction, as described in the Basler manual. The provided calibration method is scene and lighting dependent, and thus must be performed on site.

Once calibrated, be sure to save your correction values to the camera's non-volatile memory as described in the same section of Basler's manual.

Setup for Camera Link Cameras

You must configure Camera Link cameras for optimum performance with the MVS-8600 and MVS-8600e, and with CVL. With other camera types, camera setup is accomplished with DIP switches on the camera body, or with a menu system embedded in the

camera's firmware. By contrast, all cameras that meet the Camera Link specifications provide a firmware setup interface accessible by means of the Camera Link serial communication protocol.

The Camera Link specification provides for a virtual serial port interface that is accessed using RS-232 serial communication protocols over the LVDS Camera Link bus. Some camera manufacturers provide a utility to set up and configure their Camera Link cameras, but not all do.

CVL includes the Cognex Camera Link Serial Communication Utility, which you can use to communicate with and configure any Camera Link camera connected to an MVS-8600 or MVS-8600e frame grabber. To configure your Camera Link cameras you can use the manufacturer's utility, if provided, or the Cognex-provided utility.

One-Time Setup Steps for Camera Link Cameras

If you purchase a supported Camera Link camera from Cognex, the camera is delivered ready to use with the MVS-8600 or MVS-8600e. If you acquire your cameras elsewhere, you must configure them for use with the MVS-8600 or MVS-8600e as described in this section.

This section provides the steps to set up a Camera Link camera using the Cognex Camera Link Serial Communication Utility. Later sections describe in more detail the utility itself and the format of command set files.

These steps presume you have:

- Installed CVL, specifying the installation of the MVS-8600 device driver
- Installed an MVS-8600 or MVS-8600e frame grabber
- Connected a supported Camera Link camera to the MVS-8600 or MVS-8600e
- Stopped image acquisition by disabling triggers, if you have a CVL-based acquisition running at the same time.

To set up a Camera Link camera, follow these steps:

1. In Windows Explorer, navigate to *%VISION_ROOT%\cogclser*
2. Identify the Camera Link command file (CLC file) appropriate for your camera. For example, for the Basler A202k camera, locate this CLC file:

   ```
   basler_a202k_8bit.clc
   ```

   Cognex provides a CLC file for every supported Camera Link camera.
3. Start the Camera Link utility by double-clicking *cogclserial.exe.*
4. Click the **Load** button, and select the CLC file identified in step 2.
5. Specify a camera port number in the **Camera Link Port** field, as follows:

- For one MVS-8601 always specify 0.
- For one MVS-8602 or MVS-8602e, specify 0 or 1 to designate the camera connected to physical port 0 or 1, respectively.
- For cameras connected to a second MVS-8600 or MVS-8600e, specify 2 or 3 to designate cameras connected to physical ports 0 or 1, respectively.

6. Click the **Connect** button.

   On successful connection to the designated camera, the **Send** button is activated.

7. Click the **Send** button to send the commands in the loaded CLC file to the camera.
8. Click **OK** to exit the utility.

Ongoing Setup for Camera Link Cameras

If you want to adjust camera settings such as gain and offset, consult the camera manufacturer's documentation for the syntax of the command to use. You can also look through the comments in your camera's CLC file for suggested commands.

1. Connect to the camera using steps 1, 3, 5, and 6 of the previous section.
2. Type the camera setting command in the **Serial Port Command Window**.
3. Click the **Send** button.
4. If you want to save your camera setting commands for re-use, use the **Save** or **Save As** buttons.
5. Click the **OK** button to exit the utility.

Cognex Camera Link Serial Communication Utility

This section discusses the Windows-based utility included with CVL that allows you to communicate with Camera Link cameras attached to an MVS-8600 or MVS-8600e frame grabber. The Cognex Camera Link utility is provided in both 32-bit and 64-bit versions.

Uses for the Utility

Use the Cognex Camera Link utility for two purposes:

- To perform a one-time configuration of Camera Link cameras for use with the MVS-8600 or MVS-8600e, and with CVL
- To perform camera adjustments such as changes in gain and offset.

Note The Camera Link utility can be run at the same time as a CVL application. However, it is recommended that you disable triggers and stop image acquisition while making camera adjustments, just as you would do while making FIFO property changes.

Note The Camera Link utility is not designed to be multi-process safe.

Utility Description

After installing CVL, locate the Cognex Camera Link utility in the *cogclser* directory of your top-level CVL directory. By default, the utility and its associated files are found in:

```
%VISION_ROOT%\cogclser
```

Start the utility by double-clicking the *cogclserial.exe* icon in this directory. (Both the 32-bit and 64-bit versions are named *cogclserial.exe*.) The utility has a single window as illustrated and explained in Figure 15.

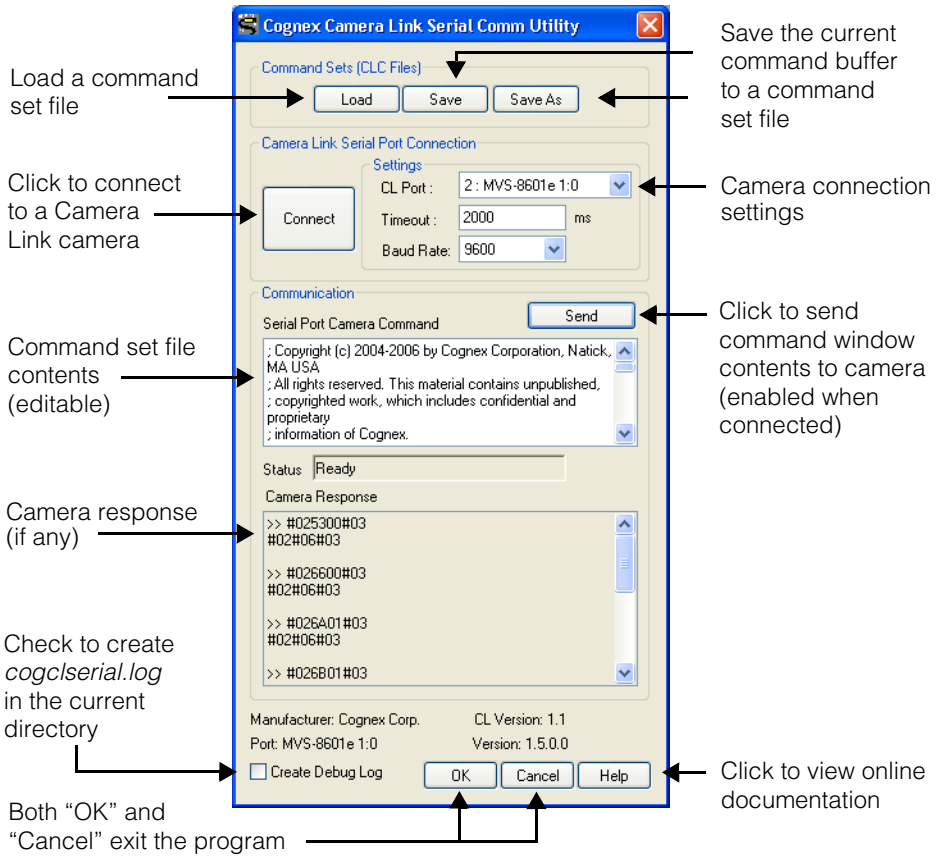


Figure 15. Cognex Camera Link Serial Communication Utility

Camera Link Command Set Files

The directory containing the Cognex Camera Link utility also contains a command set file for each supported video format. Command set files are ASCII text files with a .clc extension. The names of each command set file corresponds to the video format file (CCF file) that supports the Cognex-supported configuration for each camera.

Syntax of CLC Files

The Camera Link specification designates the communication protocol, but does not specify the syntax of commands that control and configure each camera. Thus, the syntax of the setup commands in each CLC file are specific to each manufacturer and camera.

The Cognex Camera Link utility respects the following conventions for commands entered in the command window of the utility.

- Each character typed is treated as plain ASCII text, unless preceded by the pound sign (#).
- Any line beginning with a semicolon (;) is assumed to be a comment and is not transmitted to the camera.
- The pound sign introduces a hexadecimal character. For example, the ASCII carriage return character (0x0D) can be sent as #0D.
- To send a single pound sign to the camera, send ##.
- Each line of text typed in the command window is sent to the camera separately.

Acquiring with GigE Vision Cameras

This section provides information about using GigE Vision cameras with CVL. GigE Vision cameras do not use a frame grabber, but rather connect to a gigabit ethernet adapter board that you plug into your computer's PCI Express bus.

To the CVL acquisition system and to the vision tools, a GigE Vision camera appears to be a frame grabber.

Note Be sure to read the GigE Vision Cameras User's Guide included with your CVL distribution to learn how to create a camera network, how to connect your camera to the network, and how to assign IP addresses to cameras.

Acquisition Using GigE Vision Cameras

Image acquisition using GigE Vision camera is very similar to image acquisition using a Cognex frame grabber. The main difference is that for frame grabber-based acquisition, the frame grabber class object (such as **cc8500**) refers to a frame grabber to which one or more cameras are connected, while for GigE Vision acquisition there is an instance of a **ccGigEVisionCamera** object created to represent each GigE Vision camera connected to the computer system.

Enumerating and Identifying Cameras

The **ccGigEVisionCamera::count()** function returns the number of GigE Vision cameras that are connected to an Ethernet card in your computer or to an Ethernet switch. You can use **ccGigEVisionCamera::get()** to get an individual camera.

If the cameras have not yet finished negotiating their connection with the network they will not be included in the count, and you will not be able to get a reference to them. Be sure that your cameras are completely powered up and connected to the network before starting your application.

Once you have obtained an individual **ccGigEVisionCamera** reference that refers to a particular camera, the **ccGigEVisionCamera::name()** function returns the name of the particular camera, usually the manufacturer and the camera model.

GigE Vision Video Formats

The video formats supported for GigE acquisition are listed in the *CVL Cameras* document, available in HTML format as part of your CVL release. You use a video format to obtain an acquisition FIFO in exactly the same way as with frame grabber-based acquisition, as shown in the following example:

```
const ccStdVideoFormat& fmt =
     ccStdVideoFormat::getFormat(cmT("Generic GigEVision
(Mono)"));

ccStdGreyAcqFifoPtrh fifo = fmt.newAcqFifo(cam);
```

The generic GigE vision video formats do not specify an image size. If you call the **ccVideoFormat** properties **width()** and **height()**, both will return 1. To determine the actual size of the acquired image, get the region of interest immediately after creating the acquisition FIFO like this:

```
ccPelRect imageSize(fifo->properties().actualRoi());
```

Image Acquisition

Once you have created an acquisition FIFO, you use it in exactly the same way that you use a frame grabber-based FIFO to acquire images.

GigE Vision Camera Feature Support

Many commonly used GigE vision camera features are supported directly through CVL image acquisition properties.

For features that are not supported directly by CVL you can use the feature read/write functions such as **ccGigEVisionCamera::writeIntegerValue()**.

The following GigE Vision camera features are supported directly through CVL image acquisition properties:

GenICam Feature	CVL Property	Comments
AcquisitionMode AcquisitionStart AcquisitionStop		Controlled by the combination of **triggerModel()**, **triggerEnable()**, and **start()** functions.
PixelFormat		Based on video format used.
TriggerMode	ccTriggerModel	Based on current **triggerModel()**.
ExposureTimeAbs ExposureTimeRaw	ccExposureProp	

Table 32. GenICam features supported by CVL

GenICam Feature	CVL Property	Comments
BlackLevelRaw GainRaw	ccContrastBrightnessProp	
OffsetX OffsetY	ccRoiProp	
Width Height	ccRoiProp	

Table 32. GenICam features supported by CVL

One case where you would want to use the feature read/write functions directly is to control strobing. Not all GigE Vision cameras support strobing, and those that do, don't all do it the same way. The following example shows how you might set up strobing for a Basler camera.

```
// Strobe output for a Basler camera, configure time to activate
// at beginning of exposure.
fg.writeEnumValue(cmT("TimerSelector"), cmT("Timer1"));
fg.writeEnumValue(cmT("TimerTriggerSource"),
                                  cmT("ExposureStart"));

// Strobe delay (in uSec):
fg.writeDoubleValue(cmT("TimerDelayAbs"), 100);

// Strobe duration (in uSec):
fg.writeDoubleValue(cmT("TimerDurationAbs"), 200);

// Output timer on line 2
fg.writeEnumValue(cmT("LineSelector"), cmT("Out2"));
fg.writeEnumValue(cmT("LineSource"), cmT("TimerActive"));
```

To learn more about accessing camera-specific features, see the sample code *%VISION_ROOT%\sample\cvl\gige_features.cpp*.

Debugging GigE Vision Applications

GigE Vision cameras require that the application periodically send pediodic network packet, called a heartbeat, to the camera to indicate that the application is communicating with the cameras. If the camera does not receive a hearbeat pulse before a timeout period is over, it assumes that the connection has been lost.

The default heartbeat timeout period is 3 seconds.

When you debug GigE Vision applications, you will want to increase the heartbeat timeout interval so that the camera does not go off line while you're in a breakpoint. You can add this code to your program:

```
#ifdef _DEBUG // defined for MS Visual Studio debug builds
   // Set timeout to 1 minute (unit is mSec)
   fg.writeIntegerValue(cmT("GevHeartbeatTimeout"), 60*1000);
#endif
```

Saving and Restoring Camera State

You can save the state of any features that you change to the camera's memory by using the camera's UserSetSave command like this:

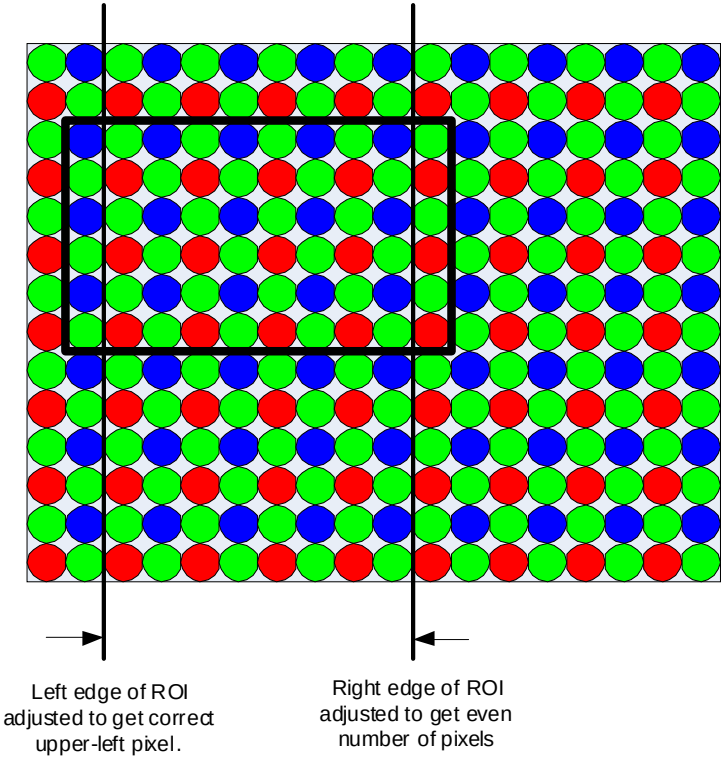
```
fg.writeEnumValue(cmT("UserSetSelector"), cmT("UserSet1"));
fg.executeCommand(cmT("UserSetSave"));
```

When your application accesses a GigE Vision camera for the first time, it uses the camera's UserSetLoad to restore the features you set

Although the features listed in Table 32 on page 150 are also restored, they are immediately set to the default CVL values.

Working with Bayer Images

The CVL routines that convert Bayer format images expect that the image has an even number of pixels and lines and that the upper left pixel is the start of the Bayer pattern. The CVL GigE Vision acquisition software adjusts the region of interest (ROI) to make sure that these conditions are met. The actual ROI of your image may be up to two pixels narrower or shorter then you requested.



Managing GigE Vision Bandwidth Use

If you are using more than one camera, it's possible that the cameras' data rate will exceed the bandwidth of the GigE network. To manage bandwidth in these cases, you may be able to set the cameras' frame rate.

An example of this technique is shown in the sample program *%VISION_ROOT%\sample\cvl\gige_features.cpp*

Acquiring from an Imaging Device

This section provides information about acquiring images using an Imaging Device provided by a third-party camera or frame grabber manufacturer.

The CVL Imaging Device interface provides a way for manufacturers of cameras, frame grabbers, and other devices that generate digital images to provide CVL programs with direct access to the images generated by those devices.

Image acquisition using an Imaging Device is similar to image acquisition using a Cognex frame grabber or GigE Vision camera.

Imaging Device Architecture

The CVL image acquisition is based on a modular architecture, with separate acquisition modules for each type of image acquisition device. CVL comes with a module for each type of Cognex frame grabber, a module for GigE Vision cameras, and a module for Imaging Devices.

The Imaging Device module provides an open interface that third party manufacturers can use to provide transparent access to their image acquisition equipment. The manufacturer does this by implementing an Imaging Device Adapter. This adapter

conforms to the Cognex-defined Imaging Device interface and provides access to the manufacturer's device using the manufacturer's SDK and API.Figure 16 provides an overview of the CVL acquisition system's modularity.

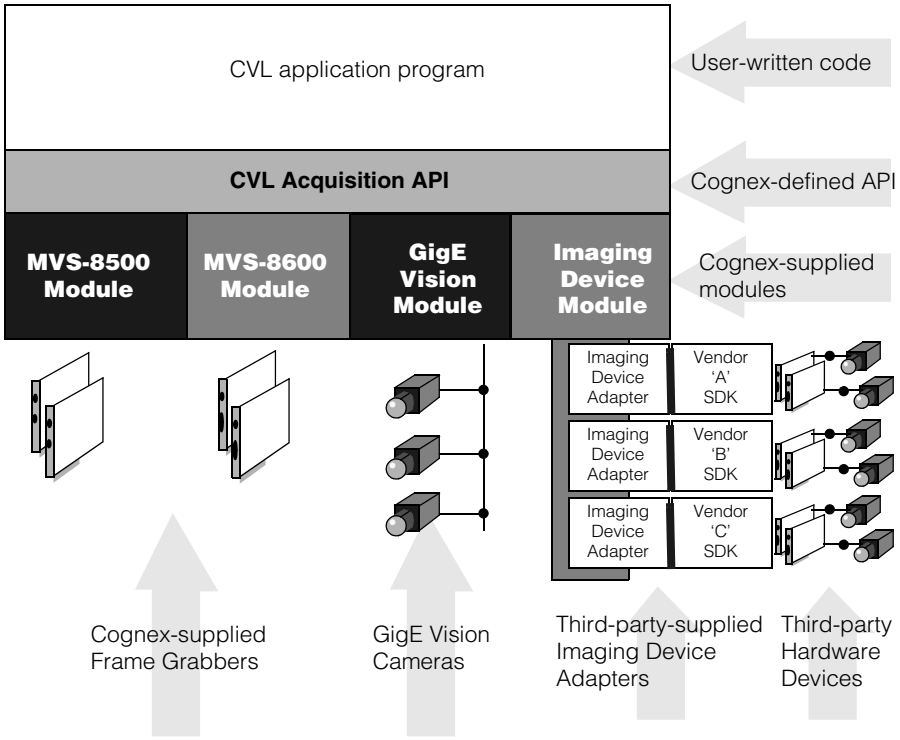


Figure 16. CVL Acquisition Modularity

A single PC can contain any number of Imaging Device Adapters. A CVL application simply uses the normal CVL acquisition interface to query the available devices, instantiate a device, configure it, and acquire images from it.

Note The specific details of the Imaging Device Adapter interface are available from Cognex for the use of qualified third-party developers. For information on obtaining this information, contact your Cognex representative.

Enumerating and Identifying Imaging Devices

The **ccImagingDevice::count()** function returns the number of Imaging Devices that are installed and enabled in your system. You can use **ccImagingDevice::get()** to get an individual device.

Once you have obtained an individual **ccImagingDevice** reference that refers to a particular device, the **ccImagingDevice::name()** function returns the name of the device.

If you expect multiple Imaging Devices to be installed, you should enumerate all the Imaging Devices checking the returned name until you locate the specific device that you wish to use.

Note The Imaging Device architecture treats all devices as having a single acquisition channel. Imaging Device adapters used to provide access to multi-channel hardware will present each channel as an independent device.

Imaging Device Video Format

Imaging Devices do not use video formats, but because the CVL acquisition programming model does, CVL defines a special null video format for use with Imaging Devices. The format name is "**Cognex NullFormat**" and a singleton instance can be obtained by calling the global function **cfNullFormat()**.

To use this null video format to create an Imaging Device acquisition FIFO, you use code similar to the following:

```
// Use the first Imaging Device in the system

ccImagingDevice myID = ccImagingDevice.get(0);

// Create an 8-bit grey acquisition FIFO

ccAcqFifoPtrh fifo = cfNullFormat().newAcqFifoEx(myID);
```

The specific properties of the acquisition FIFO such as the image width and height are determined by the Imaging Device itself upon creation of the FIFO. To determine the image width and height, simply use **ccAcqProps::actualRoi()**:

```
ccPelRect imageSize(fifo->properties().actualRoi();
```

Note The image size may vary from acquisition to acquisition depending on the specific device and how it is configured.

Image Acquisition

Once you have created an Imaging Device acquisition FIFO, you use it in exactly the same way that you any other FIFO to acquire images.

```
// Use the first Imaging Device in the system

ccImagingDevice myID = ccImagingDevice.get(0);

// Create an 8-bit grey acquisition FIFO

ccAcqFifoPtrh fifo = cfNullFormat().newAcqFifoEx(myID);

// Set device properties and trigger model

fifo->triggerEnable(false);
fifo->properties().exposure(20e-3);
fifo->triggerModel(cfManualTrigger());
fifo->triggerEnable(true);

// Acquire a single image using a manual (software) trigger

fifo->start();
ccAcqImagePtrh img = fifo->completeAcq();
ccPelBuffer<c_Uint8> pb = img->getGrey8PelBuffer();
```

Trigger Models

You specify the trigger model for Imaging Device acquisition the same way as for other acquisition: you select from manual, automatic, or semi-automatic triggering. It is the responsibility of the Imaging Device Adapter (supplied by the third party) to properly map the calls you make to **start()** and **completeAcq()** to the appropriate operations. Keep in mind that if you are using a device that does not support external triggers, you should specify manual triggering.

Pixel Formats

Since you are calling the standard image acquisition function **completeAcq()** to complete the acquisition, the returned image will be a **ccAcqImage** object. You can determine the image type (pixel format) of the acquired image by calling **ccAcqImage::formatAcquired()**.

A specific Imaging Device may provide native support for various pixel types. For more information, refer to the documentation supplied with your Imaging Device.

Setting Imaging Device Properties

Imaging Devices can represent any type of underlying device, and the device, in turn, may support an unlimited range of properties. In order to support this range of device capabilities, the Imaging Device interface allows the creator of an Imaging Device Adapter to expose a set of named features. The Imaging Device interface allows you to set the value of each feature.

A limited set of common properties are set through the standard CVL acquisition properties interface. These basic properties are listed in Table 33.

Property	CVL Class and Method
Exposure	**ccExposureProp::exposure()**
Contrast Brightness	**ccContrastBrightnessProp::contrastBrightness()**
Region of Interest	**ccRoiProp::roi()**
Triggering	**ccTriggerProp()** **ccTriggerModel()**

Table 33. Generic acquisition properties supported for Imaging Devices

Device Features

To allow you to set any property supported by any Imaging Device, **ccImagingDevice** provides a pair of functions, **readValue()** and **writeValue()**, that let you get and set the value of any named property supported by the Imaging Device.

Setting Feature Values

The code shown below provides an example of how you use the Imaging Device feature interface.

```
// Use the first Imaging Device in the system

ccImagingDevice myID = ccImagingDevice.get(0);

// Set tachyon pulse width property to 100

myID.writeValue(“tachyonPulseWidth”, “100”);
```

All property names and values are read and set as strings. In all cases, refer to the Imaging Device documentation for the available properties and their legal values.

Note Do not use the custom properties to set a supported generic property listed in Table 33. Always use the standard CVL interface listed in the table.

Persistent Camera Enumeration

In this section, Persistent Camera Enumeration (PCE) is discussed. PCE is relevant only to GigE cameras in this release.

When your CVL application launches, CVL automatically generates a list of all connected GigE Vision cameras. The list orders cameras by their IP addresses that you set with the GigE Vision Configurator utility. You application can get a reference to a camera in this list by calling **ccGigEVisionCamera::get()**.

Normally (without using dynamic discovery), this camera list remains unchanged until the application exits, regardless of any GigE Vision cameras you connect to your production environment.

If your multi-camera application relies on a specific camera being returned by **ccGigEVisionCamera::get()**, problems can occur if the cameras are unpowered or not connected when the application launches.

For example, in an application with two GigE cameras, where CAM1 is normally returned by **ccGigEVisionCamera::get(0)** and CAM2 is returned by **ccGigEVisionCamera::get(1)**, consider what happens if a problem prevents CAM1 from powering on.

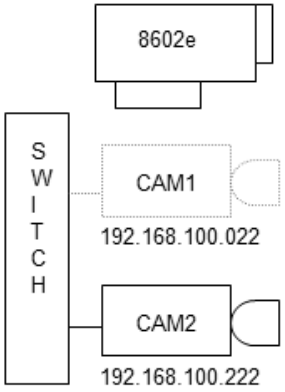


Figure 17. CAM1 not powering on

Without PCE, **ccGigEVisionCamera::get(0)** will return a reference to CAM2, and **ccGigEVisionCamera::get(1)** will throw an exception.

With PCE, the cameras expected by your application are enumerated from the *VPCameraOrder.ini* file. Thus, in the example above, **ccGigEVisionCamera::get(0)** will still return a reference to CAM1, but it will not be functional because the camera is not

actually present. Calling **ccGigEVisionCamera::get(1)** will return a reference to CAM2 as expected. This situation is analogous to if both cameras were present when the application started, but CAM1 was removed while the application was running. If CAM1 is (re)attached, it will become functional without the need to restart the application.

VPCameraOrder.ini Usage

PCE is controlled by the file *VPCameraOrder.ini* located in

%PUBLIC%\Cognex\Common\VPCameraOrder.ini

By default, PCE is disabled. You need to edit the file and set "enable bit = true" to enable PCE. When your application is started after enabling PCE, newly discovered cameras will be "persisted" in the INI file. On subsequent executions, cameras will be enumerated first from the INI file, and then from the network with any additional cameras added to the INI file. In this way, each individual camera will have a fixed index when calling **ccGigEVisionCamera::get()** every time the application is run.

After PCE is enabled and a couple of cameras have been added, the contents of the INI file will look like this for example:

```
[enable PCE]

enable bit=true

; Cognex Corporation. INI file for Persistent Camera Enumeration

(PCE)
;The legal values for enable bit are 'true' and 'false'. PCE is
enabled if the enable bit is 'true'
;PCE doesn't work if the file name is changed
;Modifying the enable PCE bit requires a restart of your
application for the change to take effect
;Do not delete the contents above this line. Make a backup of this
INI file
;in case the file is accidentally deleted

[GigE Camera 1]
IP_Addr=169.254.43.71
Subnet_mask=255.255.0.0
IPCurrentConfig=83886080
MacAddr=00-06-BE-00-43-60
Host_IPAddr=169.254.1.1
```

```
Host_subnet_mask=255.255.0.0
Host_macAddr=00-30-64-22-00-A8
Host_mtu=9000
name=GigE Vision: Baumer Optronic: HXG20
serialNo=0172480612
bigEndian=0

[GigE Camera 2]
IP_Addr=192.168.100.95
Subnet_mask=255.255.255.0
IPCurrentConfig=83886080
MacAddr=00-30-53-0F-81-EB
Host_IPAddr=192.168.100.1
Host_subne_maskt=255.255.255.0
Host_macAddr=00-30-64-22-00-A9
Host_mtu=9000
name=GigE Vision: Basler: acA640-100gm
serialNo=21016299
bigEndian=0

[GigE Camera 3]
IP_Addr=169.253.81.96
Subnet_mask=255.255.0.0
IPCurrentConfig=83886080
MacAddr=00-30-53-0F-5F-50
Host_IPAddr=169.253.31.21
Host_subnet=255.255.0.0
Host_macAddr=00-30-64-22-00-AA
Host_mtu=9000
name=GigE Vision: Basler: acA640-100gm
serialNo=21007440
bigEndian=0
```

Camera Removal

If you want to remove a camera from the camera list, remove it from the INI file, adjust the camera numbering so there are no gaps, and restart the CVL application. For example, if there are three cameras in the INI file and the camera list, and you remove camera 2 without adjusting the camera numbering leaving the INI file with cameras 1 and 3, it would produce an error condition and only camera 1 would be enumerated from the INI file. Changing camera 3 to camera 2 fixes the problem.

Camera Order Change

If you want to change the order of the cameras in the camera list, change their order in the INI file, adjust the camera numbering so that it is in an increasing sequential order and there are no gaps, and restart the CVL application.

Camera Replacement

In CVL releases prior to CVL 8.0, if a camera stopped functioning it was possible to replace the failed camera while the application was running provided the replacement camera was the same model as the failed camera and was programmed with the same IP address. This behavior is still possible with this CVL release, but PCE provides an additional option.

For example, consider the case where CAM1 fails and is replaced with CAM3 with the same IP address:

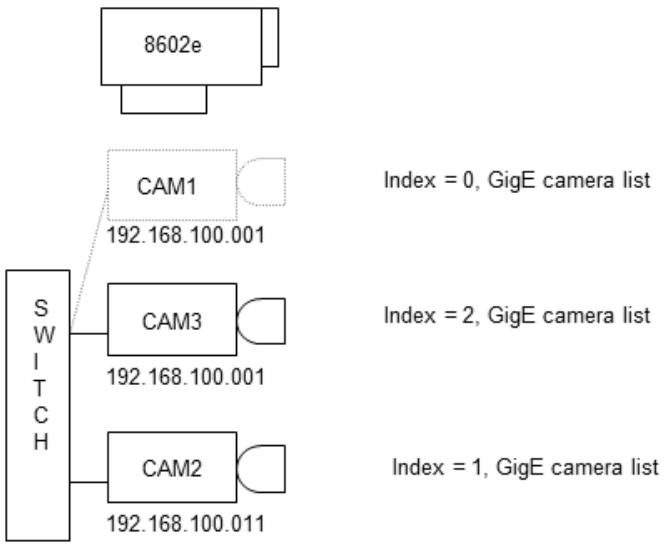


Figure 18. CAM1 got replaced by CAM3

With PCE enabled, when the application restarts CAM3 will be identified as a new camera and added to the end of the list. This happens because CAM3 has a unique MAC address that does not match the MAC address of CAM1. However, when your application attempts to use CAM1, the IP address of CAM1 will be used for communication with the camera. Since CAM3 has the same IP address, it will respond as if it were CAM1, and the application will work (provided it does not try to use CAM3).

If CAM3 is a permanent replacement for CAM1, it is recommended to edit the INI file to remove CAM1 and change CAM3 to be CAM1.

Error Handling

When an error occurs reading the INI file, PCE simply enumerates the cameras it was able to successfully read from the file. Error can occur due to subtle syntax errors introduced when editing the file by hand.

There is currently no mechanism to explicitly determine that an error has occurred and what the specific error is. It is possible to physically remove all cameras, then run your application and note which cameras were successfully enumerated from the INI file.

One way to diagnose problems with the INI file is to make a copy of the INI file, then edit the INI file and remove all contents except for PCE enable:

```
[enable PCE]
enable bit=true
```

Then, run your application and compare the updated INI file to the copy of the INI file. The data written by CVL to the “new” INI file can then be edited as needed.

Dynamic Discovery of Cameras

The **ccBoard::RefreshCameraList()** function allows newly connected cameras to be discovered while an application is running. In the current release, only GigE Vision cameras can be discovered after an application has been started.

Note that newly discovered cameras are added to the end of the **ccGigEVisionCamera** “list” that is accessible via **ccGigEVisionCamera::get()**. Once a camera is in this list, it will never be removed and its location in the list will not change while the application is running. In other words, during the lifetime of an application, **ccGigEVisionCamera::get(n)** will always return a reference to the same camera.

If a camera is physically removed from the system, it will not be removed from **ccGigEVisionCamera** even if **ccBoard::RefreshCameraList()** is called. However, it will no longer be possible to acquire images from the camera until it is physically re-attached.

Dynamic Discovery and Persistent Camera Enumeration

When dynamic discovery is used without Persistent Camera Enumeration enabled, it is possible for cameras to appear in a different order each time the application is run. For example, if only CAM2 is present when the application starts, and then CAM1 is discovered during a call to **ccBoard::RefreshCameraList()**, then

ccGigEVisionCamera::get(0) will return a reference to CAM2 and **ccGigEVisionCamera::get(1)** will return a reference to CAM1. If the application is later restarted with both cameras attached, then it is possible that **ccGigEVisionCamera::get(0)** will return a reference to CAM1 and **ccGigEVisionCamera::get(1)** will return a reference to CAM2.

If Persistent Camera Enumeration is enabled, then newly discovered cameras will be added to the INI file, and the order will not change when the application is restarted.

Camera-Specific Usage Notes

This section provides information on using certain cameras with CVL.

N/A

Frame Grabber Acquisition Usage Notes

This section describes techniques and workarounds specific to a frame grabber or frame grabber family.

MVS-8500 Usage Notes

This section includes notes that apply to the MVS-8504 frame grabber.

Unexpected Strobe Firing

Under the following conditions, a strobe attached to an MVS-8504 frame grabber may fire one unexpected time after you disable strobes:

1. You are using a strobe that fires on low-to-high transitions.
2. Enable strobes and acquire a number of images.
3. Disable strobes.
4. When any of the following events occur, the strobe may flash one more time, even though strobing is now disabled:
    - Using manual or semi trigger models. you call **start()**.
    - Using auto trigger model, you call **triggerEnable(true)**.

If this occurs, the unexpected strobe flash may spoil the first image acquisition after you disable strobing. To avoid this event, use one of the following workarounds:

- Use a strobe that fires on high-to-low transitions.
- Add a 100 ohm pull-down resistor to the strobe line.
- Write your application to discard the first acquisition after disabling strobes.

Displaying Images

4

This chapter describes how to use CVL to display images, in the following major sections:

Some Useful Definitions defines certain terms you will encounter as you read.

Overview describes the general steps involved in displaying images.

Using the Display Classes introduces the display classes that provide drawing environments for different Cognex frame grabbers.

Displaying Pel Buffers shows you how to display pel buffer images that you have acquired or created synthetically.

Color Maps describes the use of color maps for defining colors in images and graphics.

Displaying Color Images describes how to display images that are acquired or generated in color.

Displaying Live Images shows you how to display live video and how to acquire single images while displaying live video.

Customizing Image Display Environments describes how to add custom functionality to your CVL application by deriving your own display class from **ccDisplay**.

Some Useful Definitions

The following terms may be useful in reading this chapter.

color map A table of RGB values used to map pixel values to colors for display.

display A data structure used to present the image stored in a pel buffer in a window.

image layer A buffer that may contain an image and static and manipulable graphics.

interpolated zooming A special algorithm used when magnifying an image in order to reduce "blockiness" and to smooth the image by interpolating neighboring pixel values.

live display The display of camera-acquired images using a frame grabber at a rate set by the acquisition FIFO.

magnification The up scale or down scale of an image and associated graphics.

non-integer scaling The up scale or down scale of an image and associated graphics using a floating point value (for example, 0.31 or 1.71).

overlay layer A buffer used to render non-destructive static and manipulable graphics. When displayed, the overlay layer is on top of the image layer and uses a pass-through value to reveal the image layer underneath it.

pel buffer An array of pixel values that represent an image. In CVL, these are objects of type **ccPelBuffer**.

pseudo-live display A user-written code sequence (that is, a loop) that displays camera-acquired images or synthetic pel buffers at a rate set by the user, usually to mimic live display.

Overview

The Cognex Vision Library allows you to display live and acquired images. The basic steps to display images are:

1. Create a display object.
2. Set the display object's attributes.
3. Display the image.
4. Release the display object.

The example that follows shows these steps for a Cognex MVS-8100M frame grabber running under Windows and using a **ccDisplayConsole** window.

Note The following example is designed to illustrate several techniques in a small amount of space. In an actual vision application, use only those techniques that are appropriate for your application.

```
#include <ch_cvl/vp8100.h>
#include <ch_cvl/acq.h>
#include <ch_cvl/vidfmt.h>
#include <ch_cvl/windisp.h>
#include <ch_cvl/xform.h>

int cfSampleMain(int, TCHAR** const)
{
  // Find the frame grabber
  cc8100m& fg = cc8100m::get(0);
  const ccStdVideoFormat& fmt =
       ccStdVideoFormat::getFormat(cmT("Sony XC-ST50 640x480"));
  ccAcqFifoPtrh fifo = fmt.newAcqFifoEx(fg);

  // Step 1: Create a display console
  ccDisplayConsole *display;
  display = new ccDisplayConsole(cmT("Display Console"),
    ccIPair(20,20), ccIPair(480,360));

  // Step 2: Set the display console's attributes
  display->mag(-2);
  display->closeAction(ccDisplayConsole::eCloseDisabled);

  // The next three settings are the defaults
  display->showToolBar(true);
  display->showScrollBar(true, true);
  display->showStatusBar(true);
```

```
  // Use the Status Bar Text for any text you choose
  display->statusBarText(cmT("Ready"));

  // Start an acquisition
  fifo->start();

  // Get the acquired image
  ccAcqImagePtrh img = fifo->completeAcq(
                 ccAcqFifo::CompleteArgs().acquireInfo(&info));
    if (!img.isBound())
  {
    MessageBox(NULL, cmT("Acquisition failed"),
           cmT("Display Error"), MB_OK);
    return 0;
  }
  // Set the transformation object
  // In this example, 40 image units (pels)
  // map to 1 client coordinate unit
  cc2Xform xform(cc2Vect(0,0), ccRadian(0), ccRadian(0),
         40.0, 40.0);

  // Step 3: Display the image
  display->image(img, &xform);

  // Draw some graphics into the image layer
  ccUITablet tablet;
  ccPoint where(2,2);
  tablet.drawPointIcon(where, ccColor::red);
  tablet.draw(cmT("Point (2,2)"), where, ccColor::blue,
         ccColor::yellow, ccUIFormat());
  display->drawSketch(tablet.sketch(),
         ccDisplay::eDisplayCoords);
  display->drawSketch(tablet.sketch(), ccDisplay::eImageCoords);
  display->drawSketch(tablet.sketch(), ccDisplay::eClientCoords);
  // Display message box to pause application
  MessageBox(NULL, cmT("Display complete"),
            cmT("Display Sample"),MB_OK);

  // Step 4: Release display object
  fifo = ccAcqFifoPtrh(0);          // delete the fifo...
  delete display;                   // ... before the display
  return 0;
}
```

This example produces the display show in Figure 19 below.

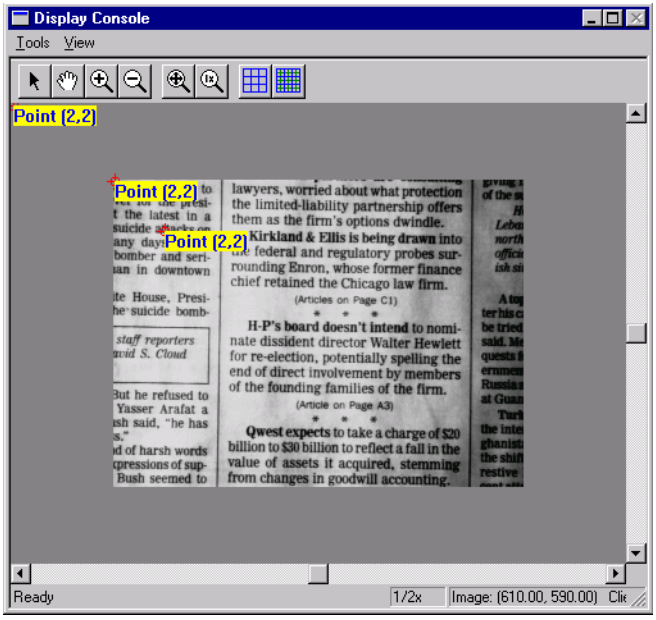


Figure 19. Example display

The example creates a point icon and a label at the same location (2,2). The location specifies the upper left-hand corner of the label. The point icon at 2,2 is small and difficult to see in this figure. The example then displays the point icon and the label using three different coordinate systems to illustrate their differences. *eDisplayCoords* displays the pair at location 2,2 of the display window. *eImageCoords* displays the pair at location 2,2 of the image. *eClientCoords* displays the pair at location 80,80 of the image because in our example, one client unit is equal to 40 pixels for both x and y. (Note that the label size is not scaled by the client coordinate transform).

CVL Display Programming Requirements

All CVL Display applications must provide their own mutex locks or semaphores around all calls into CVL Display code. Only one thread at a time is allowed to make calls to CVL Display routines.

Using the Display Classes

CVL uses classes derived from **ccDisplay** to provide a display environment in which you can display pel buffers. Pel buffers are images that you acquire or synthetic images that you create. These display classes are summarized in Table 34.

ccDisplay-derived class	Description
ccDisplayConsole	A full-featured display and drawing environment for use with Microsoft MFC for execution on a Windows host computer. Tool bars, status bar, and scroll bars are provided by MFC. Images and graphics are rendered on the host computer.
ccWin32Display	A display class based on the Microsoft Win32 API for execution on a Windows host computer. The display window contains only features you add with code in your application. Images and graphics are rendered on the host computer.

Table 34. ccDisplay-derived classes.

Figure 20 shows the display class derivation hierarchy. The **ccDisplay** base class is platform independent and contains all the basic functions needed to display images and graphics. For example, it contains methods for adding and removing images and graphics, and controls for zooming and panning.

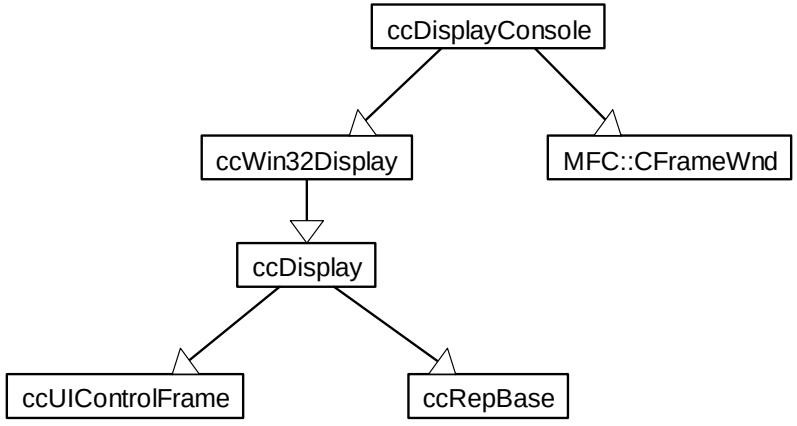


Figure 20. ccDisplay class inheritance hierarchy

The derived class **ccWin32Display** is intended for Microsoft Windows applications that do not use MFC, or for any application that requires a simple display window. **ccWin32Display** windows do not include a title bar, menu bar, tool bar, and status bar like those built into **ccDisplayConsole** windows. The class does provide scroll bars and methods you can call for zooming, panning, and other features.

The **ccDisplayConsole** class is derived from **ccWin32Display** and also from the MFC class **CFrameWnd**. MFC provides all the GUI features Windows users are used to seeing, such as title bar, tool bars, status bars, and scroll bars. You can easily customize your display window with tools. menus, and controls by calling MFC functions.

You can derive your own display class from any of these display classes to create your own custom display environment.

Much of the display API is located in the **ccDisplay** base class and is available to all derived display classes. All of the display classes have the following capabilities:

- Display images (pel buffers)
- Display live images from Cognex frame grabbers
- Add static graphics to the display
- Add interactive graphics to the display
- Automatic conversion to the proper desktop depth
- Zoom, expand, sample, and interpolate
- Option to clip the image
- Apply a filter color map
- Mouse support

Table 35 summarizes the **ccDisplay** member functions.

Function	Set	Get	Description
image()	x		Display an image.
displayFormat()^{v}		x	Returns the display format.
hasImage()^{v}		x	Returns true if the display has an image.
hasImage8()^{v}		x	Returns true if the display has an 8-bit image.
hasImage16()^{v}		x	Returns true if the display has a 16-bit image.
hasImage32()^{v}		x	Returns true if the display has a 32-bit image.
imageSize()^{v}		x	Returns the image size.

Table 35. ccDisplay member functions

Function	Set	Get	Description
imageOffset()[v]		x	Returns the image offset.
addShape()	x		Adds a UI shape to the image.
removeShape()	x		Removes a UI shape from the image.
drawSketch()	x		Draws a sketch in the image.
eraseSketch()	x		Erases the currently displayed sketch.
getSketch()		x	Returns the currently displayed sketch.
disableDrawing()	x		Disable drawing in the image.
enableDrawing()	x		Enable drawing in the image.
drawingDisabled()		x	Returns true if image drawing is disabled.
pan()	x	x	Pan the image by delta-x and delta-y.
maxPan()	x	x	Defines the maximum **pan()** x,y values.
minPan()	x	x	Defines the minimum **pan()** x,y values.
defaultPan()	x	x	True if panning is limited to the image size.
panDelta()	x		Pan the image by delta-x and delta-y.
mag()	x	x	The image magnification factor (integer).
magExact()	x	x	The image magnification factor (double).
fit()	x		Fits the image to the display size (integer scale).
fitExact()	x		Fits the image to the display size, exactly.
grid()	x	x	Defines the number of pixels per grid.
gridColor()	x	x	Defines the grid color.
subpixelGrid()	x	x	The number of subpixel grid lines per pixel.
subpixelGridColor()	x	x	The subpixel grid color.
colorMap()	x	x	Map from 8-bit pseudo color to RGB.
colorMapEx()	x	x	Map for non-8-bit desktops.

Table 35. ccDisplay member functions

Function	Set	Get	Description
clearColorMap()	x		Sets default color map.
mouseMode()^{v}	x	x	Defines mouse mode.
selectAreaStart()^{v}	x		Start a selection rectangle in the image.
selectAreaEnd()^{v}	x		End a selection rectangle in the image.
selectArea()^{v}	x		Starts and ends a selection rectangle.
selectPointStart()^{v}	x		Start a point selection in the image.
selectPointEnd()^{v}	x		Ends a point selection in the image.
selectPoint()^{v}	x		Starts and ends a point selection.
image8()		x	Returns an 8-bit grey scale image bound to this display.
imageRGB16()		x	Returns a 16 -bit color image bound to this display.
imageRGB32()		x	Returns a 32 -bit color image bound to this display.
canBlink()^{v}		x	Returns true if the display supports blinking colors.
enableBlink()^{v}	x		Start blinking.
disableBlink()^{v}	x		Stop blinking.
blinkRate()^{v}	x	x	The blink rate.
blinkColor()^{v}	x	x	The colors that blink.
startLiveDisplay()^{v}	x		Start live display.
stopLiveDisplay()^{v}	x		Stop live display.
isLiveEnabled()^{v}		x	Returns true if live display is active.

(^{v}) = Virtual functions that must be overridden in a derived class.

Table 35. ccDisplay member functions

The remainder of this section covers how to use the **ccDisplayConsole** and **ccWin32Display** display classes.

ccDisplayConsole User's Guide

This section describes how to use the various GUI features of a CVL display console window.

The **ccDisplayConsole** class provides a full-featured display and drawing environment and graphical user interface you can use when programming with MFC for execution on a Windows host computer. **ccDisplayConsole** is an MFC-derived object that provides a client window and built-in Windows support for tool bars and scroll bars with controls for panning, zooming, and grid lines. An optional status bar provides text status, magnification status, and mouse location. Images are displayed in the client window, which is resized with the frame window. The **ccDisplayConsole** display is commonly referred to as the *display console*.

Figure 21 shows the components of a typical CVL display console window.

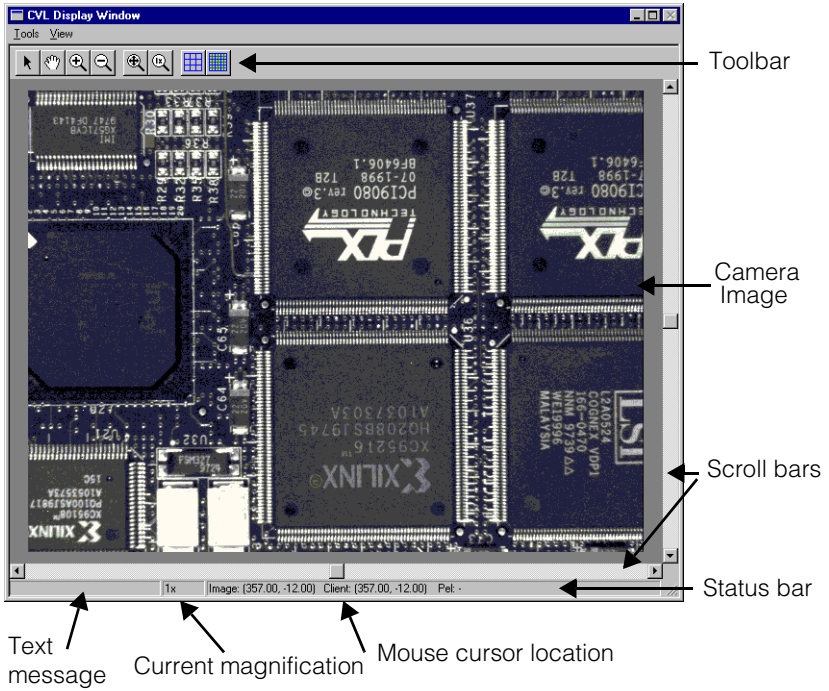


Figure 21. CVL Display Console

The optional status bar shows the current magnification, the location of the mouse in image and client coordinates, and the value of the pixel at the mouse location. The text area on the left of the status bar allows you to display a short text message.

The following section describes the tool bar menus and icons.

Toolbar and Menu Reference

The CVL display console's toolbar buttons shown in Figure 21 duplicate the commands in the Tools menu. Table 36 shows the functions supported by both menu commands and toolbar buttons.

Menu Command	Toolbar Button	Function
Tools -> Pointer		Makes the cursor a standard pointing arrow which performs no action on the displayed image.
Tools -> Pan		Makes the cursor a panning hand. Left-click this cursor to move the displayed image within the borders of the display console window.
Tools -> Zoom In		Makes the cursor a plus sign. Click the left mouse button to magnify the image 1X per click. Click the right mouse button to reduce 1X per click.
Tools -> Zoom Out		Makes the cursor a minus sign. Click the left mouse button to reduce the image 1X per click. Click the right mouse button to magnify 1X per click.
Tools -> Fit Image		When selected, the image zooms out to fit the current display console window size, using the magnification level that allows the best fit.
Tools -> Zoom 100%		When selected, the image is zoomed to 100% size, independent of the display console's window size.
Tools -> Pixel Grid		When selected, a grid of blue lines, one pixel in width and height, is overlaid on the image. The grid is not visible unless the image is magnified several times.

Table 36. CVL display console's toolbar and menus

Menu Command	Toolbar Button	Function
Tools -> Subpixel Grid		When selected, a grid of green lines, one quarter-pixel in width and height, is overlaid on the image. The image must be magnified many times for the subpixel grid to be visible.
View -> Toolbar	None	Toggles the display of the toolbar.
View -> Status Bar	None	Toggles the display of the status bar.
View -> Horizontal Scroll Bar	None	Toggles the display of the horizontal scroll bar.
View -> Vertical Scroll Bar	None	Toggles the display of the vertical scroll bar.

Table 36. CVL display console's toolbar and menus

Using ccDisplayConsole

The **ccDisplayConsole** class provides a full-featured display and drawing environment and a GUI that you can use when writing your program under Microsoft MFC. This section describes how to use the **ccDisplayConsole** class in your programs. The class has relatively few member functions of its own, which are summarized in Table 37 below. Most applications use primarily the inherited MFC functionality from **CFrameWnd** but the inherited CVL methods from **ccWin32Display** and the **ccDisplay** base class are also available. **ccDisplay** functions are summarized in Table 35 on page 174 and **ccWin32Display** functions are summarized in Table 38 on page 189. The inherited **CFrameWnd** functionality is covered in Microsoft documentation.

Function	Set	Get	Description
showStatusBar()	x		Show/hide the display status bar.
showToolBar()	x		Show/hide the display tool bar.
statusBarText()	x		Displays a text message in the status bar.

Table 37. ccDisplayConsole member functions

Function	Set	Get	Description
closeAction()	x	x	What to do when the user clicks the display close button or chooses the close command.
RecalcLayout()	x		How to position your custom bars after a resize.

Table 37. ccDisplayConsole member functions

An example of using **ccDisplayConsole** in an application is included in the sample program *disp.cpp* included with your CVL release. Running sample programs is discussed in *Programming Examples* on page 41. Also see *Console Display Sample* on page 43.

If you are not programming under MFC you may wish to use the **ccWin32Display** class discussed in *Using ccWin32Display* on page 187.

Creating Display Consoles

To create a display console, create an instance of **ccDisplayConsole**. However, you cannot create a **ccDisplayConsole** at static initialization time. That means you cannot make a **ccDisplayConsole** a global variable or a static variable of a function.

Typically, a display console is created with the **new** operator or as a local variable.

The example program on page 170 creates the display console like this:

```
ccDisplayConsole *display;
display = new ccDisplayConsole(cmT("Display Console"),
  ccIPair(20,20), ccIPair(480,360));
```

Using this constructor for **ccDisplayConsole** creates a window named “Display Console” 480 pixels wide, 360 pixels high, and located at point (20,20) on the desktop.

You can create as many display consoles as your application requires.

Although you can create a display console as a local variable, this is not recommended, especially if you choose to have the window deleted when the user clicks the close button. See the next section for more information.

The **ccDisplayConsole** constructors let you specify additional information about the window such as whether it is initially visible or initially minimized.

Releasing Display Consoles

When your program is finished using the display console, if you allocated it dynamically with **new**, remember to delete it. If you are using a display console to display pel buffers acquired through an acquisition FIFO, be sure to delete the FIFO first:

```
fifo = ccAcqFifoPtrh(0); // Delete the fifo
delete display;                 //Delete the display console
```

If you prefer, you can use the C++ standard library's **std::auto_ptr** template class to define a display console pointer that is deleted automatically when the function where it is defined returns.

Setting Display Console Attributes

Since the display console provides a full-featured user interface, it offers more control over its appearance than other CVL displays. In many cases, the default attributes of the display console are sufficient. The example program on page 170 sets some of them and the following code examples illustrate using the tool bar, scroll bars, and status bar.

```
display->showToolBar(true);
display->showScrollBar(true, true);
display->showStatusBar(true);
display->statusBarText(cmT("Ready"));
```

Close Button Action

The **closeAction()** function lets you specify what will happen when the user clicks the display console's **Close** button or chooses the **Close** command from the pull down menu. For example,

```
display->closeAction(ccDisplayConsole::eCloseDisabled);
```

disables the **Close** button. You can also choose to have the window hidden or deleted when the user clicks the **Close** button.

Note If you use **ccDisplayConsole::eCloseDelete**, which deletes the window when the **Close** button is clicked, you must not create the display console as a local variable.

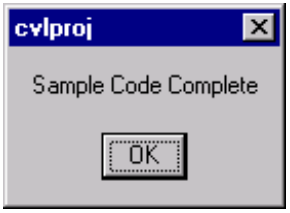
Windows Message Handler

ccDisplayConsole does not have a Windows message handler so when you use this class in your application you must provide one so that the window will respond to window events such as mouse clicks and keyboard events. Without a message handler the **ccDisplayConsole** window will be unresponsive. You will be unable to move or resize it and the tool bar may not appear.

A simple way to add a message handler is to call **MessageBox()** in a separate thread. For example,

```
MessageBox(NULL, cmT("Sample Code Complete"), m_pszExeName,
                                                        MB_OK);
```

displays the following window:



The thread blocks until you click OK. While it is blocking, it will process window events for your application.

A more elegant solution is to add a separate thread to your program to process window events. The following code segment shows how to do this using a *message pump* routine.

```
static ccDisplayConsole *cgDisplayPtr = 0;

static void cfMessagePump(void *arg)
{
  ccSemaphore *sema = reinterpret_cast<\ccSemaphore*\>(arg);
  try
  {
    // important: construct the console in the thread that has the
    // message pump
    ccDisplayConsole *display = new
      ccDisplayConsole(ccIPair(cDefImageWidth, cDefImageHeight));
    display->closeAction(ccDisplayConsole::eCloseHide);
    cgDisplayPtr = display;

    // let other thread know we completed initialization
    sema->unlock();

    MSG msg;

    // The function PeekMessage returns immediately and does not
    // block, unlike GetMessage which will block on the next event
    // for this thread.
    //  while(PeekMessage(&msg,0,0,0,PM_REMOVE))

    // wait here for any messages
    while(GetMessage(&msg,0,0,0))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
```

```
    cgDisplayPtr = 0;
    delete display;
  }
  catch (cc_Resource::BrokenLock&)
  {
  }
}


int cfSampleMain(int, TCHAR** const argv)
{
  ccSemaphore sema(0,1);
  ccThreadID pumpID_;
  try
  {
    pumpID_ = cfCreateThread(cfMessagePump,&sema);
  }
  catch(...)
  {
    cogOut << cmT("Thread error") << cmStd endl;
  }

  // wait for the message pump to get up and running;
  sema.lock();

  // ...
  // you can place your code here

  // for example
  // cgDisplayPtr->startLiveDisplay( ... );
  // ccTimer::sleep(60);
  // cgDisplayPtr->stopLiveDisplay();
  // ...


  // this will shut down the message pump thread, destroying
  // the console
  if(pumpID_)
  {
    // this will shut down the message pump
```

```
    // calling PostMessage returns immediately,
    // calling SendMessage blocks until handled
    cgDisplayPtr->PostMessage(WM_QUIT,0,0);
    cfWaitForThreadTermination(pumpID_);
  }
}
```

Changing the Cursor

When you change the cursor position in a **ccDisplayConsole** application with one of the following commands:

```
display->root().cursor(ccUIRootObject::eHandOpened) or,
```

```
display->mouseMode(...)
```

(For other cursor choices, see the **Cursor** enum in *uifrmwrk.h*).

The cursor position does not change until you move the mouse. To force the cursor to change immediately, send a **WM_SETCURSOR** message to the window like this:

```
PostMessage(display.window(), WM_SETCURSOR, 0, 0);
```

Or use the following code:

```
POINT pt;
GetCursorPos(&pt);
SetCursorPos(pt.x, pt.y);
```

Tool, Status, and Scroll Bars

The following functions ensure that the Toolbar, the Status Bar, and both scroll bars are visible.

```
display->showToolBar(true);
display->showScrollBar(true, true);
display->showStatusBar(true);
```

These are the default settings for display consoles. They are included in the example program for illustration.

The following function sets the text in the leftmost panel of the Status Bar.

```
display->statusBarText(cmT("Ready"));
```

You can use this area to display your own messages.

Using Custom Fonts

If you plan to display text in the display console using a font other than the system font, you can create its font table when you set the other display console attributes. To set up a font table, you create a vector of **HFONT** elements using the Windows **CreateFont()** function and pass it to **ccDisplayConsole::fontTable()**. Later, when you want to draw text, you specify the font by using its index in the font table.

```
cmStd vector<HFONT> ftable;
ftable.resize(2);
ftable[0] = CreateFont(8, 0, 0, 0, FW_REGULAR, false
  false, DEFAULT_CHARSET, OUT_DEFAULT_PRECIS,
  CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
  DEFAULT_PITCH, "MS Sans Serif");
ftable[1] = CreateFont(8, 0, 0, 0, FW_REGULAR, false, false,
  false, DEFAULT_CHARSET, OUT_DEFAULT_PRECIS,
  CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
  DEFAULT_PITCH, "Courier");
display->fontTable(ftable);
```

Display Console Message IDs

If your application does not respond to Windows message IDs, there may be conflicts between the event IDs used in your application windows and those used in CVL display consoles. The following table lists the message IDs used in **ccDisplayConsole**.

Message	ID
Used by winres.rc	
ID_INDICATOR_DEFAULT_COORDS	501
ID_INDICATOR_TEXT	505
ID_INDICATOR_DEFAULT_ZOOM	506
IDC_POINTER	600
IDC_CROSSHAIR	601
IDC_MAGPLUS	602
IDC_MAGMINUS	603
IDC_HANDOPENED	604
IDC_HANDCLOSED	605
IDC_INSERTION	606

Message	ID
IDC_HORZ_SCROLLBAR	607
IDC_VERT_SCROLLBAR	608
IDC_DONE	609
IDC_DIAGTREE	610
IDC_DIAG_MENU	611
IDC_DIAG_OBJECT	612
IDD_DIAG_TREE	613
IDB_TREE_STATE	614
IDR_IMAGECONSOLE	628
IDR_DISPTYPE	629

Used by diagnostics display in cogdiag.dll

IDD_DIAG_ABOUTBOX	700
IDR_DIAG_MAINFRAME	728
IDR_DIAG_DIAGWITYPE	729
IDR_DIAG_TOOLBAR	730

Used for debugging only

IDM_TEST1	32771
ID_CONSOLE_HIDE	40501
ID_TOOLS_POINTER	40502
ID_TOOLS_PAN	40503
ID_TOOLS_ZOOMIN	40504
ID_TOOLS_ZOOMOUT	40505
ID_TOOLS_FITIMAGE	40506

Message	ID
ID_TOOLS_ZOOM100	40507
ID_HELP_ABOUTIMAGECONSOLE	40508
ID_TOOLS_REFRESH	40509
ID_TOOLS_PIXELGRID	40510
ID_TOOLS_SUBPIXELGRID	40511
ID_VIEW_HORIZ_SCROLLBAR	40512
ID_VIEW_VERT_SCROLLBAR	40513
ID_SHOW_LASTDIAG	40514
IDS_FRAMETITLE	57610

Using ccWin32Display

The **ccWin32Display** class provides a display window for images and graphics you can program using the Microsoft Win32 API, for execution on a Windows host computer. **ccWin32Display** does not depend on MFC, but it can be embedded in an MFC application. (For an example, see *Win32 Display Sample* on page 43).

Note If your CVL application or DLL is linked against static MFC libraries, deleting a **ccWin32Display** object results in a debug assertion. By design, CVL does not support static linking against MFC. You must specify Use MFC In a Shared DLL in the Visual C++ project settings when creating a CVL project.

ccWin32Display windows do not include a title bar, menu bar, tool bar, and status bar like those built into **ccDisplayConsole** windows. The class does provide scroll bars and methods you can call for zooming, panning, and other features.

If you are building a **ccWin32Display** application that does not use MFC and you want to ensure the application includes no MFC libraries, define the macro **cmNoMFCDependency** in the project's preprocessor settings to avoid including MFC header files.

ccWin32Display windows are typically used in custom GUI applications to display images. Figure 22 is an example of an such an application that can acquire images and perform live display.

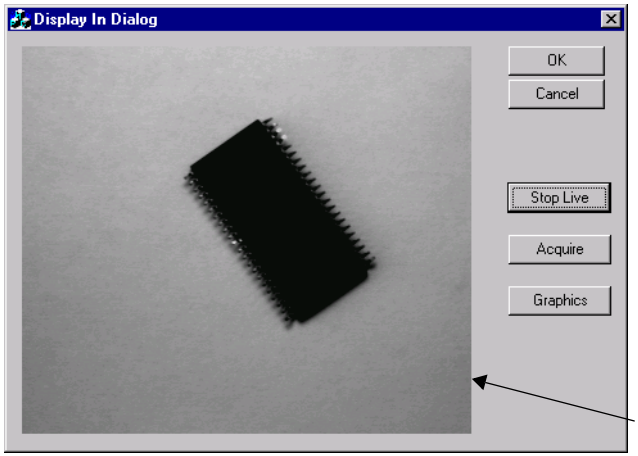


Figure 22. ccWin32Display example

An example of using **ccWin32Display** in an application is included in a sample program that is part of your CVL release. See *Win32 Display Sample* on page 43.

If you are programming under MFC, you may wish to use the **ccDisplayConsole** class which provides a full-featured display and drawing environment and a graphical user interface, all through MFC. Tool bars, status bars, and many other GUI features are provided by MFC that may improve your application's user interface. **ccDisplayConsole** is discussed in *Using ccDisplayConsole* on page 179.

The **ccWin32Display** display class has the following capabilities:

- Display images (pel buffers)
- Display live images from Cognex frame grabbers
- Add static graphics to the display
- Add interactive graphics to the display
- Automatic conversion to the proper desktop depth
- Zoom, expand, sample, and interpolate
- Option to clip the image
- Apply a filter color map
- Mouse support

In addition to the inherited functions described in Table 35 on page 174, **ccWin32Display** includes the functions summarized in Table 38 below.

Function	Set	Get	Description
window()	x	x	The window where the display appears.
fontTable()	x	x	The font table used in the display.
showScrollBar()	x		Sets scroll bars on/off.
horzScrollbarEnabled()		x	Horizontal scroll bar visible?
vertScrollbarEnabled()		x	Vertical scroll bar visible?
enableOverlay()	x	x	Enable/disable overlay plane.
chromaKey()	x	x	Chroma key color. (Deprecated)
getDC()		x	Get a Win32 device context.
releaseDC()	x		Release device context.
selectAreaStart()^{vo}	x		Start a select display area.
selectAreaEnd()^{vo}	x		End a select display area.
selectArea()^{vo}	x		Combines start/end above.
selectPointStart()^{vo}	x		Start a point selection.
selectPointEnd()^{vo}	x		End a point selection.
selectPoint()^{vo}	x		Combines start/end above.
displayFormat()^{vo}		x	Get display format.
startLiveDisplay()^{o}	x		Start live display.
stopLiveDisplay()^{vo}	x		Stop live display.
isLiveEnabled()^{vo}		x	Is live display active?
liveFrameRate()^{v}		x	Get live display frame rate.
hasImage()^{vo}		x	Does the display have an image?
imageSize()^{vo}		x	Get the image size.
imageOffset()^{vo}		x	Get the image offset.

Table 38. ccWin32Display member functions

Function	Set	Get	Description
toolsPopupMenuEnabled()	x	x	Enable/disable tools pop-up menu.
getDisplayedImage()		x	Get displayed image.
getPassThroughValue()^{v}		x	Get the pass-through value used for the overlay buffers.
multiSelectKey()	x	x	Multi-select keyboard key.
panKey()	x	x	Mouse panning key.
waitForVerticalBlank()	x		Specifies vertical blanking sync.

(^{v}) = Virtual functions that must be overridden in a derived class.
(^{o}) = An override.

Table 38. ccWin32Display member functions

Creating Win32 Displays

To create a **ccWin32Display** object you pass it a **CWnd** handle in its constructor. For example,

```
m_pWndDisp = new CWnd;
m_pWndDisp->Create(
                NULL, "img", WS_CHILD|WS_VISIBLE, cr, this, 0);
m_pDisplay = new ccWin32Display(m_pWndDisp->m_hWnd);
```

`m_pDisplay` is then a **ccWin32Display** object you can include in your application and program to display images as well as graphics.

Win32 Display Attributes

The following lines of code add scroll bars and a pop-up tools menu to your **ccWin32Display** window.

```
m_pDisplay->showScrollBar(true, true);
m_pDisplay->toolsPopupMenuEnabled(true);
```

With the tools pop-up menu enabled, when you right-click in the window, a tools menu appears that allows you to select tools with your mouse. Figure 23 is a **ccWin32Display** example with scroll bars enabled, and the tools pop-up window active.

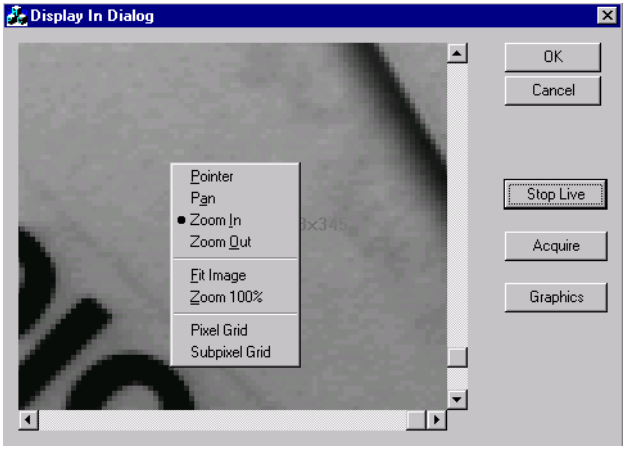


Figure 23. ccWin32Display example

Displaying an Image in a Win32 Window

To acquire an image and display it in the **ccWin32Display** window described above, you could use code like the following:

```
fifo_->start();
ccAcqImagePtrh img = fifo_->completeAcq();
m_pDisplay->image(img);
```

Releasing Win32 Displays

The following code deletes the **ccWin32Display** window.

```
delete m_pDisplay;
m_pDisplay = 0;
delete m_pWndDisp;
m_pWndDisp = 0;
```

Common Display Functionality

The following functionality is common to all **ccDisplay** windows.

Mouse Actions in ccWin32Display Console

ccDisplay objects handle mouse clicks as follows:

- Left button down: confine the mouse to the **ccDisplay** window. This ensures that the mouse is confined to the window while dragging.
- Left button up: release the mouse.

Display Creation and Destruction Restrictions

The following restriction governs the creation and destruction of objects derived from **ccDisplay**.

- You must not destroy **ccDisplay**-derived objects at static destruction time if those objects are currently used to display **ccPelBuffer**s created by **ccAcqFifo** objects.

There are two ways to avoid this:

- Do not construct **ccDisplay**-derived objects at static initialization time (STI) or as local **static** objects.
- Before your program terminates, call **ccDisplay::removeImage()**.

If you fail to observe this restriction, your program may hang or crash when it terminates.

Calling CVL Display API from Your Own DLL

If you create your own DLL that initializes CVL display objects and calls their functions, then you must call **cfInitializeDisplayResources()** in your application's initialization and setup code.

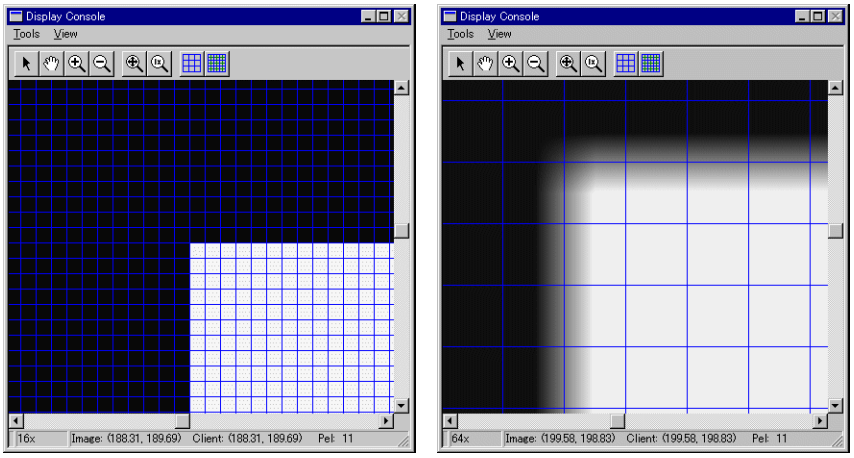
This is required because the resources for CVL display windows (the menus, buttons, strings, and so on) are embedded in the CVL DLL that contains the display functionality, *cogdisp.dll*. (CVL ships with eight versions of *cogdisp.dll*, covering the ANSI, Unicode, release, and debug versions for the two supported compilers, as described in *CVL Libraries for Visual C++ .NET 2012, 2013, and 2015* on page 38.) When your application calls a CVL display API in your DLL, Windows looks in your DLL for the necessary resources, but does not find them there. Calling **cfInitializeDisplayResources()** in your application informs Windows that the CVL display resources are to be found in *cogdisp.dll*, and not in your DLL.

Your application *must* call **cfInitializeDisplayResources()** if your application instantiates a **ccWin32Display** or **ccDisplayConsole** object in a DLL instead of in your application's executable.

In all other cases, the work performed by **cfInitializeDisplayResources()** is redundant and benign, but your application can call it without producing an error.

Interpolated Display

CVL supports display interpolation when displaying magnified images using any **ccUITablet**-derived class such as **ccDisplay**. Display interpolation smooths the otherwise blocky or jagged appearance of a magnified image, as shown in Figure 24.



Uninterpolated Interpolated

Figure 24. Display Interpolation

You control interpolation using the **ccUITablet::interpolation()** setter, passing one of the following **ccUITablet::InterpolationModes** as an argument:

- *ccUITablet::eNone*: do not interpolate on zoom (the default). This option produces a blocky display when zooming, but requires the least amount of time.
- *ccUITablet::eBilinear*: use bilinear interpolation to compute the value of each pixel. This option produces a somewhat blurry display, but requires only a modest amount of time. It reduces the image in size by 2 pixels along the top, bottom, and on each side.
- *ccUITablet::eHighPrecision*: use a high-precision interpolation method that considers additional pixel values. This option produces a smooth, precise image, but requires the most time. It reduces the image in size by 3 pixels along the top, bottom, and on each side.

For example, the following code sets the interpolation mode of a Win32 display to bilinear:

```
display->interpolation(ccUITablet::eBilinear);
```

Displaying Pel Buffers

To display a pel buffer in a display derived from **ccDisplay**, use the **image()** member function. For example, this is how you might create an Display Console object and display an image in it:

```
#include <ch_cog/windisp.h>
...
const ccStdVideoFormat &vf =
   ccStdVideoFormat::getFormat(cmT("Sony XC-ST50 640x480"));
ccDisplayConsole *display;
ccAcqFifoPtrh fifo;
ccAcquireInfo info;

// Create the display
display = new ccDisplayConsole(cmT("Display Console"),
   ccIPair(20,20), ccIPair(480,360));

// Acquire an image
fifo = vf.newAcqFifoEx(cc8100m::get(0));
fifo->start();

ccAcqImagePtrh img = fifo->completeAcq(
               ccAcqFifo::CompleteArgs().acquireInfo(&info));
// Image can be displayed directly
if (!info.failure())
   display->image(img);
```

As an alternative, you can convert the pel buffer, then display it. In this case, use the following example lines in place of the last two lines above.

```
ccPelBuffer<c_UInt8> pb = img->getGrey8PelBuffer();
// display it in the display console
if (!info.failure())
   display->image(pb, false);
```

The **image()** function copies the pel buffer into the image plane. To change the contents of the display console and display a new image, simply call **image()** again with the new pel buffer.

The image that you display in a display does not have to be acquired through a camera. You can display any **ccPelBuffer** including those you create synthetically or those returned by a function such as **ccBlobSceneDescription::makeImage()**.

Coordinate Systems

When **image()** displays a pel buffer, it applies any transformation object associated with it to create the pel buffer's client coordinate system. By default, the client coordinate system and the image coordinate system are identical.

The example program at the beginning of this chapter creates a transformation object that maps 40 image units (pels) to one client unit:

```
cc2Xform xform(cc2Vect(0,0), ccRadian(0), ccRadian(0),
  40.0, 40.0);
pb.imageFromClientXform(xform);
display->image(pb, false);
```

If you are using a ccAcqImage object, substitute the following for the last two lines above:

```
display->image(img, xform);
```

To learn more about client coordinates, see *Understanding Client Coordinates* on page 236.

Resizing and Positioning

Member functions of the **ccDisplay** class allow you to resize and reposition images. The **mag()** function scales the image by integral factors, while the **magExact()** function scales the image by any factor. For example,

```
display->mag(-2); // one-half actual size
display->magExact(1.52); //1.52x actual size
display->magExact(0.52); //0.52x actual size
```

The first line reduces the magnification to 1/2x. Negative integers passed to **mag()** zoom out, and positive integers passed to **mag()** zoom in. The last two lines change the magnification by exactly 1.52x and 0.52x, respectively. Numbers passed to **magExact()** greater than 1.00 zoom in, numbers less than 1.00 zoom out, and the number must be greater than 0.0.

If you use non-integer values with **magExact(),** the image must be sampled, which may slow performance. To restore performance after using **magExact()**, use **mag()**.

The **fit()** function centers and changes the magnification of the image to an integral value that best fits the current display window size. The **fitExact()** function centers and changes the magnification of the image to a value that best fits the current size while preserving the aspect ratio.

Retrieving the Displayed Image

The **ccDisplay::getDisplayedImage()** function retrieves the image and graphics displayed in a specific layer of the display. It returns the combined image and overlay layers by default. This function always uses display coordinates. It has three overloads for 8-bit, 16-bit, and 32-bit pel buffers. Choose the overload that matches the current desktop depth to avoid losses that may occur during data conversion.

Note Remember to size the pel buffer to the size of the client area you wish to retrieve.

Color Maps

The following sections provide detailed information on color maps, how they work at various desktop settings, and how to program them in CVL using the latest display member functions.

Color Mapping of Grey Scale Images

On 8-bit desktops, only one color map is in use. You have control over a limited range of this color map using the **ccDisplay::colorMap()** function.

The color map applies for both Windows host-based applications and non-Windows embedded applications.

Index	R	G	B	Color	Matching Stock Colors
0	0	0	0	Black	`ccColor::blackColor();`
1	128	0	0	Dark Red	`ccColor::dkRedColor();`
2	0	128	0	Dark Green	`ccColor::dkGreenColor();`
3	128	128	0	Dark Yellow	
4	0	0	128	Dark Blue	
5	128	0	128	Dark Magenta	
6	0	128	128	Dark Cyan	
7	192	192	192	Light Grey	`ccColor::ltGreyColor();`
8	192	220	192	Money Green*	
9	166	202	240	Sky Blue*	
10	10	10	10	Near Black*	
...					
...					
240	240	240	240	Near White*	
241					
242					
243	243	243	243	Pass Color**	`ccColor::passColor();`
244	160	20	200	Purple	`ccColor::purpleColor();`
245	250	125	50	Orange	`ccColor::orangeColor();`
246	255	251	240	Cream*	
247	160	160	164	Intermediate Grey*	`ccColor::greyColor();`
248	128	128	128	Medium Grey	`ccColor::dkGreyColor();`
249	255	0	0	Red	`ccColor::redColor();`
250	0	255	0	Green	`ccColor::greenColor();`
251	255	255	0	Yellow	`ccColor::yellowColor();`
252	0	0	255	Blue	`ccColor::blueColor();`
253	255	0	255	Magenta	`ccColor::magentaColor();`
254	0	255	255	Cyan	`ccColor::cyanColor();`
255	255	255	255	White	`ccColor::whiteColor();`

ccDisplay::colorMap() Sets This Range

Reserved Colors

User-Defined Colors

Cognex Reserved Colors

Reserved Colors

* Default colors that are subject to change.

** ccColor::passColor() is light grey in the image layer and pass-through in the overlay layer.

Figure 25. Color map (hardware palette) used for 8-bit desktops

On non-8-bit desktops, there are two color maps: one for overlay graphics (Figure 25) and one for images (Figure 26). The image color map contains only grey-scale values by default. You have full control over the entire 256-element range of the image color map using the extended color map function **ccDisplay::colorMapEx()**. The **ccDisplay::image()** function uses the image color map on non-8-bit desktops.

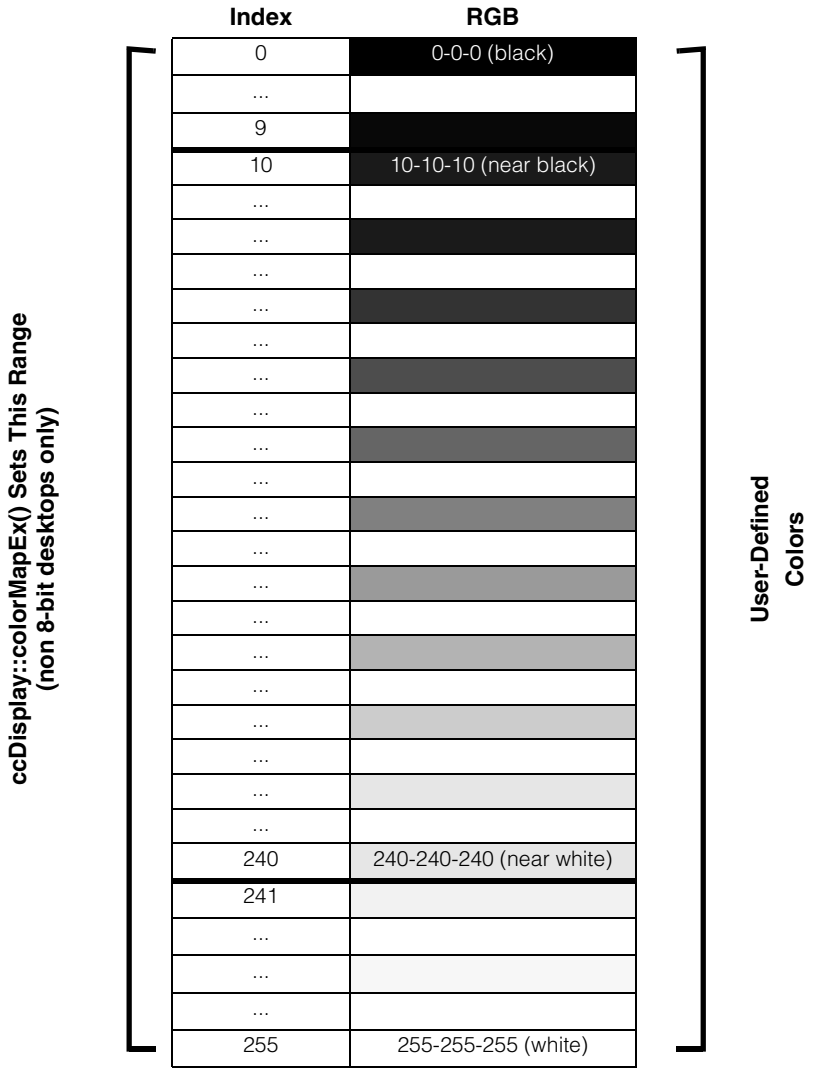


Figure 26. Image color map used for non-8-bit desktops

Color Map APIs

This section describes some of the APIs CVL provides for manipulating color maps.

User-Accessible Color Map Range

The **ccDisplay::ColorMapIndex** enumeration defines the minimum and maximum indices in the full color map (0 through 255) over which the user has control on 8-bit desktops. This enumeration has two values: *eMinUserColor* = 10 and *eMaxUserColor* = 240. The **ccDisplay::colorMap()** setter assumes that the 0th element in the input vector references the index into the color map specified by the second argument, which is always greater than or equal to 10. In other words, the **ccDisplay::colorMap()** getter does not return exactly what was supplied to the setter.

Setting Limited Color Map Range

The **ccDisplay::colorMap()** setter sets the color map within the range 10 through 240. On 8-bit desktops, the ranges 0 through 9 and 241 through 255 are reserved and cannot be changed: Microsoft Windows operating systems reserve the ranges 0 through 9 and 246 through 255, and Cognex reserves the range 241through 245. On 8-bit desktops, the color map table is also known as the *hardware palette* (see page 198).

The **ccDisplay::colorMap()** setter sets the user-accessible range of the color map for all desktop depths. This function does not alter any of the values in the reserved ranges for any desktop depth.

The **ccDisplay::colorMap()** getter returns the entire color map, including the reserved and user-accessible ranges.

Setting Full Color Map Range

The **ccDisplay::colorMapEx()** member functions allow you to get and set colors over the entire range 0 through 255 of the image color map for desktop depths greater than 8 bits (see page 199). This function is not supported for 8-bit desktops.

Cognex recommends that you use **ccDisplay::colorMapEx()** to set the color map on non-8-bit-desktops and **ccDisplay::colorMap()** to set the color map on 8-bit desktops.

Resetting the Color Map

The **ccDisplay::clearColorMap()** function resets the color map to the standard grey scale. This function takes no parameters and has a return type of *void*.

Color Format Conversion

Color mapping is the process of format conversion. For example, when you display an 8-bit image on a 16-bit desktop, the 8-bit colors must be converted to 16-bit colors. The conversion is done using the format conversion table or *color map*.

The format conversion table used on 8-bit desktops is built into the hardware and is also referred to as the *hardware palette*. It always has 256 values, the top 10 and bottom 15 of which are non-alterable (with the exception of indices 8, 9, 246, and 257, which can be altered by the operating system).

Non-8-bit desktops maintain two color maps, one for graphics and one for images. Both are used as software conversion tables. The software-based conversion table used for images is referred to as the *color map*. It too has 256 values, all of which can be set to any RGB value.

On an 8-bit desktop, each RGB value is packed into three bytes of storage space. On a 16-bit desktop, each RGB value is packed into one 16-bit word. On a 32-bit desktop, each RGB value is packed into one 32-bit double word.

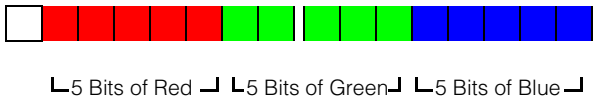
ccRGB::r() returns the red component of an 8-bit color:

ccRGB::g() returns the green component of an 8-bit color:

ccRGB::b() returns the blue component of an 8-bit color:

Figure 27. Internal representation of an 8-bit color

ccRGB::rgb15() returns a 16-bit color in 5-5-5 format (5 bits for each of the red, green, and blue components):

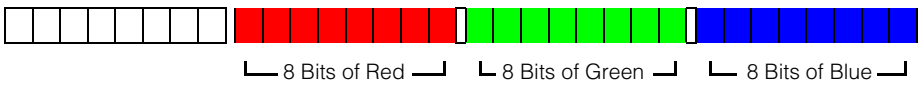


ccRGB::rgb16() returns a 16-bit color in 5-6-5 format (5 bits for each of the red and blue components, 6 bits for the green component):

Figure 28. Internal representation of a 16-bit color

Figure 28 (above) shows the internal representation of a 16-bit color and Figure 29 (below) shows the internal representation of a 32-bit color:

ccRGB::rgb() returns a 32-bit color with the red, green, blue components in the following order:



ccRGB::bgr() returns a 32-bit color with the red, green, blue components in the following order:

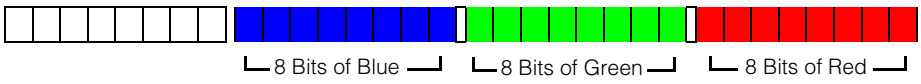


Figure 29. Internal representation of a 32-bit color

When converting between 8-bit and 32-bit colors, the mapping is straightforward. However, when converting between 16-bit and the other two formats, approximations must be made. Since the red, green, or blue component can only be represented with 5 or 6 bits on a 16-bit desktop, the system must approximate the 8-bit representation of that component.

Non-8-Bit Desktop Settings

When displaying an 8-bit image on a non-8-bit desktop using the **ccDisplay::image(ccPelBuffer<c_UInt8> pb, bool)** function, a conversion to the proper desktop depth automatically occurs using the image color map (see page 199). Note that calling **ccDisplay::colorMapEx()** has no negative effect on graphics rendering using colors.

Retrieving the Desktop Depth

To determine what desktop depth your display is set to while a CVL application is running, use **ccDisplay::displayFormat()**.

Understanding Display Quality

On 8-bit desktops, there are 256 distinct RGB values that can be seen. This means that on 8-bit desktops you can see 256 different grey values or 256 different colors. On newer video cards, the palette has 8 bits of precision for each RGB value. On older video cards, there are only 6 bits of precision.

When you display a pel buffer that has increasing pixel values from 0 to 255, the image looks smooth. Even when the image is magnified, it still looks smooth on a video card that has an 8-bit precision palette. However, it may look “blocky” on a video card that has only 6 bits of precision.

On 16-bit desktops, the data are stored in what is known as a 5-6-5 format: 5 bits of red, 6 bits of green, and 5 bits of blue. This means that there are only 32 different R values, 64 different G values, and 32 different B values. This becomes apparent when you zoom in on a 16-bit image, which may look “blocky.” Each block of four rows has the same value, because of bit packing and limited RGB values. This is not a bug.

You will also see a slight hint of green in a 16-bit image. This is because there is one more bit in the green value. The result is that a monochrome pel buffer may look better on some 8-bit desktops than on a 16-bit desktop. Both 24-bit and 32-bit pel buffers have 8 bits of precision in each of the R, B, and G values, so monochrome pel buffers display well at these depths.

The green image problem is specific to 16-bit desktops and to 8-bit desktops that only support 6-bit DACs.

Constructing Color Objects

The Cognex static predefined **ccColor** objects (see *%VISION_ROOT%\defs\ch_cvl\color.h*) are constructed using either index values or RGB values depending on the desktop depth.

- When you provide explicit RGB values, for example **ccColor(255,0,0)**, non-8-bit desktops use the actual RGB values given, but 8-bit desktops use the nearest color that is available in the palette.
- When you provide an index value, for example **ccColor(249)**, 8-bit desktops reference the color map, and non-8-bit desktops reference the *graphics* color map to retrieve the RGB values for that index.

- When you use the Cognex predefined **ccColor::<colorname>Color()** functions, 8-bit desktops use the index and non-8-bit desktops use the RGB value to construct the color. The **ccColor::isIndexColor()** and **ccColor::isRGBColor()** both return true for these colors.

Cognex recommends that you use one of the predefined color functions, for example **ccColor::redColor()**, rather than index values to construct color objects. This will make your CVL application code more portable when running on different desktops.

When do Color Maps Apply?

When color maps are applied and how color maps are applied depends on your frame grabber, the bit depth of the acquisition FIFO, the bit depth of your desktop, and in some cases the current display magnification factor. These conditions determine one of five possible paths images take from acquisition to display. Table 39 summarizes these conditions.

Frame grabber	8-bit desktop	16 & 32-bit desktops
MVS-8600 (all FIFOs)	Path D	Path C
MVS-8500 (8, 16, & 32-bit FIFOs)	Path A (8-bit FIFOs only)	Path C (for non-matching FIFO and desktop bit depths) Path E (for matching FIFO and desktop bit depths)

Table 39. Image path summary

Frame grabbers can acquire images at 8, 16, or 32-bit depths as well as display the images at any of these same levels. You acquire 8-bit images using a **ccGreyAcqFifo**, 16-bit images using a **ccRGB16AcqFifo**, and 32-bit images using a **ccRGB32AcqFifo**. In Table 39 we say we have a *matching* condition when the FIFO depth is the same as the desktop depth, and a *non-matching* condition when they are different. Figure 30 shows the matching and non-matching cases supported by CVL.

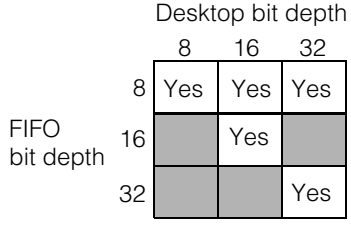


Figure 30. FIFO and desktop support summary

The five different paths images can take are described in the following sections.

Path A: Displaying 8-Bit Images on 8-Bit Desktops

The path that images acquired by 8-bit frame grabbers takes when displayed on an 8-bit desktop is shown below in Figure 31. This diagram applies only to certain frame grabbers, as listed in Table 39 on page 204.

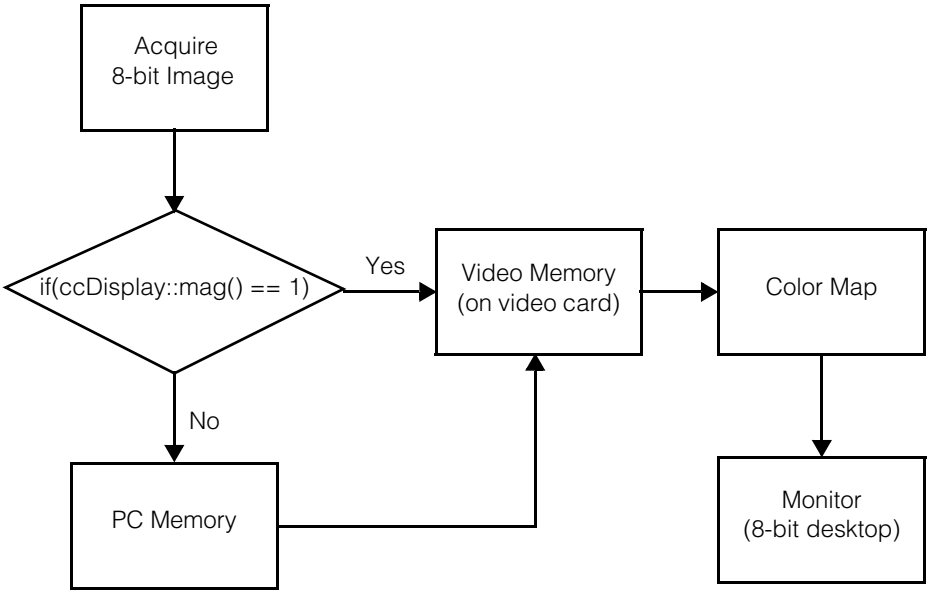


Figure 31. Flowchart of 8-bit image display on 8-bit desktops

For these conditions, the display magnification (zoom) setting determines whether or not the image is first copied to PC memory and thus affects the speed of the image display. The color map is always applied.

Path C: Displaying Images on 16 and 32-Bit Desktops

The path that the image takes when displayed on a 16 or 32-bit desktop is shown below in Figure 32. This diagram applies only to certain frame grabbers, as listed in Table 39 on page 204.

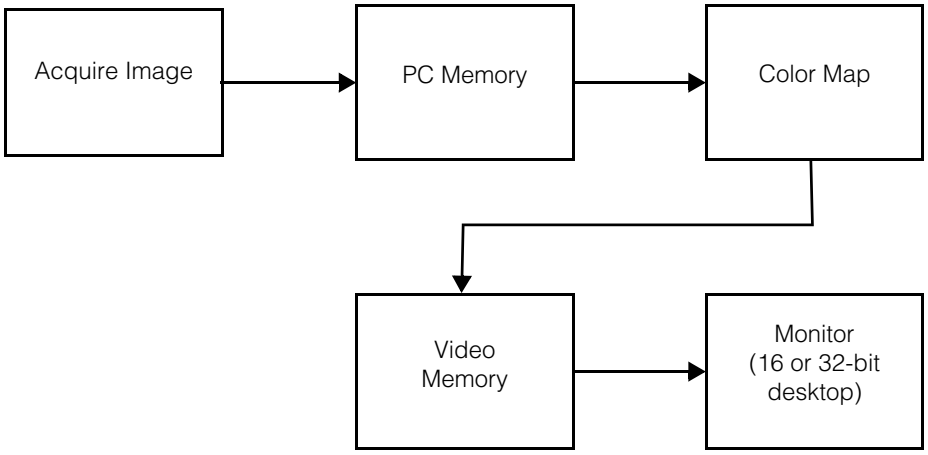


Figure 32. Flowchart of image display on 16 or 32-bit desktops

The **makeLocal()** argument to the **fifo->complete()** call is ignored. In other words, the image is always copied directly into PC memory and the color map always applies.

Path D: Displaying 8-Bit Images on 8-Bit Desktops

The path that the image acquired by an MVS-8600 frame grabber takes when displayed on an 8-bit desktop is shown below:

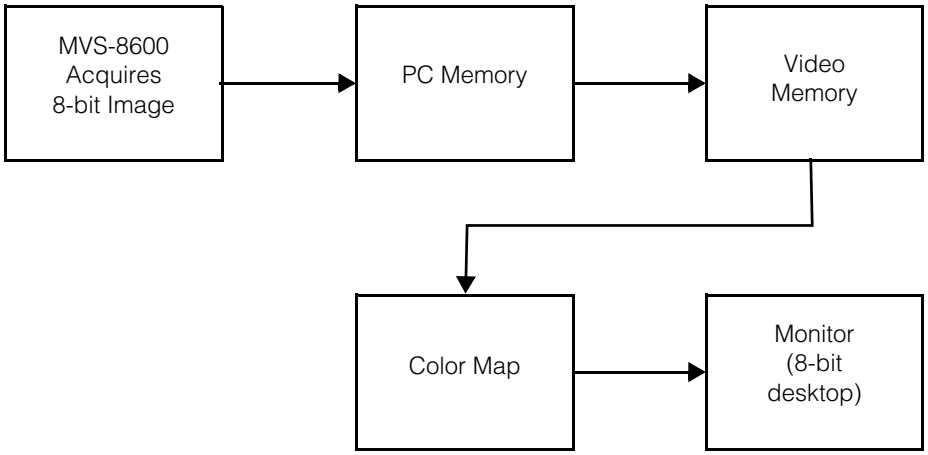


Figure 33. Flowchart of MVS-8600 8-bit image display on 8-bit desktops

The **makeLocal()** argument to the **fifo->complete()** call is ignored. In other words, the image is always copied directly into PC memory and the color map always applies.

Path E: Displaying Images on 16 and 32-Bit Desktops

The path that the image takes when displayed on a 16 or 32-bit desktop is shown below. This path applies only to certain frame grabbers, as listed in Table 39 on page 204.

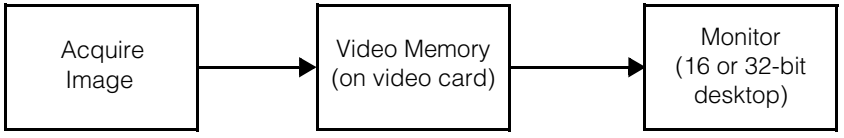


Figure 34. Flowchart of image display on 16 and 32-bit desktops

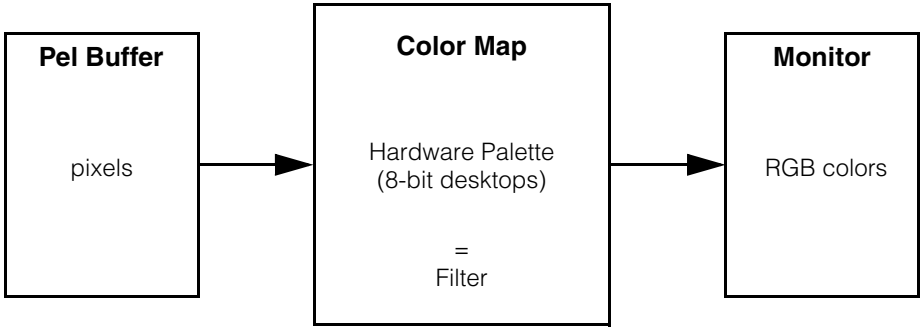
The color map never applies regardless of the magnification setting.

Issues with Color Maps

This section describes some issues you must be aware of when using color maps.

Windows Focus Affects Palette on 8-Bit Desktops

Pixel information from an acquired or live image is first stored to a buffer (the so-called *pel buffer*), then the pixel values are mapped to RGB colors using the color map. The second step is also known as *color conversion* or *filtering*. Finally, the filtered colors are displayed on the video monitor.



On 8-bit desktops, color conversion is performed in hardware. There is no software conversion for 8-bit desktops. A single hardware palette is used to map pixel values to colors for all Windows desktop applications.

When you switch the focus from a CVL application that uses one color scheme to another Windows application that uses a different one, such as Microsoft Word, the colors in the CVL display console can change in an unexpected manner because the pixel values are mapped according to the palette used by the application that has the focus, in this case Microsoft Word. When you switch the focus back to the CVL display console, the image will return to the expected colors, as set by **ccDisplay::colorMap(cmStd vector<ccRGB>& map)**.

Switching the focus on non-8-bit desktops does not have the same problems as those experienced on 8-bit desktops.

Clipping and Color Maps

The following function displays images in a CVL display console window with an option to clip colors:

ccDisplay::image(const ccPelBuffer_const<c_UInt8>&, bool displayRaw = true)

On 8-bit desktops, input buffer pixels in the range 10 through 240 are displayed as grey values by default and pixel values in the ranges 0 through 9 and 241 through 255 are always displayed as colors (see Figure 25 on page 198). On non-8-bit desktops, all pixel values in the input pel buffer are displayed as grey values by default (see Figure 26 on page 199).

When displaying images on non-8-bit desktops, the **ccDisplay::image()** function uses the *image* color map, which defaults to an all linear grey scale mapping (see page 199). You can draw graphics with colors because they reference the *graphics* color map (see page 198). The colors in the reserved ranges of the graphics color map do not change.

Pixel values in an image are mapped to the corresponding index values in the color map when the image is displayed. For example, a pixel value of 100 in an acquired image is mapped to the color (RGB value) at index 100 in the color map.

Setting the second argument to the **ccDisplay::image()** function, *displayRaw*, to false clips the pel buffer. In other words, all pixel values less than 10 are mapped to 10, which corresponds to an RGB value of 10-10-10 (or near black), and all pixel values greater than 240 are mapped to 240, which corresponds to an RGB value of 240-240-240 (or near white). To prevent colors from being displayed in a grey-scale image on an 8-bit desktop, set *displayRaw* to *false*. Note that the original pel buffer root data are never altered since the input pel buffer is declared `const`.

Setting *displayRaw* to true (the default) does not clip the pel buffer. The pel buffer will be displayed as is and no mapping will occur. Since this is the default, if you do not specify a second argument when calling **ccDisplay::image()**, colors may appear in an otherwise grey-scale image when displayed on 8-bit desktops.

The *displayRaw* parameter is applicable to all desktop depths.

If the physical display device has a different pixel format than 8-bit, the input image will be automatically converted as appropriate, with a corresponding reduction in speed.

Synthetic Images in Dual Consoles on 8-Bit Desktops

On 8-bit desktops, applying the color map to synthetic images in different display consoles can result in seemingly odd behavior in the following scenario:

1. Create display console 1 first.
2. Then create display console 2.
3. Draw a synthetic image with the color map applied on display console 1.

When you call **ccDisplay::image()** with *displayRaw* set to true, the mapped colors are not displayed immediately. You must change the focus to console 1 to see mapped colors in the range 10 through 240. If you change the focus to console 2, mapped colors in the range 10 through 240 disappear and you will see only grey scale colors, unless you change the focus back to console 1.

When you call **ccDisplay::image()** with *displayRaw* set to false, the mapped colors are displayed immediately. But if you change the focus to console 2, the synthetic image on console1 will show black in the range 0 through 9, white in the range 241 through 255, and mapped colors in the range 10 through 240. Again, if you change the focus back to console 1, values less than 10 are clipped to 10 and values greater than 240 are clipped to 240.

Indexed Colors Can Change on Booting into Windows

Some indexed color values can change when you boot into Windows on 8-bit desktops only. Specifically, the color at index 247 is affected by your current desktop appearance and defaults Windows style. You can change your desktop decorations and Windows style by right clicking on your desktop, selecting the Properties menu item, selecting the Appearance tab, and then changing the Scheme pull-down menu. Cognex recommends that you set your desktop scheme to **Windows Standard**.

In the following color samples displayed on 32-bit, 16-bit, and 8-bit desktops, note the discrepancies between the colors at index 247:

Index	32-Bit Desktop	16-Bit Desktop	8-Bit Desktop
246			
247			
248			
249			
250			
251			
252			
253			
254			
255			

Figure 35. Indexed colors displayed on 32-bit, 16-bit, and 8-bit Windows desktops

The reserved colors at indices 8, 9, 246, and 247 of the color map are default colors that are subject to change by the operating system and the hardware (see Figure 25 on page 198). They depend in part on the color scheme selected in the Appearance tab of the Display Properties dialog in Windows. Since these are reserved indices, there is no way to set them through either CVL or the Microsoft Windows API (see *Clipping and Color Maps* on page 208).

When a user-defined **ccColor** object is supplied, the nearest system color is selected. In the case of **ccColor::greyColor()**, or color map index 247, the system can select dark grey or light blue instead of grey, depending on how the system colors are set.

Of the four colors that exhibit this behavior, only **ccColor::greyColor()** is used by CVL. The other three indices—MoneyGreen (index 8), SkyBlue (index 9), and Cream (index 246)—do not have associated **ccColor** objects. Therefore, this problem only applies to **ccColor::greyColor()** objects.

Displaying Color Images

All displays derived from **ccDisplay** can display 8-bit grey-scale images. If you are using a display that is an instance of **ccDisplayConsole** or **ccWin32Display**, you can display images that were acquired in color using one of the Cognex frame grabbers that support color display, as described in your CVL version's *Getting Started* manual.

Note Keep in mind that CVL image processing functions and vision tools do not operate on color images: they only work on grey-scale images. Color display is useful only for its visual effect when displayed.

To learn more about acquiring color images, see *Acquiring with Color Cameras* on page 127.

The following example shows how to display color images.

```
// Construct a fifo as described in "Acquiring with Color Cameras",
// in this example we assume an 8504 with a Sony DXC-390
ccFrameGrabber& fg = cc8504::get(0);
const ccStdVideoFormat& format = ccStdVideoFormat::getFormat
       (cmT("Sony DXC-390 640x480 IntDrv (3 Plane Color) CCF"));
ccAcqFifoPtrh fifo = format.newAcqFifoEx(fg);

// Create a display console to display the image
ccDisplayConsole console(ccIPair(640,480));

// Acquire and display images in a loop to get pseudo live display.
for (int loopCnt=0; loopCnt < 100; ++loopCnt)
{
  // Acquire an image for the display
  fifo->start();
  ccAcqImagePtrh img = fifo->completeAcq();

  // If we acquired a valid image
  if (img && img->isBound())
  {
    if (!console.image(img));
        // Image could not be displayed, probably because it
        // could not be converted. Try changing the Windows
        // desktop color depth setting.
        throw something;
```

```
  }
  else
  {
    // ... handle acquisition failure
  }
}
```

Displaying Live Images

For most applications, static grey-scale images contain all of the information necessary for the vision tools to perform their tasks. Human operators, however, sometimes have trouble working with still images and may prefer working with a live image. In such cases, you may wish to display live images for a human operator's convenience while acquiring single images for processing.

CVL also provides an interface that allows you to control some properties of live display.

Creating a Live Display Window

You can use any class derived from **ccDisplay** to show live images. You create a display window (or display console) for display of live images in the same way you would create one for display of single images.

For example, to create a display console for display of live images, you would instantiate a **ccDisplayConsole**, passing a pair of values specifying the dimensions of the window:

```
ccDisplayConsole display(ccIPair(640, 480));
```

You can perform live display of 8-bit grey scale images on a Windows desktop of any color depth. For best performance with most Cognex frame grabbers, however, the pixel depth of the acquisition FIFO must match the color depth of the Windows desktop. See *Displaying Live Images on Non-8-Bit Desktops* on page 217 for an example on this.

Live Display Properties

To allow you to change live display properties from the defaults, CVL provides the **ccLiveDisplayProps** class. You can set and read live display properties using the following **ccLiveDisplayProps** member functions:

- **clientTransform()** - gets/sets the image-to-client coordinate transform applied to each pel buffer acquired for live display. If this parameter is not explicitly set, live display uses the transform assigned to the pel buffer during acquisition by default.
- **frameRateInterval()** - gets/sets the sampling interval used to compute the live display frame rate. This interval is the amount of time (in seconds) the display system uses as the basis for calculating live display frame rate. A value of zero (the default) tells the display system to use the number of frames displayed since **ccDisplay::startLiveDisplay()** was called to calculate the frame rate. (See also *Setting Live Display Properties* on page 216.)
- **displayOutput()** - gets/sets whether all acquired images are displayed (the default) or not displayed.

- **restartDelay()** - gets/sets the delay between the end of one acquisition and the start of the next. A value of zero (the default) allows live display to run at the maximum frame rate possible. Increasing this value reduces the frame rate.
- **threadPriority()** - gets/sets the priority of the live display thread.
- **pelbufferCallback()** - gets/sets the callback function that is called each time a pel buffer is acquired during live display. See *Using a Live Display Callback* on page 216 for sample code demonstrating how to use a live display callback.

Starting Live Display

Derived classes of **ccDisplay** that support live display do so with the **startLiveDisplay()** member function. This function acquires and displays images from an acquisition FIFO continuously, and always as efficiently as possible.

Always call **fifo->prepare(0.0)** to initialize the acquisition system before calling **ccDisplay::startLiveDisplay()**. If **prepare()** succeeds, it is safe to start the live display. If it fails, this may mean that a camera is not connected or the acquisition system could not be set up.

Always check the return value of **ccDisplay::startLiveDisplay()**. If images acquired through the specified FIFO cannot be displayed (for example, because the pixel depth of the FIFO does not match the desktop depth of the display), **startLiveDisplay()** returns false. The example programs in this chapter generally assume that everything works correctly. Your programs should not make the same assumptions.

Since **ccDisplay::startLiveDisplay()** needs constant access to the acquisition FIFO, do not make any changes to the FIFO while live display is in progress. Set up all FIFO properties and live display properties before calling **startLiveDisplay()**. If you need to change any aspect of either the FIFO or the live display, call **stopLiveDisplay()** first. You cannot change live display properties while live display is running.

Live Display Code Examples

The following sections show some different scenarios for setting up and using live display.

Setting Live Display Properties

The following code segment shows an example of how to set live display properties:

```
const ccStdVideoFormat* format;
format = &ccStdVideoFormat::getFormat(
  "Sony XC-ST50 640x480 IntDrv CCF");
ccAcqFifoPtrh fifo = videoFormat->newAcqFifoEx(*fg);
ccDisplayConsole display(ccIPair(640, 480));
ccLiveDisplayProps props;
props.frameRateInterval(5.0);
display.startLiveDisplay(fifo, &props);
```

This code sets the frame rate interval to 5 seconds by calling the **ccLiveDisplayProps::frameRateInterval()** setter with an argument of 5.0. This is the amount of time the display system uses as the basis for calculating the live display frame rate. For example, with the frame rate interval set to 5 seconds, the display system counts the number of frames actually displayed in 5 seconds and then uses this number to calculate the live display frame rate.

You can query the live display frame rate by calling **ccDisplay::liveFrameRate()**. By default, if you do not explicitly set **ccLiveDisplayProps::frameRateInterval()**, **ccDisplay::liveFrameRate()** uses the number of frames displayed since **ccDisplay::startLiveDisplay()** was called as the basis for calculating the frame rate.

Once the frame rate interval property has been set, pass the *props* object by pointer as an argument to **ccDisplay::startLiveDisplay()**. The specified frame rate interval remains in effect until **ccDisplay::stopLiveDisplay()** is called.

Using a Live Display Callback

You can specify a callback to be called each time an image is acquired during live display. The callback function can signal the thread that is processing images that a new image is available for processing.

The callback is a class or structure that contains an overridden **operator()** method. The argument to the overridden **operator()** method is a **const void** pointer in which the acquired pel buffer can be passed to the callback function. You must allocate the callback object on the heap.

You set the callback for live display by calling the **pelbufferCallback()** setter of **ccLiveDisplayProps**.

The following code demonstrates how to set up and use a pel buffer callback function during live display:

```
// Declare the callback struct (can also be a class)
struct MyPelbufCallback : ccCallback1<const void*>
{
  void operator()(const void* ptr)
  {
    const ccPelBuffer<c_UInt8>& pelbuffer = *(
      reinterpret_cast<const ccPelBuffer<c_UInt8>*>(ptr));
    // Signal the processing thread that an image is available
    ...
  }
};
...
ccLiveDisplayProps props;
ccCallback1Ptrh callback = new MyPelbufCallback;
props.pelbufferCallback(callback);
display.startLiveDisplay (fifo, &props);
```

Live Display Recommendations

This section contains recommendations regarding the use of live display in CVL.

Displaying Live Images on Non-8-Bit Desktops

For all frame grabbers, the acquisition FIFO should match the color depth of the Windows desktop to achieve the fastest possible live display frame rate. This applies even when acquiring grey-scale images or using frame grabbers other than an MVS-8100C.

For example, when using an MVS-8100M frame grabber, a Sony XC-ST50 monochrome camera, and a 16-bit Windows desktop, the following code provides the fastest live video frame rates:

```
#include <ch_cvl/vp8100.h>
#include <ch_cvl/acq.h>
#include <ch_cvl/vidfmt.h>
#include <ch_cvl/windisp.h>
#include <windows.h>
#include <ch_cvl/xform.h>

int cfSampleMain(int, char** const)
{
  cc8100m& fg = cc8100m::get(0);
  const ccStdVideoFormat &greyCam =
       ccStdVideoFormat::getFormat(cmT("Sony XC-ST50 640x480"));
  ccStdRGB16AcqFifoPtrh liveFifo =
```

```
    greyCam.newAcqFifo(fg, ccStdVideoFormat::ckRGB16);

  ccDisplayConsole *display;

  display = new ccDisplayConsole(cmT("Display Console"),
    ccIPair(20,20), ccIPair(480,360));

  // Always call ccAcqFifo::prepare(0.0) to initialize the
  // acquisition system before starting live display.
  if(liveFifo->prepare(0.0))
  {
    // As ccLiveDisplayProps is a default argument, this code does
    // not pass any live display properties to startLiveDisplay()
    display->startLiveDisplay(liveFifo);

    // some processing would go here...

    display->stopLiveDisplay();
  }
  else
  {
    // notify the user that prepare() failed
  }

  liveFifo = ccStdRGB16AcqFifoPtrh(0);
  delete display;
  return 0;
}
```

Changing the Desktop Color Depth

Do not change the desktop color depth during live display, or even while CVL is running as this may cause the system to hang.

To safely change the desktop color depth, Cognex recommends performing these steps in the following order:

1. If a CVL-based application is running, terminate it.
2. Change the desktop color depth.
3. Reboot the PC.
4. Run the CVL-based application.

Measuring CPU Usage During Live Display

When measuring CPU usage during live display with Cognex frame grabbers, the accuracy of the CPU Usage meter within the Windows Task Manager is dependent on the timer resolution setting on your PC. The default timer resolution for Windows operating systems is set at a Microsoft-defined high value, which can cause the CPU usage measurement to underestimate the actual CPU load in some cases. Using the default setting, measurements of CPU usage during live display often report CPU loads of less than 5% when, in fact, the CPU load may be higher (depending on your desktop resolution and color depth). To more accurately control the timer resolution on your PC, you can use the Microsoft Win32 functions **timeBeginPeriod()** and **timeEndPeriod()**. These functions change the timer resolution on your PC, thereby improving the accuracy of the CPU Usage meter within the Windows Task Manager. To achieve the most accurate reading in the CPU Usage meter, set the timer resolution to the minimum resolution allowed on your PC. To determine the minimum resolution, use the Win32 function **timeGetDevCaps()**.

Caution *Changing the timer resolution on your PC by using the Win32 functions* **timeBeginPeriod()** *or* **timeEndPeriod()** *may affect other parts of the system. Other Win32 functions such as* ***sleep()*** *are also affected by the timer resolution setting. Changing the timer resolution on your PC may change the behavior of your CVL application and of all other running processes.*

Customizing Image Display Environments

Image display classes are designed to allow you to easily add functionality to your applications. For example, if you find you would like to take some specific action when a user right-clicks the mouse while the pointer in on an image, you can write your own code to accomplish this by doing the following:

1. Derive your own display class. For example, derive **my_ccDisplay** from **ccDisplay**.
2. In **my_ccDisplay** override the protected function **mouseRight_()** with your own routine that takes the specific action you desire. At the beginning of your override routine call **ccDisplay::mouseRight_()** to execute the code there before executing your new code.
3. In your application use the new **my_ccDisplay** class instead of **ccDisplay**.

ccDisplay contains a set of protected virtual functions you can override to add your own functionality to image displays. These functions are discussed in the next section.

Customizing With Overrides

The **ccDisplay** class provides a set of protected virtual functions that you can override to add custom functionality to your applications. Some functions are place holders and simply return without executing any code unless overridden. Other functions contain code that you must execute in your override.

Table 40 summarizes these protected virtual functions and includes a short description of each one.

Protected virtual function	Place holder?	Description
panChanged()	Yes, no code.	Called by **ccDisplay::pan()** for notification that a new pan has been applied to this image.
magChanged()	Yes, no code.	Called by **ccDisplay::mag()** for notification that a new magnification has been applied to this image.
imageChanged()	Yes, no code.	Called by **ccDisplay::image()** for notification that a new image has been displayed.

Table 40. Protected virtual functions summary

Protected virtual function	Place holder?	Description
mouseModeChanged()	Yes, no code.	Called by **ccDisplay::moueMode()** for notification that a new mouse mode has been applied.
colorMapChanged()	Yes, no code.	Called by **ccDisplay::colorMap()** for notification that a new color map has been applied.
imageMapChanged()	Yes, no code.	Called by **ccDisplay::colorMapEx()** for notification that a new color map has been applied.
redraw_()	No, contains implementation.	Draws the image.
dragStart_()	No, contains implementation.	Called at the start of an image drag or pan.
dragAnimate_()	No, contains implementation.	Called repeatedly during an image drag or pan to implement visual image dragging on the screen.
dragStop_()	No, contains implementation.	Called at the end of an image drag or pan.
resize_()	No, contains implementation.	Called when the parent scope size changes.
click_()	No, contains implementation.	Implements mouse mode behavior when the user clicks the mouse on an image.
dblClick_()	No, contains implementation.	Called when a double click occurs while the mouse pointer is in the image.
keyboard_()	No, contains implementation.	Implements a keyboard event when the mouse is on the image.
mouseEnter_()	No, contains implementation.	Called when the mouse pointer enters the image with the left mouse button depressed.

Table 40. Protected virtual functions summary

Protected virtual function	Place holder?	Description
idleMouseEnter_()	No, contains implementation.	Called when the mouse pointer enters the image with no mouse button depressed.
mouseLeave_()	No, contains implementation.	Called when the mouse pointer leaves the image.
mouseRight_()	No, contains implementation.	Called when the right-hand mouse button is depressed while the mouse pointer is in the image. Implement s right-click for *eZoomIn* and *eZoomOut* modes.
select()	No, contains implementation.	Called when the image state changes to *selectedState* (the image becomes selected). If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the image object itself, *parentChanged* = false. Sets the root's color map to this display's color map.
updateDisplay()	No, contains implementation.	Called to update the display. Calls **ccUITablet::update()** on a specified layer.
updatePanRanges()	No, contains implementation.	Called when the minimum or maximum pan ranges change. This can result from calls to the**ccDisplay::minPan()** and **ccDisplay::maxPan()** setters, or a call to **ccDisplay::image()**.

Table 40. Protected virtual functions summary

Images and Coordinates 5

This document describes how CVL represents images and the coordinate systems used in the Cognex Vision Library

Some Useful Definitions defines certain terms you will encounter as you read.

Images describes how CVL represents pixel images.

Coordinate Systems describes the image coordinates and client coordinates, the two coordinate systems that CVL uses to specify locations in images.

To learn more about the mathematical foundations of CVL images and coordinates, see *Math Foundations of Transformations* on page 404.

Some Useful Definitions

The following terms may be useful in reading this chapter.

arbitrary calibration Transforming image coordinates (pixels) into your own native coordinate system, such as millimeters.

client coordinates A user-defined, real-valued coordinate system used to specify locations in a window. A transformation object maps between image coordinates and client coordinates.

image coordinates A pixel-based coordinate system relative to the upper left corner of a window in which all coordinates are offset by an arbitrary value (called the offset). If the offset is (0,0), image coordinates are identical to window coordinates.

intrinsic calibration Using the transformation object to correct for the inherent distortion in all vision systems.

root image coordinates A pixel-based coordinate system in which the upper left corner of a root image is always the point (0,0).

root image A two-dimensional array of values called pixels. The value stored in each pixel of the root image indicates the light intensity or brightness of each pixel. Typically, pixel values are integers, though other values may be used as well.

transformation object A data structure consisting of a 2x2 matrix and a two-element vector used to map between coordinate systems.

window A rectangular region of a root image. Any number of windows may share the same root image.

window coordinates A pixel-based coordinate system which the upper left corner of a window is always the point (0,0).

Images

An image is a two-dimensional array of values. Each element in the root image array is called a *pixel* or a *pel* (short for picture element). The value stored in each pixel of the image indicates the light intensity or brightness of each pixel. Typically, pixel values are integers, though other values may be used as well.

In a typical application, a pixel is detected by the photoreceptors of a video camera and digitized by an image digitizer (a frame grabber). As image pixels are acquired, they are stored in an image. See *Overview of Image Acquisition* on page 63 for more information about acquiring images.

In addition to acquiring an image with a video camera, you can also generate images mathematically (synthetic images) or you can use images stored in an image database. You can use these images to test your vision application when you do not have access to a frame grabber.

Pixels and Coordinate Grids

This chapter discusses how images are represented in CVL and how you can refer to locations in those images. It is important to understand the difference between pixel locations and coordinates.

A coordinate system is a mathematical concept. The grid lines that make up a coordinate system are infinitely thin, and the points at the intersections of the grid lines are infinitely small. A pixel, on the other hand, not only occupies memory, it also occupies physical space.

CVL uses a left-handed coordinate system by default. In this coordinate system, the origin (0,0) is at the upper left corner. X values increase to the right, and Y values increase down.

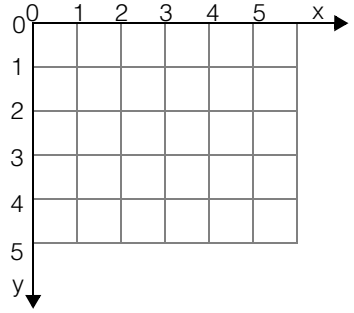


Figure 36. A left-handed coordinate system

Angles in CVL increase from the x-axis to the y-axis. In the default left-handed coordinate system, angles increase clockwise and decrease counter clockwise as shown below:

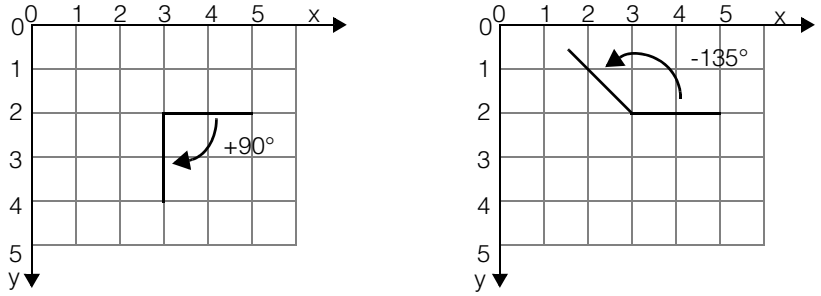
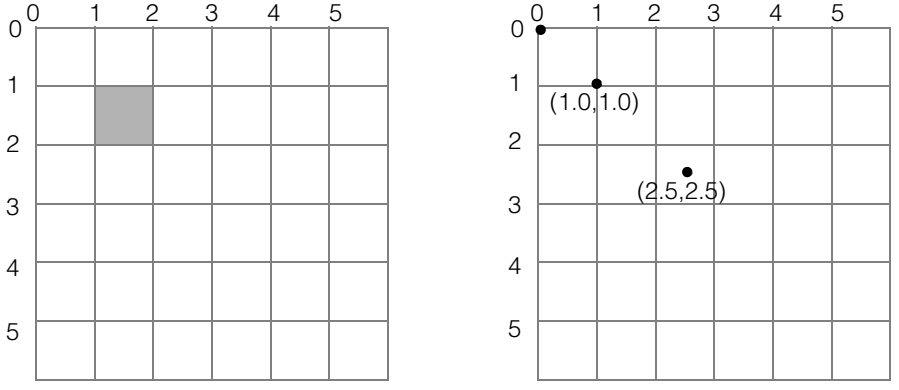


Figure 37. Angles in CVL

In CVL, locations are always given as points on the coordinate grid. The pixels occupy the space between the grid lines. For convenience, a pixel is commonly referred to by the integer coordinate of its upper left corner. Thus, the notation (1,1) can refer to a point or to a pixel, depending on the context. Figure 38 shows the difference between pixels and points.



The pixel “at” (1,1)

Points on the coordinate grid

Figure 38. Pixels and points

When you specify locations in images for the vision tools, you usually use the class **cc2Vect** which describes a two-element vector. The **cc2Vect** class provides several member functions that let you set and access the elements of the vector as x and y locations (see also *Points and Vectors* on page 409).

Root Images

In CVL, a *root image* is a block of memory used to store an image. They are called root images because, as described in the next section, it is possible (and often desirable) to work with only a region of an image. The term *root* helps you distinguish between the entire image and the subsection you are working with.

The **ccPelRoot** class is a template class that CVL uses to represent root images with pixel values. CVL provides instantiations of **ccPelRoot** for the types **c_UInt8** for 8-bit images, **c_UInt16** for 16-bit images, or **c_UInt32** for 32-bit images.

The **ccPelRoot** class maintains information about the root image, including its height and width in pixels, an alignment modulus value used for efficient byte alignment of the memory used for the image, and a reference count of the number of windows bound to this image. Figure 39 shows a typical **ccPelRoot** object.

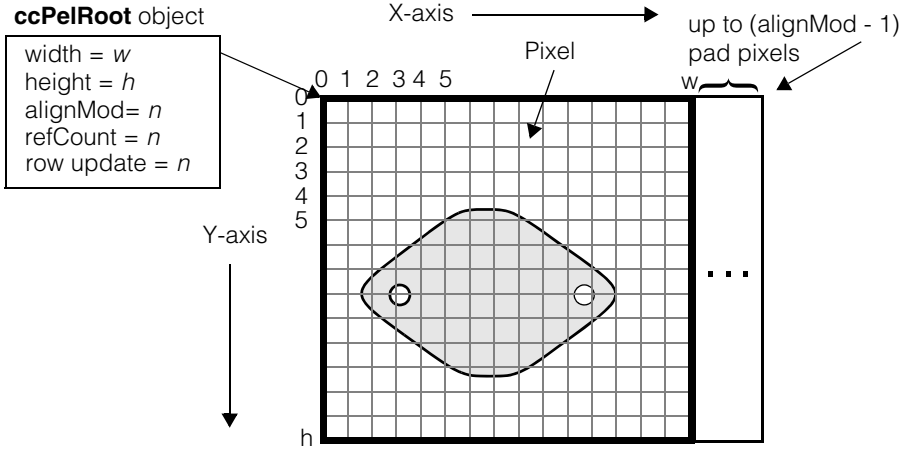


Figure 39. A ccPelRoot object is used to represent a root image

In Figure 39, the width of the root image is *w* and its height is *h*. Coordinates in a root image are always measured from the upper left corner of the image; that point is always (0, 0). In practice however, you will seldom use root image coordinates. *Coordinate Systems* on page 231 discusses the coordinate systems used with images.

Note that in Figure 39 the root image contains *pad pixels* at the end of each row. These pad pixels ensure that the first pixel in each row is aligned on a particular byte boundary to comply with addressing requirements on the host computer or to optimize image acquisition or image processing. The alignment-modulus component of the root image specifies how the memory that makes up the pixels is aligned. If the alignment-modulus is 1, the pixels are aligned on byte boundaries, and there are no pad pixels. If the alignment-modulus is 32, the pixels are aligned on 4-byte, or word boundaries. The maximum number of pad pixels is always one less than the alignment-modulus value.

All CVL images are regular. Every row has the same number of bytes, and the distance in bytes from the beginning of one row to the beginning of the next is constant through the entire image.

If you have a pointer to a pixel, you can use the *row update* value to get a pointer to the pixel immediately above it or immediately below it. Add the update row value to get the next higher y value; subtract it to get the next lower y value. The *row update* value may be different from the width of the root image because it takes into account any pad pixels.

Windows (Pel Buffers)

A window, or a pel buffer, is a rectangular region of a root image. Any number of windows may share the same root image. A window that refers to a root image is said to be *bound* to the root image. Since windows are independent data structures, it is possible to create windows with a defined width and height but without a reference to a root image. A window that does not refer to a root image is said to be *unbound*.

While pixels are acquired and stored in root images, all access to the pixels of the image is through windows. The Cognex Vision Library uses the classes **ccPelBuffer** and **ccPelBuffer_const** to represent windows. **ccPelBuffer_const** provides read-only access to the pixels of the window's root image, and the member functions of **ccPelBuffer** allow you to modify the contents of the root image.

Note In CVL, the terms *window* and *pel buffer* are interchangeable. The term window best captures the idea that a **ccPelBuffer** contains no bits but merely describes a region within a **ccPelRoot**. The term pel buffer reinforces the idea that the only way to access the pixels in an image is through a **ccPelBuffer**.

A **ccPelBuffer** object maintains information about the window's size and location within a root image. **ccPelBuffer** objects also maintain a transformation object that is used to map pixel coordinates to a user-defined coordinate system. See *Coordinate Systems* on page 231 to learn more about coordinate systems. Figure 40 shows a typical **ccPelBuffer** object.

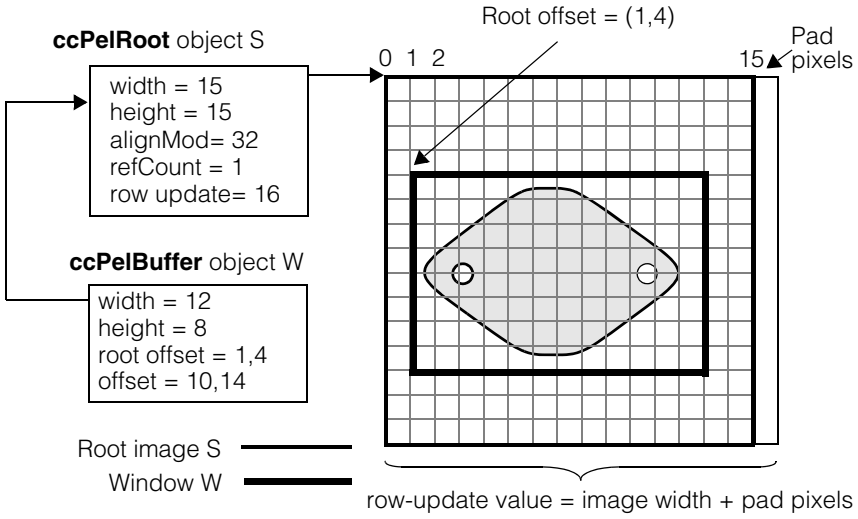


Figure 40. A ccPelBuffer object is a window into a root image

The window, *W*, contains a reference to the root image *S*. When *W* is bound to *S*, the reference count of the root image is incremented. If you were to create another window bound to *S*, the root image's reference count would increase to 2. Although **ccPelBuffer** objects conceptually contain image pixels, they really refer to a root image that contains the actual pixels.

The location of the window in the root image is called the *root offset*. It is always described as a point relative to the upper left corner of the root image. A window can encompass the entire root image that it is bound to. In Figure 40, the window is smaller than the root image. To specify the location of a window (a **ccPelBuffer** or a **ccPelBuffer_const**) within a root image, you use the member function **windowRoot()**.

Windows also maintain a value called the *offset* which is used to provide the window with a local coordinate system called image coordinates. The offset is described in *Working with Image Coordinates* on page 232.

Note It is easy to confuse the window's root offset with its offset. The root offset always denotes the vector from the upper left corner of the root image to the upper left corner of the window. The offset is a value used to provide windows with a reference coordinate system.

Coordinate Systems

When you work with CVL images, you will be concerned primarily with two coordinate systems: image coordinates and client coordinates. Image coordinates are aligned with pixels with no scaling or rotation. Client coordinates provide a real-valued coordinate system that you can use for calibration and for translation of pixel units into native units.

When you initially acquire an image, image and client coordinates have their (0,0) aligned with the upper left corner of the root image which is always the point (0,0) in root image coordinates.

Note This is not true if you specify a region of interest with the **ccRoiProp** property. Using this property has the same effect as changing the size of the window. (See *Changing the Size of a Window* on page 233 and the description of **ccRoiProp** in the *CVL Class Reference*)

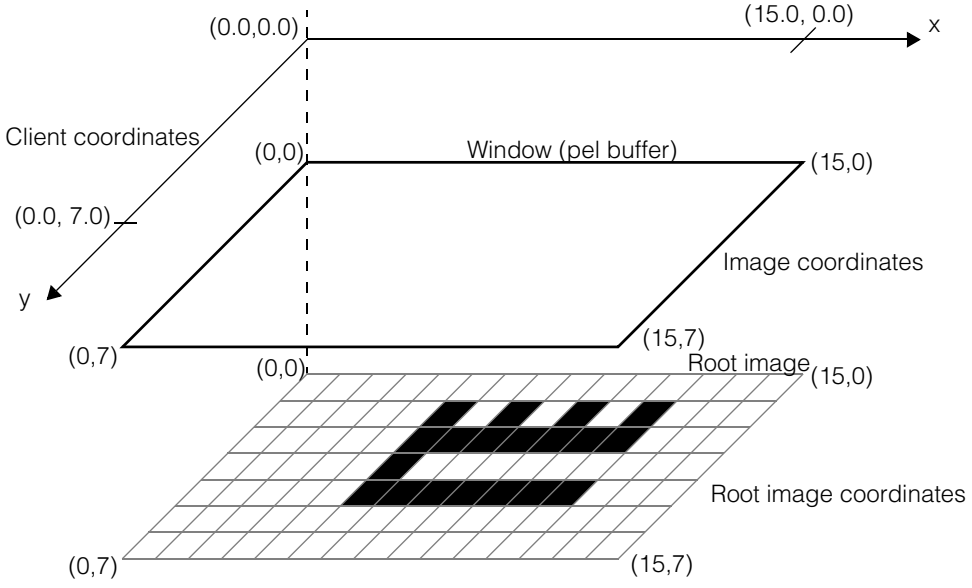


Figure 41. Initial state of coordinate systems after image acquisition

Figure 41 shows the initial state after acquiring a 15x7 image. (CVL supports no video format that would return an image this size. It is used here only for illustration.) For the rest of this chapter, assume that this image was acquired into a variable called *thePelBuf* of type **ccPelBuffer<c_UInt8>**.

Working with Image Coordinates

Image coordinates are measured relative to the window's *offset*. The offset is an arbitrary point that measures the distance from the origin (0,0) of the image coordinate system to the upper left corner of the window. Another way to think about the offset is that it labels the coordinates of the upper left corner of the window. As Figure 41 illustrates, the offset of a newly acquired image is (0, 0) and it coincides with the upper left corner of the root image. In other words, the root offset is also (0, 0).

Note Although the root offset describes the location of the window relative to the upper left corner of the root image, this value is not particularly useful when working with images.

The rest of this section uses the following function to illustrate what happens to the image coordinate system as you work with windows. Assume, also, that the image in Figure 41 is a binary image. White pixels have high values and black pixels have low values.

```
void showWindowInfo()
{
  ccIPair theOffset = thePelBuf.offset();
  ccIPair rootOffset = thePelBuf.offsetRoot();

  std::cout << "offset: (" << theOffset.x() << ", ";
  std::cout << theOffset.y() << ")" << std::endl;

  std::cout << "root offset: (" << rootOffset.x() << ", ";
  std::cout << rootOffset.y() << ")" << std::endl;

  std::cout << "width: " << thePelBuf.width();
  std::cout << " height: " << thePelBuf.height() << std::endl;
}
```

If you were to call **showWindowInfo()** immediately after acquiring the image in Figure 41, you would see the following results:

```
offset: (0, 0)
root offset: (0, 0)
width: 15 height: 7
```

The black pixel in the upper left corner of the shape is at location (5,1) in image coordinates. So the following statement would be true.

```
thePelBuf.get(5,1) == 0; // True. The pixel at (5,1) is black.
```

Changing the Size of a Window

Since the vision tools and the image processing tools usually work faster with smaller images, you will probably want to change the size of acquired image's window. The usual way to do this is with **ccPelBuffer::window()**.

To make the window from Figure 41 small enough to hold the feature of interest with a one pixel border, call **window()** like this:

```
thePelBuf.window(4, 0, 9, 6);
```

This function places the upper left corner of the window at (4, 0) in image coordinates and makes the window 9 pixels wide by 6 pixels high. Figure 42 shows the effect of calling this function.

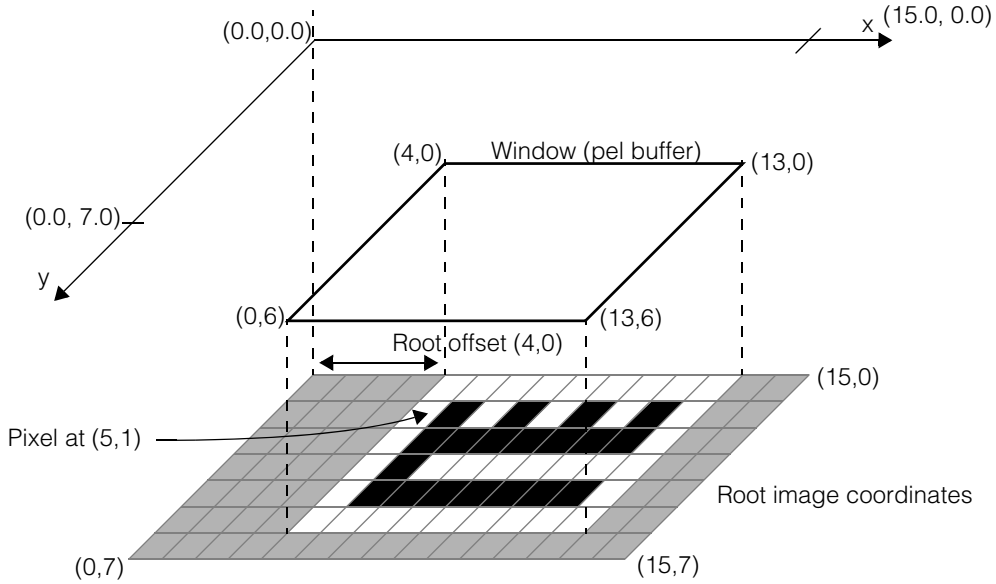


Figure 42. After calling thePelBuf.window(4, 0, 9, 6)

Calling the **window()** member function only made the window, or pel buffer, smaller. It did not change either the image coordinates or the client coordinates. All pixels in the image are still referenced the same way. For example, the pixel in the upper left corner of the shape is still at location (5,1). One important thing to keep in mind is that it is an error to try to access a point outside the bounds of a pel buffer.

Assuming that the image in *thePelBuf* is a binary image, consider the following code:

```
c_UInt8 pelVal;
pelVal = thePelBuf.get(5,1); // pelVal is set to 0
pelVal = thePelBuf.get(0,0); // Error! throws ccPel::BadCoord
```

Even though the root image has pixels at location (0,0), it is not possible to access them through *thePelBuf* because that location is outside of its bounds.

If you were to call **showWindowInfo()**, defined on page 232, again, it would print:

```
offset: (4, 0)
root offset: (4, 0)
width: 9 height: 6
```

Specifying a New Offset

As the example above showed, changing the size of the window did not change the image coordinate system. Suppose that after making the window smaller, you wanted to ensure that the upper left corner of *thePelBuf* was (0, 0) rather than (4, 0). The way to make this change is with **ccPelBuffer::offset()**.

To change the coordinate system, call **offset()** like this:

```
thePelBuf.offset(0,0);
```

This function sets coordinates of the upper left corner of the window to (0, 0). It does not change the size of the window, nor location of the window relative to the root image. Figure 43 shows the effect of calling this function.

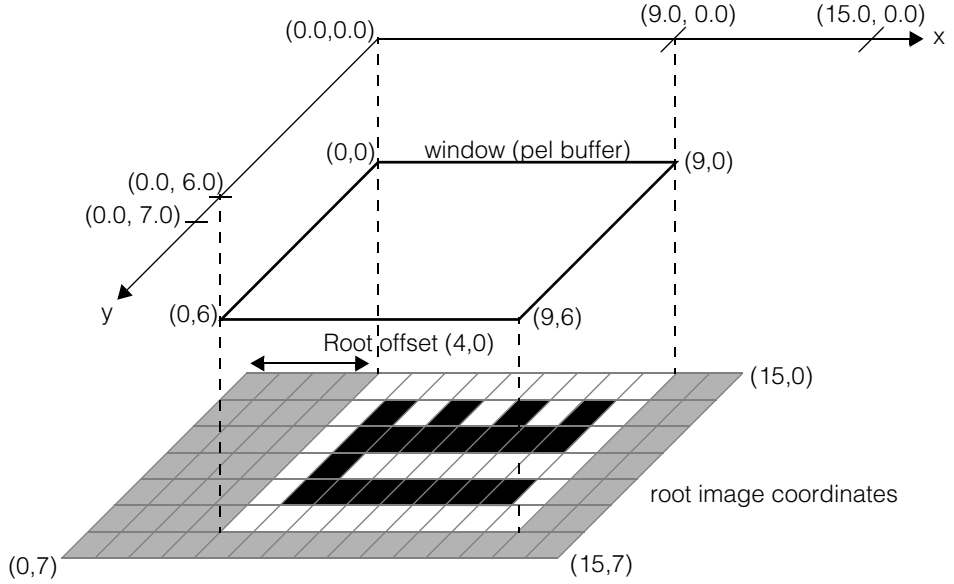


Figure 43. After calling offset(0,0)

One very important thing to notice about this operation is that changing the offset also changes the client coordinate system by the corresponding amount. As you can see from Figure 43, the root offset does not change.

If you were to call **showWindowInfo()**, defined on page 232, now, it would print:

```
offset: (0, 0)
root offset: (4, 0)
width: 9 height: 6
```

Additional Coordinate Systems

In addition to the image coordinates, CVL uses two other pixel-based coordinate systems: root image coordinates and window coordinates. They are used only to change the location of a window relative to the root image or to make changes in the size of a window.

Root Image Coordinates

Root image coordinates are used with the function **ccPelBuffer::windowRoot()** to specify the location of a window with respect to its root image. The x- and y-coordinates you pass to **windowRoot()** are relative to the upper left corner of the root image, which is always the point (0,0).

One way to enlarge the window in Figure 43 to encompass the entire root image (in other words, to make it look like the initial state in Figure 41) is to use the following code:

```
thePelBuf.windowRoot(0, 0, 15, 7);
thePelBuf.offset(0, 0);
```

Note that setting the offset to (0,0) is necessary because **windowRoot()** adjusts the offset. If you had not called **offset()**, the coordinates of the upper left corner of the window would have been (-4,0).

Window Coordinates

Window coordinates are used with the function **ccPelBuffer::subWindow()** to make a window smaller. The x- and y-coordinates that you pass to **subWindow()** are relative to the upper left corner of the window and treated as if that point were (0,0). Like other functions that change the size of a window, **subWindow()** changes the offset of the window.

Understanding Client Coordinates

You use image coordinates for image processing and for setting the sizes of windows. In reporting the results of a vision operation, your application may find it more convenient to report locations in native units such as inches or centimeters instead of pixels. These native units are defined by the client coordinate system.

The client coordinate system provides three basic services to your vision application:

- It provides a consistent coordinate system unaffected by changes in an image.
- It provides a calibration mechanism that lets you specify the orientation and units of your native coordinate system.
- It provides a real-valued coordinate system that lets you address fractional locations.

The client coordinate system uses a *transformation object* to map between client coordinates and image coordinates. A transformation object is an object of class **cc2XformBase** that is associated with every pel buffer. The transforms can be linear (**cc2XformLinear**) or nonlinear (**cc2XformPoly**) and map points from one 2D coordinate space to another. Note that while all Cognex vision tools support linear client coordinate transforms, only certain tools support nonlinear transforms. If you use a nonlinear client coordinate transform make sure the vision tools you use support it.

See *2D Transformations* on page 412 for information on linear and nonlinear transformations and how they are implemented in CVL).

The default transformation object associated with newly acquired and default-constructed pel buffers is the identity transformation in which client coordinates and image coordinates are identical.

Consistent Coordinates

The client coordinate system lets you concentrate on the location of a feature in an image rather than the location of the pixels that make up the image. In the course of analyzing an image, your vision application may perform several image processing functions that cause the pixel image to distort or change. If you did not have the client coordinate system, you would have to keep track of how each of the tools changed the location of a pixel. Since all the tools operate on the client coordinate system, they keep track of where a particular point corresponds to a particular pixel.

For example, to make your application run faster when using the Blob tool, you may want to subsample the image to reduce the number of pixels that the tool has to process. Without the client coordinate system, the blob tool would report its results in the pixel units of the subsampled image. You would have to convert those results by the subsampling factor yourself to correlate the results with your original image.

With the client coordinate system, the subsampled image's transformation object records the changes in scale so that when the blob tool returns its results, they are in client coordinates which correspond with the original image.

Calibration

The client coordinate transform provides two kinds of calibration: intrinsic calibration, which lets you correct for the inherent distortion in all vision systems; and arbitrary calibration which lets you map image coordinates to your own coordinate system and units.

Both kinds of calibration are implemented by a transformation object. Every pel buffer has a client coordinate transformation object associated with it. As illustrated in Figure 41 on page 231, the identity transformation object provides a 1:1 mapping between image coordinates and client coordinates. In your application, however, you may want to change the mapping. For instance, you may want to specify coordinates in units other than pixels when you use vision tools, or you may need to rotate or adjust the coordinate system. You can use the client coordinate transform to change the mapping between image coordinates and client coordinates.

Client Coordinate Transforms

Client coordinate transforms can be either linear (**cc2XformLinear**) or nonlinear (**cc2XformPoly**). All Cognex vision tools support linear client coordinate transforms, but only certain tools support nonlinear transforms. Some tools such as PatMax will linearize a nonlinear client coordinate transform and use the linearized version for its calculations. By far, the most common client coordinate transforms you will work with are linear. Some details of the linear transform are discussed in the following section.

Linear Transforms

The **cc2XformLinear** member functions let you specify coordinate transformations two different ways: scale-rotation and shear-angle. The difference between the two methods is the way you specify the elements of the four-element matrix of the transformation. Both methods use the two-element vector for xy-translation (see *2D Linear Transformations in CVL* on page 426).

The scale-rotation method for specifying transformation objects lets you set the rotation of the x-axis, rotation of the y-axis, the x-scale, and the y-scale. The shear-aspect method lets you specify the scale of the coordinate system, the aspect ratio of the x- to the y-axis, the shear angle, and the rotation of the coordinate system (see *2D Linear Transformations in CVL* on page 426). Both methods are equivalent, so you can use the one that suits your application or your existing algorithms. In general, the scale-rotation method is easier to work with.

In the scale-rotation specification, the components of the transformation object are used as follows to map the image coordinates $\mathrm{I}_{(x,y)}$ to the client coordinates $\mathrm{C}_{(x,y)}$:

$$\begin{bmatrix} C_x \\ C_y \end{bmatrix} = \begin{bmatrix} \cos r_x & -\sin r_y \\ \sin r_x & \cos r_y \end{bmatrix} \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} I_x \\ I_y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$= \begin{bmatrix} s_x \cos r_x & -s_y \sin r_y \\ s_x \sin r_x & s_y \cos r_y \end{bmatrix} \begin{bmatrix} I_x \\ I_y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$C_x = I_x s_x \cos r_x - I_y s_y \sin r_y + t_x$$

$$C_y = I_x s_x \sin r_x + I_y s_y \cos r_y + t_y$$

where

s_x and s_y are the x-scale and the y-scale,
r_x and r_y are the x-rotation and the y-rotation,
t_x and t_y are the x-translation and the y-translation.

In the shear-aspect specification, the components of the transformation object are used as follows to map the image coordinates $I_{(x,y)}$ to the client coordinates $C_{(x,y)}$:

$$\begin{bmatrix} C_x \\ C_y \end{bmatrix} = S \begin{bmatrix} \cos R & -\sin R \\ \sin R & \cos R \end{bmatrix} \begin{bmatrix} 1 & -\tan K \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & A \end{bmatrix} \begin{bmatrix} I_x \\ I_y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$= \begin{bmatrix} S\cos R & AS(-\sin R - \cos R \tan K) \\ S\sin R & AS(-\cos R - \sin R \tan K) \end{bmatrix} \begin{bmatrix} I_x \\ I_y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$C_x = I_x S\cos R + I_y AS(-\sin R - \cos R \tan K) + t_x$$

$$C_y = I_x S\sin R + I_y AS(\cos R - \sin R \tan K) + t_y$$

where

S is the scale,
A is the aspect,
R is the rotation angle,
K is the shear angle,
t_x and t_y are the x-translation and the y-translation.

The **cc2XformLinear** class provides member functions to set and get each of the elements. However, it is important that you not mix elements from one specification with elements from the other specification. For example, if you specified a shear angle (K) using the shear-aspect method, then $A \neq s_y / s_x$.

Table 41 illustrates the scale-rotation specification of the transformation object.

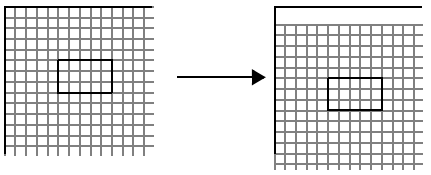
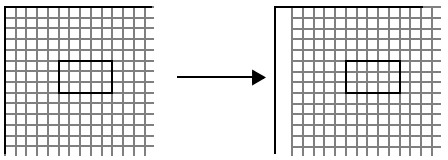
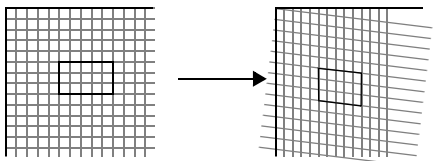
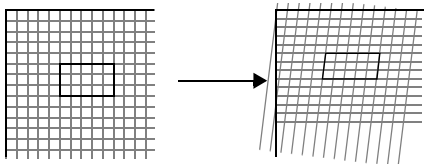
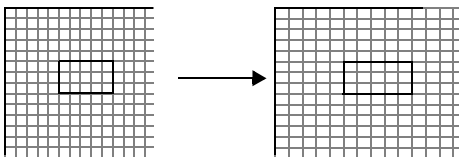
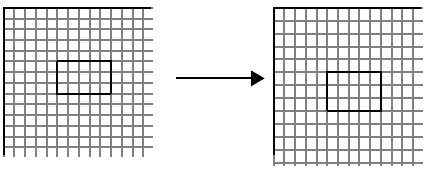
Trans-formation	Appearance	Definition
Y-Translation		Moving the origin of the client coordinate system relative to the pixel coordinate system
X-Translation		Moving the origin of the client coordinate system relative to the pixel coordinate system
X-Rotation		Rotating the x-axis of the client coordinate system relative to the pixel coordinate system
Y-Rotation		Rotating the y-axis of the client coordinate system relative to the pixel coordinate system
X-Scaling		Changing the size of the x-axis units of client coordinate system
Y-Scaling		Changing the size of the y-axis units of client coordinate system

Table 41. Scale-rotation specification of the transformation object

Table 42 illustrates the shear-aspect specification of transformation objects.

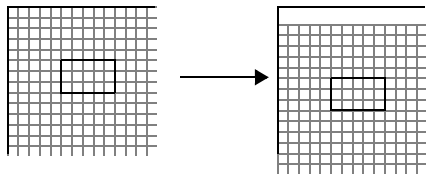
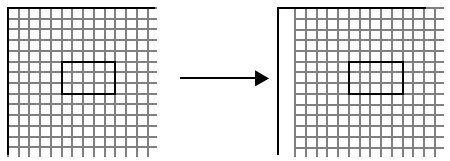
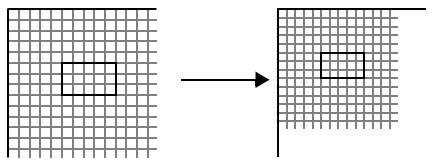
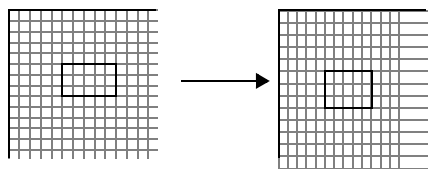
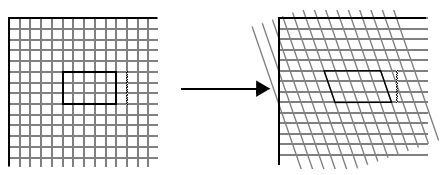
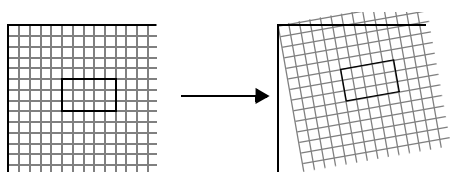
Trans-formation	Appearance	Definition
Y-Translation		Moving the origin of the client coordinate system relative to the pixel coordinate system
X-Translation		Moving the origin of the client coordinate system relative to the pixel coordinate system
Scale		Uniformly scaling the x- and y-axis units of the client coordinate system.
Aspect		Changing the ratio of the x-axis to the y-axis
Shear		Rotating the y-axis, without changing the height of the y-units.
Rotation		Rotating the x- and the y-axis.

Table 42. Shear-aspect specification of transformation objects

Working with Client Coordinates

The key to working with client coordinates is understanding transformation objects. The default transformation object associated with every newly acquired pel buffer performs a 1:1 mapping between image coordinates and client coordinates. Each unit in the client coordinate system corresponds to one pixel unit in the image coordinate system. Figure 44 shows the default client coordinate system relative to image coordinates and the root image.

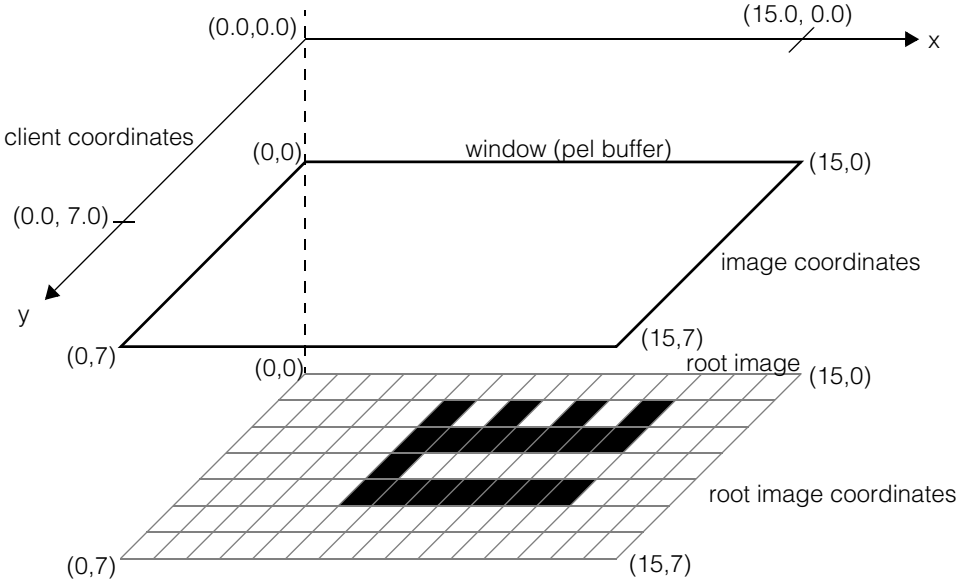


Figure 44. The default transformation object maps 1:1 to image coordinates

If your application works with pixel units and you do not need to correct for pixel or optical distortion, you can use the default transformation object. The vision tools will change the transformation object whenever you scale or rotate an image to maintain coordinate correspondence with your original image.

Remember that if you change the offset of a window, the (0,0) point of the client coordinate system moves by the corresponding amount. See Figure 43 on page 234.

Note Each window (pel buffer) has its own set of client coordinates. You can create multiple windows with different client coordinates on the same root image.

Using the Grid-of-Dots Calibration Tool

The easiest way to create a transformation object to map to standard measurements units, such as millimeters, is to use the Grid-of-Dots Calibration tool. This tool examines a grid of dots of a known size and constructs the appropriate transformation object for you.

To use the Grid-of-Dots Calibration tool do the following:

1. Create a **ccGridCalibParams** object to describe the calibration grid you are using.
2. Create a **ccGridCalibResults** object to hold the results of the calibration.
3. Acquire an image of the calibration grid.
4. Run the function **cfCalibrationRun()** to fill the **ccGridCalibResults** object.
5. Call **ccGridCalibResults::clientFromImageXform()** to get the transformation object.
6. Use **ccPelBuffer::clientFromImageXform()** to associate the transformation object with the pel buffers of the images you acquire in your application.

For example, assume that *gridImage* is a pel buffer that contains the calibration grid image, *myParams* contains your calibration parameters that describe the grid.

```
ccGridCalibResults myResults;
cc2XformLinear calibClientFromImageXform;

     // gridImage has been previously acquired
     // myParams has been previously set up
cfCalibrationRun(gridImage, myParams, myResults);

calibClientFromImageXform = myResults.clientFromImageXform();
```

When you acquire an image, you associate the transformation object from the calibration tool with the pel buffer:

```
thePelBuf = fifo->complete();
thePelBuf.clientFromImageXform(calibClientFromImageXform);
```

For more information about the calibration tool, see the description of the Calibration tool in the *Vision Tool Guide* and the classes **ccGridCalibParams**, **ccGridCalibResults**, and the function **cfCalibrationRun()** in the *CVL Class Reference*.

Mapping Between Image and Client Coordinates

All of the vision tools that operate on a pel buffer and require or return locations, perform mapping between client and image coordinates automatically.

To map between client and image coordinates yourself, use **clientFromImageXform()** and **imageFromClientXform()** along with the **cc2XformLinear** multiplication operator. To map an arbitrary point (x,y) in image coordinates to client coordinates, write something like this:

```
cc2Vect clientPt;

clientPt = thePelBuf.clientFromImageXform() * cc2Vect(x, y);
```

The function **cc2XformLinear::mapPoint()** provides the same functionality as the multiplication operator. The **cc2XformLinear** class provides several other member functions that you can use to map angles, areas, and vectors between image and client coordinates.

Setting Up Transformation Objects Manually

Although using the Grid-of-Dots Calibration tool is the easiest way to set up a transformation object, you can also create a transformation object manually. To create the transformation object, use the constructor for the specification method that suits your application.

To associate a new transformation object with a window, use either **ccPelBuffer::clientFromImageXform()** or **ccPelBuffer::imageFromClientXform()** depending on the direction of the transformation. Whichever function you use, the single transformation object associated with a pel buffer is set.

Changing Left-Handed Coordinates to Right-Handed

For example, to change the default left-handed coordinate system to a right-handed coordinate system:

```
cc2XformLinear rhClientFromLhImage(cc2Vect(0,0),
                        ccDegree(0), ccDegree(180), 1.0, 1.0);
thePelBuffer.clientFromImageXform(rhClientFromLhImage);
```

The object *rhClientFromLhImage* rotates the y-axis 180° as shown in Figure 45.

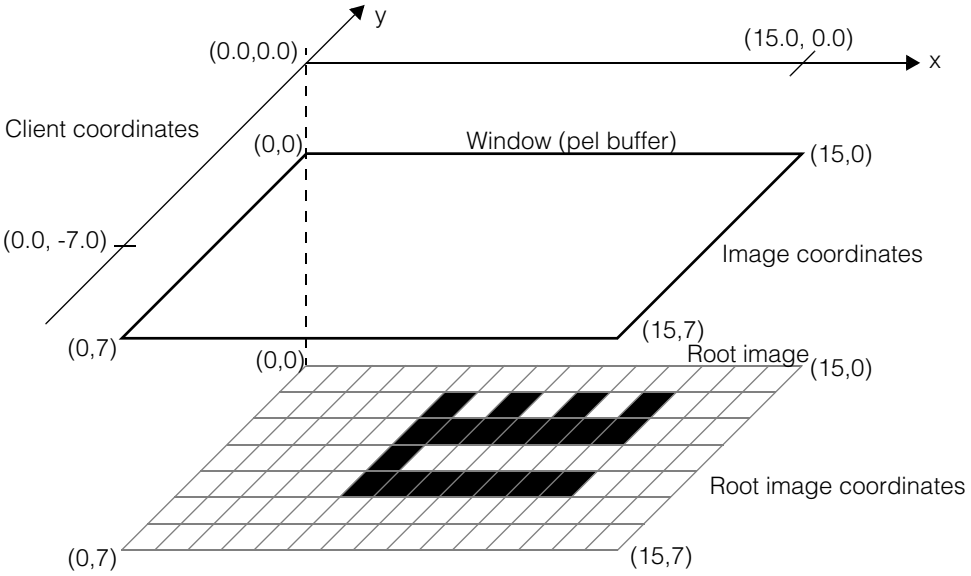


Figure 45. After rotating the y-axis 180°

Changing the Scale

To scale the x- and y-coordinates so that each pixel maps to 2 client units:

```
cc2XformLinear scaleClientFromImage(cc2Vect(0,0),
                        ccDegree(0), ccDegree(0), 2.0, 2.0);

thePelBuffer.clientFromImageXform(scaleClientFromImage);
```

The transformation object *scaleClientFromImage* scales each pixel to 2 client units as shown in Figure 46.

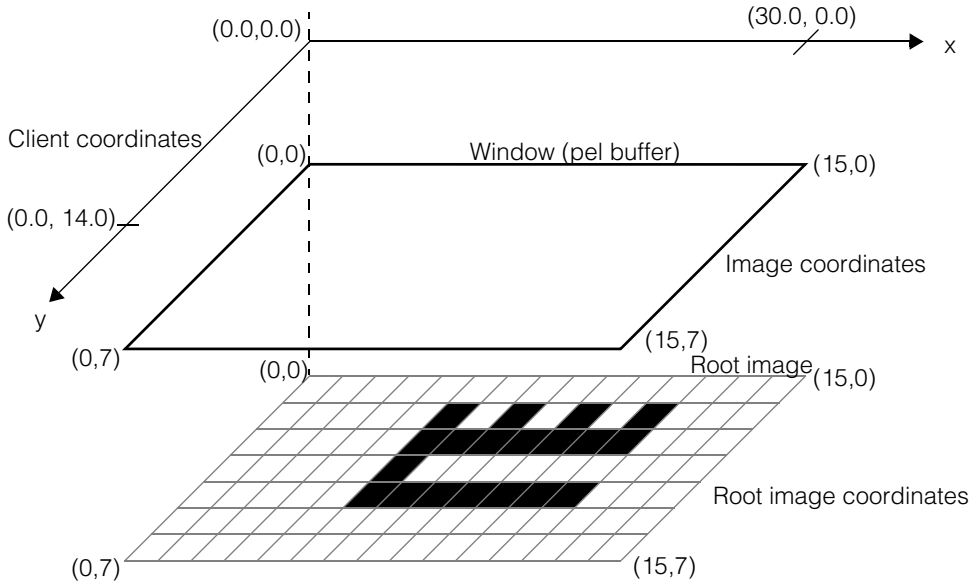


Figure 46. After scaling the x- and y-axis by 2

Scaling, Rotating, and Translating

To change to a right-handed coordinate system, scaled at 2 client units to the pixel, and the (0,0) point at the upper left pixel of the crown figure, do the following:

```
cc2XformLinear rhScaleClientFromImage(cc2Vect(-10,2),
                        ccDegree(0), ccDegree(180), 2.0, 2.0);

thePelBuffer.clientFromImageXform(scaleClientFromImage);
```

The transformation object *rhScaleClientFromImage* rotates the y-axis 180°, scales each pixel to 2 client units, and translates the (0,0) point the location corresponding to (5,1) in image coordinates. Specifying the translation can be confusing because it is

specified in terms of the rotated and scaled coordinate system. The easiest way to specify the translation is to describe it as the client coordinates that correspond to the point (0,0) in image coordinates. Figure 47 shows the effect of this transformation.

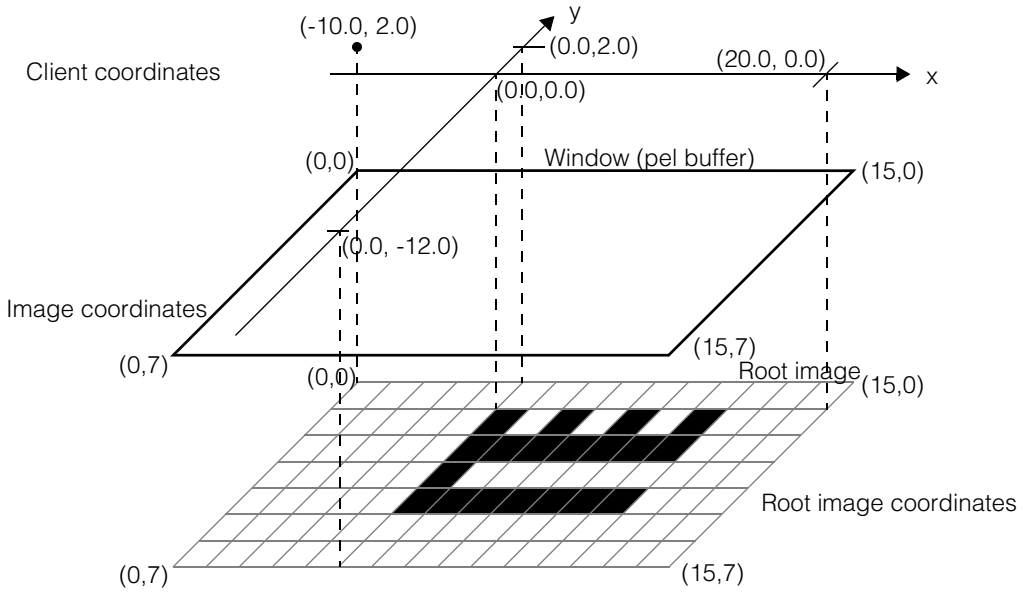


Figure 47. After scaling, rotating, and translating

Applying More Than One Transformation

You may have noticed that **ccPelBuffer::clientFromImageXform()** and **ccPelBuffer:imageFromClientXform()** allow you to set only one transformation object. By composing the matrices (using the multiplication operator) you can apply more than one transformation to a pel buffer.

The following example shows how to compose three transformation objects with one pel buffer.

- bottomFromTop
  Translates the (0,0) point so that it is +20 image units in the y-axis
- rhFromLh
  Rotates the y-axis 180° to create a right handed coordinate system
- calibLHFromImage
  A calibrated transformation returned by the calibration tool.

To use all these transformation objects with one pel buffer so that the (0,0) point is at the point (0,20) in image coordinates, the coordinate system is calibrated, and so that the coordinate system is a right handed coordinate system:

```
cc2XformLinear calibLHFromImage;
cc2XformLinear botFromTop(cc2Vect(0,-20),
                          ccDegree(0), ccDegree(0), 1.0, 1.0);
cc2XformLinear rhFromLh(cc2Vect(0,0),
                        ccDegree(0), ccDegree(180), 1.0, 1.0);

calibLHFromImage = calibResults.clientFromImageXform();

thePelBuf.clientFromImageXform(rhFromLH
                                * calibLHFromImage
                                * bottomFromTop)
```

The important thing to remember about composing transformation objects is that the transformations are applied from right to left.

Displaying Graphics 6

This chapter describes how to use the Cognex Vision Library to display static, interactive, and result graphics.

This chapter contains the following major sections:

Some Useful Definitions defines certain terms you will encounter as you read.

Overview of Graphics describes the types of graphics and the general steps involved in drawing and displaying graphics.

Using the Graphics Classes introduces the classes that provide the graphics display environment.

Defining Graphic Properties describes how to use graphic properties objects to define drawing properties such as pen color, width, style, end cap style, fill, and arrow heads.

Using the Overlay Plane describes how to use the overlay plane to render graphics independently of images.

Displaying Static Graphics shows you how to draw static graphics in a display.

Displaying and Using Interactive Graphics shows you how to draw interactive graphics in a display.

Displaying Result Graphics shows you how to draw result graphics in a display.

Graphic Display Application Notes describes special situations you may encounter in your graphic display applications.

Display Examples describes the sample code shipped with CVL that illustrates graphics display topics.

Some Useful Definitions

The following terms may be useful in reading this chapter.

graphic A data structure used to hold graphics geometry. Also called a shape or an object.

graphic list A collection of static graphics stored in an iterable list

image layer A plane holding the contents of an image that can be independently processed. Static and manipulable graphics added to either the overlay layer or the image layer are automatically redrawn when the image in the image layer changes.

interactive graphics Graphics that can be manipulated with a mouse. These graphics are also called UI shapes

object An instantiated graphics class is sometimes referred to as an object. It may also be called a graphic or a shape.

overlay layer A buffer used to render non-destructive static and manipulable graphics. When displayed, the overlay layer is on top of the image layer and uses a pass-through value to reveal the image layer underneath it.

result graphics Graphics that show the result of a vision tool operation.

shape A data structure used to hold graphics geometry. Also called a graphic or an object.

sketch Graphic contents of a tablet.

static graphics Graphics that cannot be manipulated with the mouse after they are drawn on the display.

tablet An abstract drawing space in which you can draw graphics and text. The graphics in a tablet, called the sketch, can then be drawn on a display console.

UI shapes The family of interactive graphic classes that can be displayed and manipulated in a **ccDisplay**-derived class window. These shapes classes are named **ccUI***.

Overview of Graphics

Graphics are text, diagrams, and other synthetic images that can be drawn in a display window. Graphics are used to annotate images and provide visual results of vision tool operations.

CVL supports three types of graphics:

- Static graphics that remain stationary on the display and cannot be manipulated with a mouse. Static graphics are stored in tablet objects.
- Interactive graphics that can be manipulated with a mouse. Interactive graphics are stored in display objects.
- Result graphics that show vision tool operations. Result graphics are static graphics produced by vision tools that are stored in an iterable list before they are displayed.

All types of graphics can be drawn in the image plane or in a separate graphics overlay plane. The basic steps to display static graphics are:

1. Create a tablet object.
2. Create a graphic properties object to define pen color, style, width, fill, end cap style and other drawing properties.
3. Create graphics shapes and draw them on the display using the tablet **draw()** function.
4. Remove the graphics from the display.

Static and Interactive Graphics Code Sample

The code sample that follows shows how to display static and interactive graphics in a **ccDisplayConsole** window. The code sample in the *Displaying Result Graphics* on page 292 shows how to display results graphics.

```
#include <ch_cvl/vp8100.h>
#include <ch_cvl/acq.h>
#include <ch_cvl/vidfmt.h>
#include <ch_cvl/windisp.h>
#include <ch_cvl/xform.h>
#include <ch_cvl/shapes.h>
#include <ch_cvl/uishapes.h>

int cfSampleMain(int, TCHAR** const)
{
  // Find the 8100 frame grabber
  cc8100m& fg = cc8100m::get(0);
  const ccStdVideoFormat& fmt =
```

```
        ccStdVideoFormat::getFormat(cmT("Sony XC75 640x480"));
ccStdGreyAcqFifoPtrh fifo = fmt.newAcqFifo(fg);

// Step 1: Create a display console
ccDisplayConsole *display;
display = new ccDisplayConsole(cmT("Display Console"),
                               ccIPair(20,20), ccIPair(480,360));

// Step 2: Set the display console's attributes
display->mag(-2);
display->closeAction(ccDisplayConsole::eCloseDisabled);

// The next three settings are the defaults
display->showToolBar(true);
display->showScrollBar(true, true);
display->showStatusBar(true);

// Use the Status Bar Text for any text you choose
display->statusBarText(cmT("Ready"));

// Start an acquisition
fifo->start();

// Get the acquired image
ccPelBuffer<c_UInt8> pb = fifo->complete();
if (!pb.isBound())
{
  MessageBox(NULL, cmT("Acquisition failed"),
        cmT("Display Error"), MB_OK);
  return 0;
}
// Set the transformation object
// In this example, 40 image units (pels)
// map to 1 client coordinate unit
cc2Xform xform(cc2Vect(0,0), ccRadian(0), ccRadian(0),
                                                    20.0, 20.0);
pb.imageFromClientXform(xform);

// Step 3: Display the image
display->image(pb, false);

// Draw some graphics into the image layer
ccUITablet tablet;
ccPoint where(24,2);
ccGraphicProps props;
props.penColor(ccColor::greenColor());
```

```
  tablet.drawPointIcon(ccPoint(200, 100), props);
  tablet.draw(cmT("Point (24,2)"), where, ccColor::blue,
       ccColor::yellow, ccUIFormat());

// Display the tablet sketch in three different coord systems
  display->drawSketch(tablet.sketch(),
                                      ccDisplay::eDisplayCoords);
  display->drawSketch(tablet.sketch(),ccDisplay::eImageCoords);
  display->drawSketch(tablet.sketch(),ccDisplay::eClientCoords);

// Draw a 20x20 rectangle in display coordinates
  ccUIRectangle *uiRect = new ccUIRectangle;
  uiRect->rect(ccRect(cc2Vect(0,0), cc2Vect(20,20)));
  uiRect->color(ccColor::white);
  uiRect->condVisible(true);
  uiRect->drawLayer(ccUITablet::eOverlayLayer);
  display->addShape(uiRect,ccDisplayConsole::eDisplayCoords);

// Draw a 20x20 rectangle in image coordinates
  ccUIRectangle *uiRect1 = new ccUIRectangle;
  uiRect1->rect(ccRect(cc2Vect(0,0), cc2Vect(20,20)));
  uiRect1->color(ccColor::white);
  uiRect1->condVisible(true);
  uiRect->drawLayer(ccUITablet::eOverlayLayer);
  display->addShape(uiRect1, ccDisplayConsole::eImageCoords);

// Draw a 20x20 rectangle in client coordinates
  ccUIRectangle *uiRect2 = new ccUIRectangle;
  uiRect2->rect(ccRect(cc2Vect(0,0), cc2Vect(20,20)));
  uiRect2->color(ccColor::white);
  uiRect2->condVisible(true);
  uiRect->drawLayer(ccUITablet::eOverlayLayer);
  display->addShape(uiRect2, ccDisplayConsole::eClientCoords);

  // Display message box to pause application
  MessageBox(NULL, cmT("Display complete"),
     cmT("Display Sample"), MB_OK);

  // Step 4: Release display object
  fifo = ccStdGreyAcqFifoPtrh(0);   // delete the fifo...
  delete display;                   // ... before the display
  return 0;
}
```

Figure 48 below shows the resulting display when you run this sample code.

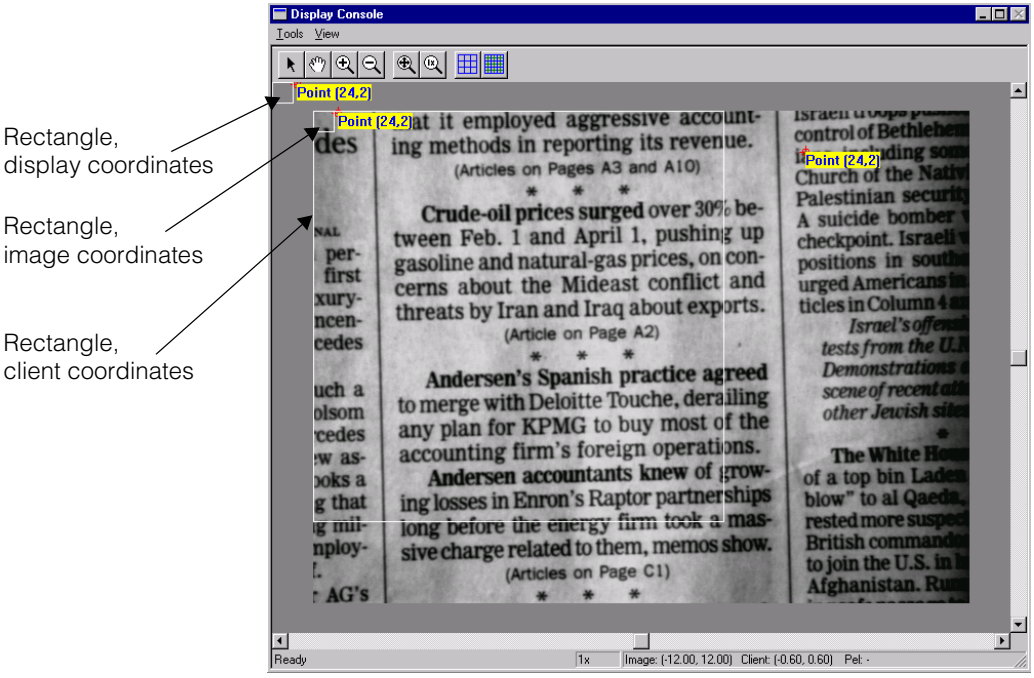


Figure 48. Display example

The example creates a point icon and a label at the same location (24,2). The location specifies the upper left-hand corner of the label. The point icon at 24,2 is too small to see in this figure. The example then displays the point icon and the label using three different coordinate systems to illustrate their differences. *eDisplayCoords* displays the pair at location 24,2 of the display window. *eImageCoords* displays the pair at location 24,2 of the image. *eClientCoords* displays the pair at location 480,40 of the image because in our example, one client unit is equal to 20 pixels for both x and y. (Note that the label size is not scaled by the client coordinate transform).

The example also creates three rectangles 20x20 pixels and displays each at location 0,0 in *eDisplayCoords*, *eImageCoords*, and *eClientCoords*. Note that the display space rectangle and the image space rectangle are the same size, whereas the client coordinate rectangle is larger because of the client coordinate transform.

Using Tablets and Displays

All of the display classes derive from **ccUITablet**. Display classes support image display, and they support graphics display using the inherited **ccUITablet** sketch functionality. **ccUITablet** objects cannot display graphics directly, but can record graphics that can be displayed later using display class functions.

You can use tablets in either of the following two ways:

1. Instantiate a **ccUITablet** object directly, and call its **draw()** functions to draw graphics into its sketch storage. Recording must be enabled for the drawings to be internally retained. Recording is enabled by default.

   Later you can display the recorded graphics using a display class member function. For example:

   ```
   disp->drawSketch(tablet.sketch(),ccDisplay::eClientCoords);
   ```

2. Call the **ccUITablet** inherited **draw()** functions from a display class and the graphics will be displayed immediately. Note that if **recordOn()** == false, the drawn graphics are not recorded in the tablet's sketch list and will be lost if you update the screen. To make the graphics a permanent part of the display, set **recordOn()** == true before calling the draw functions.

CVL Display Programming Requirements

All CVL Display applications must provide their own mutex locks or semaphores around all calls into CVL Display code. Only one thread at a time is allowed to make calls to CVL Display routines.

Using the Graphics Classes

CVL uses the following classes to provide an environment in which you can draw static, interactive, and result graphics.

Class	Description	Header File Location
ccSketch **ccSketchMark** **ccUITablet**	Classes that provide the drawing and display environment for graphics.	*uitablet.h*
ccAffineRectangle **ccCircle** **ccEllipse2** ...	A set of classes that provides various static shapes.	*shapes.h*
ccUIAffineRect **ccUICircle** **ccUIEllipse** ...	A set of classes that provides various user interactive shapes.	*uishapes.h*
ccGraphicList	A class that provides an iterable list of graphic items.	*glist.h*
ccGraphic **ccGraphicBuiltIn** **ccGraphicSimple** ...	A set of classes that provides various graphic list shapes.	*glist.h*
ccGraphicProps	A class that defines graphic drawing properties, such as pen color, width, style, join, fill, pen end cap styles, whether to show vertices or arrow heads, and if arrow heads are shown whether they are pointing forward or backward.	*dispprop.h*

Table 43. Graphics classes.

Defining Graphic Properties

The **ccGraphicProps** class allows you to define graphic properties for all static graphics and manipulable UI elements. Every manipulable (**ccUIShapes**-derived) shape has a **ccGraphicProps** member into which user-defined graphic properties are stored. Static graphics can use the appropriate **ccUITablet::draw()** member functions and specify the graphic properties object as an argument.

Setting any graphical property to anything other than the default will impact the rendering performance of graphics. The best performance is achieved by using the default values.

The **ccUIShapes::props()** member function gets or sets the user-defined graphic properties for a UI shape. The **ccUIShapes::getGraphicProps()** member function gets the current rendering state of the shape, which depends on whether or not it is selected.

Setting the Color of Graphic Elements

The default color of all new graphic elements, as set by the **ccGraphicProps** default constructor, is cyan.

For manipulable shapes, you can change the drawing color, for example to green, by using the **ccGraphicProps::penColor()** setter as follows:

```
ccGraphicProps p;
p = someUIShape.props();
p.penColor(ccColor::greenColor());
someUIShape.props(p);
```

For static graphics, you can pass a separately instantiated graphic properties object as an argument to **ccUITablet::draw()**. For example, you can change the drawing color of a static line segment as follows:

```
ccGraphicProps p;
p = someUIShape.props();
p.penColor(ccColor::greenColor());
tablet.draw(ccLineSeg(ccPoint(0,0), ccPoint(100,100), p);
```

Whether you use the **draw()** function or one of the convenience functions to draw graphics (see *Drawing Shapes* on page 263), you must specify the color of the shape. Note that the **draw()** functions take a **ccGraphicProps** object as the second argument, whereas the **draw<shape>()** convenience functions take a **ccColor** object as the second argument. See also *Setting the Color of Graphic Elements* on page 256.

You can use one of the predefined stock colors, as in the preceding example, or use indexed or RGB colors. Because of the different color maps used to display graphics depending on the desktop depth, Cognex recommends using RGB colors rather than

indexed colors whenever possible. This will make your applications more portable to different desktop settings. See *Constructing Color Objects* on page 203 for details. See *Color Maps* on page 197 for more information on color maps.

Modifications to ccUIScope

The pure virtual **ccUIScope::draw()** member functions that took a *const ccColor&* as the second argument in pre-6.0 versions of CVL have been obsoleted and replaced with functions that take a *const ccGraphicProps&* as the second argument. Using the obsolete functions will cause compilation or link errors or both.

The **ccUIScope** class is not documented in the *CVL Class Reference* as it is an internal class used only by the display framework. Classes that derive from **ccUIScope** (for example, **ccWin32UIScope**) implement the drawing functions to display graphics on various operating systems.

If your application includes classes that derive directly from **ccUIScope** and have drawing functions with signatures such as:

```
virtual void draw(const ccLineSeg&, const ccColor&) = 0;
```

the new display framework will call functions such as:

```
virtual void draw(const ccLineSeg&, const ccGraphicProps&) = 0;
```

resulting in link errors indicating that the class has changed.

The following are a few examples of link errors that may occur if you have pre-6.0 code with display objects derived from **ccUIScope** and recompile it using CVL 6.0 or later:

```
cvl-disp-winnt-winscope.obj : error LNK2001: unresolved external
symbol "public: virtual void __thiscall
ccWin32UIScope::drawToRel(long,long,class ccColor const &)"
(?drawToRel@ccWin32UIScope@@UAEXJJABVccColor@@@Z)
```

```
cvl-disp-winnt-winscope.obj : error LNK2001: unresolved external
symbol "public: virtual void __thiscall
ccWin32UIScope::drawTo(class ccPoint const &,class ccColor const
&)" (?drawTo@ccWin32UIScope@@UAEXABVccPoint@@ABVccColor@@@Z)
```

```
cvl-disp-winnt-winscope.obj : error LNK2001: unresolved external
symbol "public: virtual void __thiscall
ccWin32UIScope::draw(class ccPoint const &,class ccColor const &)"
(?draw@ccWin32UIScope@@UAEXABVccPoint@@ABVccColor@@@Z)
```

To avoid these link errors, do the following:

1. Modify your code to change the signatures of the virtual **ccWin32UIScope::draw** member functions that use a second argument of type *const ccColor&* to use a *const ccGraphicProps&* instead (see above).
2. In the implementations of your draw functions, you can still access the color property from the **ccGraphicProps** interface, for example as follows:

```
virtual void draw(const ccLineSeg& ls,
  const ccGraphicProps& gp) {
    const ccColor& color = gp.penColor();
    ...
}
```

If your application derives from **ccWin32UIScope** and you have overloaded the draw functions that take a *const ccColor&*, change your derived class to now take a *const ccGraphicProps&*. This will prevent compilation or link errors caused by the argument mismatch.

Setting the Pen Style and Width

The **penStyle()** and **penWidth()** member functions of **ccGraphicProps** allow you to get and set the style and width, respectively, of the pen used to draw graphic elements. For example, you can set the pen style to dashed as follows:

```
ccGraphicProps p;
p = someUIShape.props();
p.penStyle(ccGraphicProps::ePenStyleDash);
someUIShape.props(p);
```

Multiple Selection and Deletion

To delete multiple selected shapes once the delete command is received, call **ccDisplay::removeShape()** for each selected shape. Note that you should not call delete on a shape object added to a display with **ccDisplay::addShape()**. These shapes should only be deleted with **ccDisplay::removeShape()**.

You can select and deselect multiple **ccUIShapes** using the <*Shift*> key if the *multiselect* attribute is enabled for the shape. You enable the *multiselect* attribute by calling **ccUIObject::multiSelectable(true)** for the shape before adding it to the display. Note that multi-selection only works on objects that are siblings of one another.

CVL also allows you to configure the modifier keys used for multiple graphics selection and panning using the **ccWin32Display::multiSelectKey()** and **ccWin32Display::panKey()** functions. You can assign a given modifier key, for example *<Ctrl>* or *<Shift>*, to only one of these functions.

Specifying Virtual Keys for Multiple Selection and Panning

The **ccWin32Display::multiSelectKey()** function specifies the modifier key used to select multiple graphic objects. By default, you hold down the *<Shift>* key while clicking to select multiple graphic objects. The **ccWin32Display::multiSelectKey()** function allows you to specify either *<Ctrl>* or *<Shift>* as the multi-select key.

Specifying the Multi-Select Key

To specify *<Ctrl>* as the multi-select key for a Win32 display, use:

```
display->multiSelectKey(ccKeyBoardEvent::eControl);
```

To specify *<Shift>* as the multi-select key, use:

```
display->multiSelectKey(ccKeyBoardEvent::eShift);
```

To disable the use of any modifier key for multi-selection, use:

```
display->multiSelectKey(ccKeyboardEvent::eNoVKey);
```

Specifying the Pan Key

The **ccWin32Display::panKey()** function specifies the modifier key used to select the pan mode. By default, you hold down the *<Ctrl>* key while clicking to pan the image. The **ccWin32Display::panKey()** function allows you to specify either *<Ctrl>* or *<Shift>* as the pan key.

To specify *<Shift>* as the pan key for a Win32 display, use:

```
display->panKey(ccKeyBoardEvent::eShift);
```

To specify *<Ctrl>* as the pan key, use:

```
display->panKey(ccKeyBoardEvent::eControl);
```

To disable the use of any modifier key for panning, use

```
display->panKey(ccKeyboardEvent::eNoVKey);
```

Using the Overlay Plane

Using the overlay plane to render graphics independently of images can greatly improve the performance of your vision application. When you draw in the image plane, each time you display a new image, your drawings must be regenerated, adding time to your application. If you draw in the overlay plane, new images do not affect your drawings and your application will run faster.

The performance improvement that you actually realize depends upon how the overlay plane is used, your application, the video card in the host PC, and whether you use an AGP or PCI video card. Here are some considerations and recommendations for using the overlay plane.

Enabling the Overlay Plane

You can use **ccUITablet::overlaySupported()** to determine whether your hardware platform supports an overlay plane. On some platforms, the overlay plane is automatically enabled. On other platforms, such as the MVS-8120, the overlay plane must be explicitly enabled by calling:

```
ccWin32Display::enableOverlay(ccWin32Display::eOverlayPlane)
```

See the following summary:

Display type	Overlay default	Description
ccWin32Display	Disabled	Call **ccWin32Display::enableOverlay()** to enable.
ccDisplayConsole	Disabled	

Note If the overlay plane is not enabled, no graphics will be displayed.

Live Display with Overlay Graphics

Use of the overlay plane is best suited for applications that use live display and have many graphics. This is because the graphics in the overlay layer do not need to be re-rendered when running live display unless the image transform changes. If this happens, all graphics on both the image layer and overlay layer will be rendered again.

Video Memory Requirements

Using the overlay plane requires additional video memory. To use the overlay plane, your video card must provide at least 16 MB of video memory for each display window. If your desktop setting is 32 bits, you may need 32 MB of video memory for each display window.

If your video card does not contain sufficient memory for the overlay plane, your application may render graphics and perform live display slower with the overlay plane enabled than without. This is because system memory is being used in place of video memory, which can result in much slower rendering and displaying.

Even with the recommended video memory, because of differences among video cards, always test your application with and without the overlay plane enabled.

Using the Color Map with Overlay Graphics

Two CVL display functions provide access to the overlay color map : **ccDisplay::overlayColorMap()** and **ccDisplay::getPassThroughValue()**.

The following sections describe the use of overlay graphics on host-based displays.

Retrieving the Color Map for Overlay Graphics

The **ccDisplay::overlayColorMap()** function retrieves the overlay color map, a format conversion table that maps pixel values in an overlay display buffer to RGB values. The size of the vector returned is both platform and hardware-configuration dependent. There is a one-to-one correspondence between the vector's index and pixel values in the overlay layer display buffer. The length of the vector returned is one more than the maximum pixel value that can exist in the overlay display buffer. The longest-possible color map table has 256 entries.

Use **ccDisplay::getDisplayedImage()** to retrieve the overlay display buffer.

You cannot set the overlay color map for any **ccDisplay**-derived platform in CVL 6.0.

Retrieving the Pass-Through Value for Overlay Graphics

The **ccDisplay::getPassThroughValue()** function retrieves the pass-through value used for overlay display buffers. All overlay display buffers are initialized with this value. The pass-through value is the value used when the overlay display buffer is combined with the image display buffer such that any value in the overlay display buffer equal to the pass-through value is transparent, revealing the image display buffer underneath it.

The pass-through value can be any number from 0 through 255. In the CVL display classes, it is set to the following default values:

- Within the **ccDisplay** base class, the pass-through value is set to 0.
- The **ccWin32Display** derived class implements the pass-through value as 243.

Some **ccDisplay**-derived classes require the overlay layer to be explicitly enabled, but enabling of the overlay layer is not required for this function.

operator==() Overloads

Operator== overloads are available for **ccPackedRGB16Pel** and **ccPackedRGB32Pel** objects, for example to allow you to compare a 16 or 32-bit value to a pass-through value of the same bit width.

Sample Code Using ccDisplay::getPassThroughValue()

The following sample code demonstrates the use of **ccDisplay::getPassThroughValue()** together with operator==() for **ccPackedRGB16Pel** to determine which pixels in the overlay layer of an image have the pass-through value:

```
ccPelBuffer<ccPackedRGB16Pel> my_pb(640,480);
display.getDisplayedImage(my_pb, eOverlayLayer);
ccPackedRGB16Pel pass_value;
display.getPassThroughValue(pass_value); // New function

// Test some pixel values for the pass-through value.
ccPackedRGB16Pel *ptr = my_pb.pointToRow(0);
for(i = 0; i < my_pb.width() < ++i) {
  if(ptr[i] == pass_value )   // test for equality
  {
    // This pixel is a pass-through value.
  }
```

Displaying Static Graphics

As you design your vision application, you may want to add text and graphics to annotate the images in display consoles. This section describes how to draw static graphics. Static graphics are graphics that do not respond to mouse clicks.

Static Graphics Overview

To draw static graphics, you create a tablet. A tablet is a drawing environment that accumulates shapes and text in an internal data structure called a sketch. To render the graphics on a display, you draw the contents of the tablet's sketch.

For example, the following code creates a tablet, creates two different graphic properties objects for drawing in blue and red, draws a rectangle using a blue pen and a line shape using a red pen, and then displays the resulting sketch on the display console.

```
ccUITablet tablet;
ccGraphicProps redProp(ccColor::redColor());
ccGraphicProps blueProp(ccColor::blueColor());
tablet.draw(ccRect(cc2Vect(10,10), cc2Vect(240, 100)),
  redProp);
tablet.draw(ccLineSeg(cc2Vect(100,100), cc2Vect(101,100)),
  blueProp);
display->drawSketch(tablet.sketch(),
  ccDisplay::eImageCoords);
```

Drawing Shapes

CVL provides a rich set of graphics shapes which are defined in *shapes.h*. To draw a shape you call the **ccUITablet::draw()** function which has overloads for all of the shapes. For example, the following code draws a circle:

```
// Draw circle using ccUITablet::draw()
ccGraphicProps redProp(ccColor::redColor());
tablet.draw(ccCircle(cc2Vect(100,100), 30.0), redProp);
```

The following table lists the shapes that **ccUITablet::draw()** can draw into a sketch.

Shapes drawn by ccUITablet::draw()

cc2Wireframe	**ccGenAnnulus**	**ccPoint**
ccAffineRectangle	**ccGenPoly**	**ccPointSet**

Table 44. Shapes drawn by ccUITablet::draw()

Shapes drawn by ccUITablet::draw()		
ccCircle	**ccGenRect**	**ccPolyline**
ccCoordAxes	**ccGraphic**	**ccRLEBuffer**
ccCross	**ccLine**	**ccRect**
ccEllipse2	**ccLineSeg**	**ccCvlString**
ccEllipseAnnulusSection	**ccPelBuffer**	**ccUISketch**
ccEllipseArc2		

Table 44. Shapes drawn by ccUITablet::draw()

Specifying the Drawing Layer

All of the drawing functions take an optional argument that specifies whether the graphics will be drawn in the plane that is used for the image, called the image layer, **ccUITablet::eImageLayer**, or in a separate plane for graphics, called the overlay layer, **ccUITablet::eOverlayLayer**. The advantage to drawing graphics in a separate layer is that you can update the image layer without having to redraw the graphics. This makes updating of the image layer faster.

For example, the first two lines of the following code specify that a red circle be drawn in the image layer. The second two lines specify that the same circle be drawn in the graphics overlay layer.

```
ccGraphicProps redProp(ccColor::redColor());
tablet.draw(ccCircle(cc2Vect(100,100), 30.0), redProp,
                                    ccUITablet::eImageLayer);

ccGraphicProps redProp(ccColor::redColor());
tablet.draw(ccCircle(cc2Vect(100,100), 30.0), redProp,
                                   ccUITablet::eOverlayLayer);
```

By default, graphics are drawn in the image layer.

If you draw into an overlay layer on a platform that does not support it, you will not see your graphics, and they will be ignored. When you want to ensure that your drawing routines work in any display environment, draw only in the image plane, or use **ccUITablet::overlaySupported()** to determine whether your drawing environment supports overlay layers. For additional information about overlay layers see *Customizing Image Display Environments* on page 220.

Drawing the Sketch

After you have drawn on the tablet, you get its sketch (the graphic contents of the tablet) and pass it to the display's **drawSketch()** function, specifying which coordinate system the sketch will be drawn in, as shown in the following example:

```
display->drawSketch(tablet.sketch(),
  ccDisplay::eDisplayCoords);
display->drawSketch(tablet.sketch(),
  ccDisplay::eImageCoords);
display->drawSketch(tablet.sketch(),
  ccDisplay::eClientCoords);
```

When drawing sketches, there are three important things to be aware of. The first is that the display maintains three different planes for each of the three coordinate systems (display coordinates, image coordinates, and client coordinates). The second is that you can reuse the same sketch in any of the coordinate systems. The third is that each call to **drawSketch()** appends the given graphics to the given plane. To erase any graphics already being displayed, call the **eraseSketch()** function.

In the preceding code, the sketch is first drawn in the display coordinate system. These coordinates are relative to the display console window. Anything you draw in this coordinate system remains in the same place in the window regardless of how you move or zoom the image in the display console.

The second time, the program draws the sketch in the image coordinate system. This coordinate system is relative to the pel buffer's offset. (See *Coordinate Systems* on page 231 to learn more about image coordinates.)

The third time, the program draws the sketch in the client coordinate system. By default, the client coordinate system and the image coordinate system are identical. But since the example program created a transformation object in which one client unit corresponds to 40 pixels, the sketch is drawn in a different place.

```
cc2Xform xform(ccVector<2>(0,0), ccRadian(0), ccRadian(0),
  40.0, 40.0);
pb.imageFromClientXform(xform);
```

To learn more about client coordinates, see *Understanding Client Coordinates* on page 236.

Erasing the Sketch

To erase a sketch, use the **eraseSketch()** function. For example, to erase the sketch in the client coordinates, write:

```
display->eraseSketch(ccDisplay::eClientCoords);
```

This removes the graphics from the display and refreshes the screen.

Showing Vertices on a Generalized Polygon

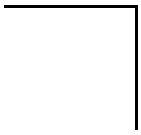
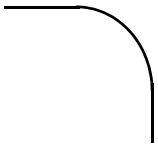
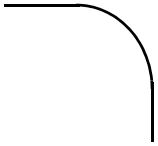
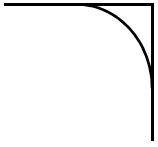
The **ccGraphicProps::showVertex()** method shows or hides the vertices for a generalized polygon (**ccGenPoly**) shape:

```
void showVertex(bool showVertex);
```

Each vertex of a **ccGenPoly** has a corner rounding value, interpreted as follows:

- A corner rounding value of 0 (the default) causes the corner to be drawn as a sharp intersection of the two segments.
- A positive corner rounding value causes the corner to be drawn as the radius of a circular arc, or *fillet*, which smoothly blends one segment into the other.
- A negative corner rounding value tells the pattern matching tools to ignore what would be the circular arc, or *fillet*, if the value were positive.

The **ccGraphicProps::showVertex()** method controls only how vertices with negative corner rounding values are drawn. It is ignored when the rounding value is 0 or positive. Interactions between the corner rounding value of a **ccGenPoly** and **showVertex()** produce the vertex renderings shown in the following table:

ccGenPoly Corner Rounding Radius	ccGraphicProps:: showVertex()	Rendering
0 (default)	N/A	
Positive	N/A	
Negative	false (default)	
	true	

Displaying and Using Interactive Graphics

In addition to static graphics, CVL classes allow you to display interactive graphics. Interactive graphics are graphics that can be manipulated with a mouse. For example, you click on an interactive graphic to select it. It then changes color and handles appear that you can use to resize, move, and rotate the graphic.

Interactive Graphics Overview

Interactive graphics use the same basic shapes defined in *shapes.h* and listed in Table 44 on page 263. An interactive graphics class is defined for each shape in *uishapes.h* and these classes are listed in Table 45 on page 270. For each interactive graphics object you create, you associate it with a corresponding graphic shape which you can then display and manipulate.

The basic steps for using interactive graphics are:

1. Create the interactive graphics shapes.
2. Set the drawing layer if desired.
3. Add the interactive graphics to the display.
4. Allow the user to interact with the graphics.
5. Retrieve information on the final state of the graphics.
6. Remove the graphics from the display.

The following code creates an interactive rectangle shape, sets its properties, and then displays it on the display console in client coordinates.

```
ccUIRectangle *uiRect = new ccUIRectangle;
uiRect->rect(ccRect(cc2Vect(10,10), cc2Vect(240, 100)));
uiRect->color(ccColor::red);
uiRect->condVisible(true);
uiRect->drawLayer(ccUITablet::eOverlayLayer);
display->addShape(uiRect, ccDisplayConsole::eClientCoords);
```

Drawing Interactive Graphics in the Overlay Layer

All interactive shapes are drawn to the image layer by default. If you want to draw the shape on a different layer (for example, the overlay layer), you can set the drawing layer using **ccUIObject::drawLayer(lyr)**. Drawing interactive graphics in the overlay layer makes updating the image layer faster since the graphics do not have to be redrawn.

Cognex recommends changing the drawing layer before adding the shape to the display. Changing the drawing layer once a shape has been added to the display will result in lower performance when the graphics are re-rendered.

You can use **ccUITablet::overlaySupported()** to determine whether your drawing environment supports the overlay layer.

Note If you attempt to use the overlay layer and it is not enabled, no graphics will be displayed. Keep in mind that not all platforms enable the overlay plane by default.

For additional information about overlay layers see *Customizing Image Display Environments* on page 220.

Interactive Graphics Applications

Although interactive graphics classes allow you to display and manipulate simple geometric shapes like the rectangle example above, these tools are designed and intended for much more complex use. Typical applications involve many graphic shapes, sometimes hundreds, displayed at once to represent a real world problem. The complete problem representation may be made up from the graphic shapes supplied by Cognex, such as rectangles, circles, and lines, or may include custom graphic shapes provided by your own class derivations.

The real power of the interactive graphics framework is that these graphics shapes can be programmed to be related in complex ways. Functions you can do with your mouse can also be done in your program. For example, one set of graphics may represent a subsection of your application that you wish to manipulate as a single unit. Here, you can program the objects so that if you click on any part of the subsystem with your mouse, the entire subsystem is selected and you can then move it, resize it, or rotate it as a unit even though it is made up of many individual graphic shapes.

Performance Considerations

All interactive shapes are drawn to the image layer by default. If you want to draw the shape on a different layer (for example, the overlay layer), you can set the drawing layer using **ccUIObject::drawLayer(lyr)**. Cognex recommends changing the drawing layer before adding the shape to the display. Changing the drawing layer once a shape has been added to the display will result in lower performance when the graphics are re-rendered.

When assigning a parent to interactive shapes using **ccUIShapes::parent(newParent)**, all children must be on the same drawing layer as the parent. Cognex recommends setting the parent of an interactive shape before adding the shape to the display. Changing a shape's parent once the shape has been added to the display will result in lower performance when the graphics are re-rendered.

For optimal rendering performance, do not change the drawing layer or parent of an interactive shape after adding the shape to the display.

Creating Interactive Graphics

To create an interactive graphic shape you can manipulate, you first instantiate the interactive graphic class for the desired shape. These interactive classes have the same names as the static classes, except they begin with the prefix **ccUI** and provide the functions that respond to mouse clicks and drags.

Next, you associate a corresponding static shape with the interactive graphics class. Each interactive graphics class has a setter member function that links it with a particular static shape. For example, the **ccUICircle** class has a **circle()** function that associates a **ccCircle** object with it.

Table 45 table lists the interactive graphics classes and the corresponding static classes.

Interactive graphics classes	Corresponding static classes
ccUIAffineRect	**ccAffineRectangle**
ccUICircle	**ccCircle**
ccUICoordAxes	**ccCoordAxes**
ccUIEllipse	**ccEllipse2**
ccUIEllipseAnnulusSection	**ccEllipseAnnulusSection**
ccUIGenAnnulus	**ccGenAnnulus**
ccUIGenRect	**ccGenRect**
ccUIIcon	**ccRLEBuffer**
ccUILabel	**ccCvlString**
ccUILine	**ccLine**
ccUILineSeg	**ccLineSeg**
ccUIPointIcon	None (Built-in cross-hair icon)
ccUIPointSet	**ccPointSet**
ccUIRLEBuffer	**ccRLEBuffer**
ccUIRectangle	**ccRect**

Table 45. Interactive graphics classes and corresponding static classes

All of these interactive graphics classes are derived from **ccUIShapes** and **ccUIObject**. Figure 49 shows the derivation hierarchy of the UI shapes classes including the relationship to **ccDisplay** windows where interactive graphics are displayed.

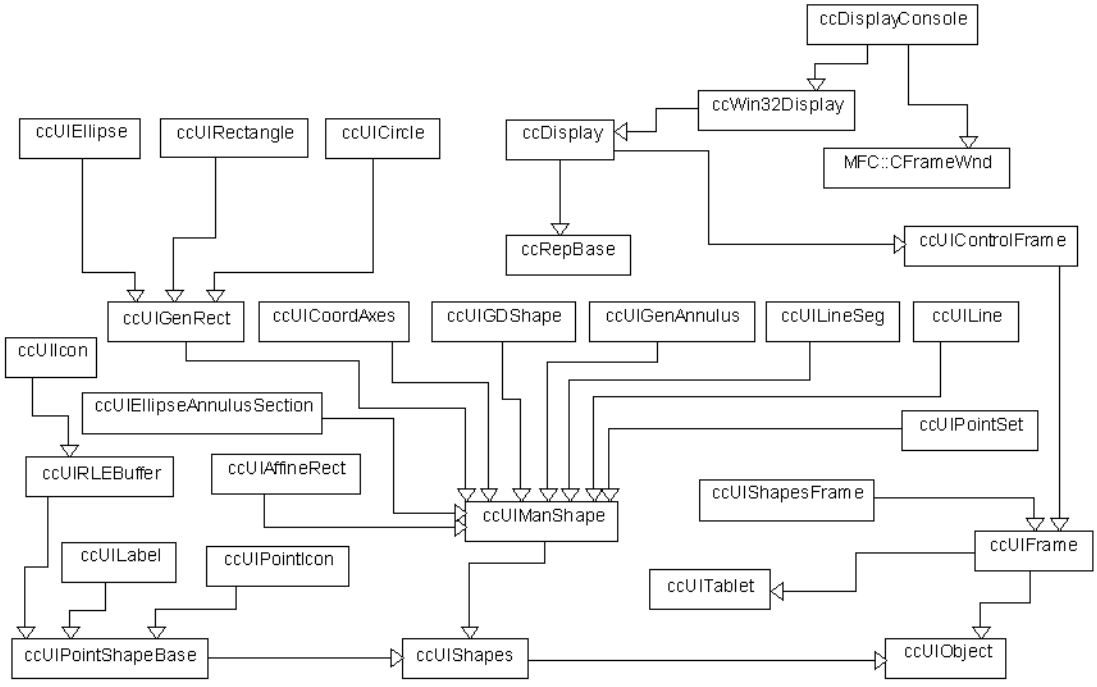


Figure 49. Interactive graphics class derivation hierarchy

Interactive Graphics Object Relationships

Interactive graphic objects relate to one another as parents, children, and siblings. Every displayed object has a parent object while other relationships are optional. Graphics objects that have the same parent are siblings.

When you add an object to a display window its parent is automatically set to one of the coordinate frame objects internal to the display. In your applications, there is no need to access these internal coordinate frame parent objects.

Functions provided in the **ccUIObject** and **ccUIShapes** classes allow you to obtain information about these relationships. These functions are described in Table 46.

Function	Description
parent()	Set/get the parent object of this shape.
frontSib()	Returns this object's parent's last added child. If this object was the last added, it returns itself.
backSib()	Returns this object's parent's first added child. If this object was the first added, it returns itself.
closerSib()	Returns the sibling added to the parent object just after this object. If there is none, it returns 0.
fartherSib()	Returns the sibling added to the parent object just before this object. If there is none, it returns 0.
frontKid()	Returns the child object of this shape that was last added. If there is none, it returns 0.
backKid()	Returns the child object of this shape that was first added. If there is none, it returns 0.
numKids()	Returns the number of child objects owned by this shape.

Table 46. Interactive graphics shape relationship functions

When your application requires a hierarchy of parent and child objects build the hierarchy when you construct the shapes by naming the parent objects in the constructors. The top-level object in each hierarchy must be default constructed with the parent set to NULL. You then add only the top-level objects to the display. For example, call **display->addShape(...)** only on the top-level parent. All child objects down the chain are then added automatically.

For child objects to display properly, child objects must be on the same layer as the parent (either image layer or overlay layer). You must assign the parent layer and each child layer individually. Note that for most applications, graphics should be in the overlay layer. This will save execution time since graphics will not have to be redrawn each time the image changes.

To properly create a parent/child relationship between two **ccUIShapes**, do the following:

1. Create the parent first on the appropriate layer.
2. Create the child on the same layer using the parent object as the parameter to the child constructor.

3. Call **display->addShape(...)** on the parent to add the parent and the child to the display.

Do not call **addShape(...)** on the child as this destroys parent/child relationship.

The following code demonstrates the above procedure:

```
ccUIRectangle *uiRect = new ccUIRectangle;
uiRect->rect(ccRect(cc2Vect(0,0), cc2Vect(150,150)));
uiRect->color(ccColor::white);
uiRect->selectColor(ccColor::white);
uiRect->drawLayer(ccUITablet::eOverlayLayer);
uiRect->condVisible(true);

ccUILabel *uiLabel = new ccUILabel(uiRect);
uiLabel->label(ccCvlString("Example Label"));
uiLabel->pos(ccPoint(75,75));
uiLabel->color(ccColor::white);
uiLabel->selectColor(ccColor::white);
uiLabel->borderWidth(1);
uiLabel->borderColor(ccColor::white);
uiLabel->drawLayer(ccUITablet::eOverlayLayer);
uiLabel->condVisible(true);

console->addShape(uiRect, ccDisplayConsole::eClientCoords);
```

By default, the child object's upper left-hand corner is displayed at the center of the parent object. Note in this example that we have specified that the label be offset from the parent's center so that it appears at the rectangle's lower right-hand corner. See Figure 50.

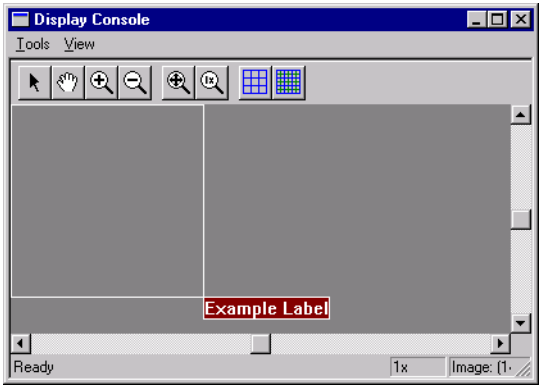


Figure 50. Parent/child display example

Printing a Display Hierarchy

The following code can be used to print an entire display hierarchy.

```
void dumpObjects(ccUIObject* obj, int level = 0)
{
cogOut << ccCvlString(level*3, cmT(' ')) << typeid(*obj).name()
       << std::endl;
for(ccUIObject* p = obj->backKid(); p; p = p->closerSib())
  dumpObjects(p, level+1);
}

dumpObjects(display->root());
```

More About Parent/Child Relationships

Interactive graphics classes are of two varieties depending on the class derivation hierarchy. One group is derived from **ccUIManShape** and the other from **ccUIPointShapeBase**. See Figure 51.

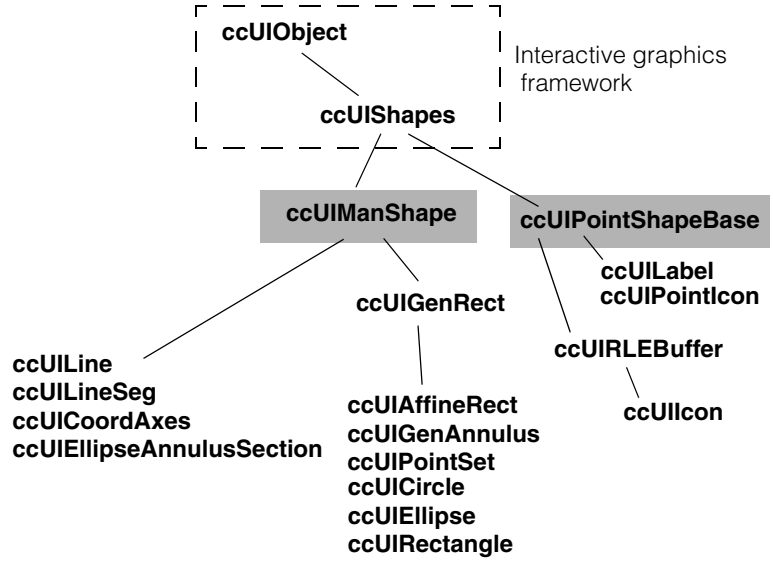


Figure 51. UI shapes derivation hierarchy

There are a few special cases where you must be aware of whether your shape was derived from **ccUIManShape** or **ccUIPointShapeBase**. These cases are discussed below.

1. An important restriction does not allow parent/child relationships where both parent and child objects derive from **ccUIManShape**. (See Table 47 below). For example, a **ccUIRectangle** cannot have a **ccUICircle** as a child. You can construct such relationships, but they will not function properly.

2. When you work with UI shapes you need to be aware of the shape's coordinate frame reference. The frame reference is dependent on the shape class derivation hierarchy and the shape's display parent, in the following ways:

   a. Shapes derived from **ccUIManShape** specify their coordinates relative to the tablet where they are displayed, where the 0,0 point is the upper left-hand corner of the tablet display space.

   b. The coordinate reference for shapes derived from **ccUIPointShapeBase** is dependent on the shape's display parent class: If the parent derives from **ccUIManShape**, the child shape specifies coordinates relative to the parent's origin. For example, if you specify the child shape's location as **pos(0,0)**, it will be located at the center of its parent.

   If the parent derives from **ccUIPointShapeBase**, and the child shape also derives from **ccUIPointShapeBase**, the child shape specifies coordinates relative to the tablet as in (a) above.

   Note: The case for a **ccUIPointShapeBase** parent and a **ccUIManShape** child shape is not supported. See the following table.

Parent shape derived from	Child shape derived from	
	ccUIManShape	**ccUIPointShapeBase**
ccUIManShape	Not supported	Coordinates relative to parent's origin
ccUIPointShapeBase	Not supported	Coordinates relative to tablet where displayed

Table 47. Parent/child shape relationships

When shapes derived from **ccUIPointShapeBase** are added directly into a display rather than as a child of another shape, the parent automatically becomes one of the coordinate frame objects internal to the display which is the same as the tablet coordinate frame. So, for shapes added directly to a display there is no difference between **ccUIManShape** derived shapes and **ccUIPointShapeBase** derived shapes. It is only shapes that have another shape as a parent that you need to be aware of these differences.

3. When you drag a parent shape with your mouse, its child shapes will move with it. The child shape does not have to be selected. Note that only the parent shape is animated during the drag. Child shapes are not animated and just appear repositioned when the drag is completed. It is also important to note that while you can drag a parent/child pair by selecting and dragging the parent, you cannot do the same thing by selecting and dragging the child. If you select and drag the child shape, the parent is not affected.

   Note that this parent/child interaction does not extend to graphical manipulations. For example, if you resize the parent, the child is not resized. These kinds of interactions are discussed in *Working With Multiple Shapes* on page 289.

Interactive Graphics States

Each **ccUIObject** has three state indicators that track its status in a **ccDisplay** window. To manipulate a graphic object it must be *visible*, *enabled*, and *selected*. These three states are defined in Table 48.

State	Meaning
visibleState	The object is displayed on the screen and is visible unless covered by another object. The following conditions specify this state: **visible()** = true **condVisible()** = true Parent **visible()** = true
enabledState	The object is enabled and responds to mouse clicks. The following conditions specify this state: **enabled()** = true **condVisible()** = true **condEnabled()** = true Parent **enabled()** = true
selectedState	The object is displayed using its *selected* color, showing handles (if any). The following conditions specify this state: **selected()** = true **condVisible()** = true **condEnabled()** = true **condSelected()** = true Parent **selected()** = true

Table 48. Interactive graphics states

See the **ccUIObject** reference page for more information about interactive graphics states and the associated member functions.

Interactive Graphics Attributes

After creating an interactive graphics object, you can change its attributes from the default values if you desire. Some of these attributes are defined in **ccUIShapes**, and some are defined in **ccUIObject**, both of which are base classes for all interactive graphics classes. (See Figure 49 on page 271). Table 49 describes these attributes and also includes some attribute-associated getter functions. See the **ccUIObject** and **ccUIShapes** reference pages for more detailed information about each attribute.

Attribute	Set	Get	Description
props()	x	x	Graphics pen color, width, style, and other pen attributes.
clickable()	x	x	True if the graphic responds to mouse clicks.
draggable()	x	x	True if the graphic can be dragged.
dragging()		x	True if the object is currently being dragged.
dontMove()	x	x	If true, do not move the object to the front when selected. Used to maintain a fixed screen order independent of mouse clicks.
keepSel()	x	x	If true, the object cannot be deselected. Set true when grouping UI shapes.
dwell()	x	x	If true, the system generates successive **click()** events while the left mouse button is held down.
multiSelectable()	x	x	If true, this object can be selected along with one or more other objects.
multiSelected()		x	Returns true if the specified object's **multiSelectable()** flag is true.
multiDraggable()		x	Returns true if this object and all of its multi-selected siblings can be dragged.
userSelect()	x		Specifies how to implement click-select behavior in combination with the shift key. Use this to select an object with your program. See the **ccUIObject::userSelect()** reference.

Table 49. Interactive graphics attributes

Attribute	Set	Get	Description
rightButtonMode()	x	x	Specifies choices for actions taken when the right mouse button is pressed down.
autoDelete()	x	x	If true, this object will be automatically deleted. (Overrides **new()**). By default, any UI shape added to a **ccDisplay** will be automatically deleted.
mark()	x	x	If true, an object is marked. You can use this flag in your application for identification.
selectColor()	x	x	Specifies the color of all selected objects in the window.
deselColor()		x	Specifies the object's color when deselected. Hard coded to cyan.
dragColor()	x	x	Specifies the object's color when dragged.
curSelColor()		x	Returns **selectColor()** if the object is selected. Returns **deselColor()** if the object is deselected.
lightColor()		x	Use these functions instead of the color so that if in the future the header file changes, you will not have to change your application code.
shadowColor()		x	
testColor()		x	
faceColor()		x	
drawLayer()	x	x	A tablet layer where this object will be drawn. Derived classes actually choose the layer. The default is *ccUITablet::eImageLayer*.

Table 49. Interactive graphics attributes

Drawing Interactive Graphics

To draw a graphic, you specify its drawing layer, if necessary, and then call **ccDisplay::addShape()** to add it to the display and to specify its coordinate system. In the example on page 268, this is how the rectangle is drawn in the display:

```
uiRect->drawLayer(ccUITablet::eImageLayer);
display->addShape(uiRect, ccDisplayConsole::eClientCoords);
```

Removing the graphic from the display does not destroy it. However, if you delete the display, all of the graphics associated with the display are deleted as well.

Note that adding or removing graphics in a display will not automatically refresh and redraw the display to show your changes. You must do this yourself in your program. The following are ways to accomplish this:

1. To redraw all layers.

```
display->fullDraw();
```

2. To redraw a specific layer; in this case the overlay layer.

```
display->fullDraw(ccUITablet::eOverlayLayer);
```

3. Redraws all layers.

```
display->root()->globalUpdate();
```

4. Redraws all layers.

```
disableDrawing();
...
...  // Make your changes
...
enableDrawing(true);
```

   Use this technique when making many changes so that you need refresh only once.

Selecting and Deselecting Interactive Graphics

To manipulate a graphic object you must first select it by left-clicking your mouse pointer on the graphic outline. Your mouse pointer changes to a cross when you are close enough to select it. Once selected the graphic changes to the color set by the last call to **selectColor()**. The default color is green. Selected graphics are displayed with handles you use to manipulate the object. Most graphics have handles for resizing and rotating the shape.

To move a selected graphic, left-click on the graphic outline holding the mouse button down, and drag the graphic to the new location.

You deselect a graphic by selecting another graphic or by left-clicking your mouse in free space anywhere in the window. Deselected graphics return to their original color which you can set by calling **props()**. The default color is cyan.

Using Program Control

You can perform these same functions under program control by calling member functions of the graphic of interest. To select or deselect a graphic, use **uishape->condSelected()**, passing *true* to select the graphic and *false* to deselect it. When the graphic is selected, it changes color as it does when you use the mouse. When a graphic is deselected, it returns to cyan. The deselect color cannot be changed.

To select an interactive graphic, use the following code:

```
uishape->condSelected(true);
```

To deselect an interactive graphic, use:

```
uishape->condSelected(false);
```

Do not use **uishape->select()** or **uishape->deselect()** to select or deselect graphics. These methods are intended to be overridden to respond to state changes, for example when you want to take some specific action in response to a graphic being selected or deselected.

Getting Information About Interactive Graphics

Table 50 describes functions in **ccUIObject** and **ccUIShapes** that allow you to retrieve information about graphic objects. See the **ccUIObject** and **ccUIShapes** reference pages for more detailed information about these member functions.

Function	Set	Get	Description
pos()	x	x	The center of this graphic relative to its parent, in the coordinate system in which it is drawn.
absPos()		x	Get the absolute location of this graphic in tablet coordinates.
whoIsTouched()		x	Returns a pointer to the front-most visible object touched by the point (*p*).
opNew()		x	Returns true if this object was created with **new()** and therefore requires a **delete()** to destroy it. Note that **ccDisplay** automatically deletes UI shapes for you unless you call **autoDelete(***false***)**.
root()		x	Get the root object of this graphic, if there is one.

Table 50. Information retrieval functions

Function	Set	Get	Description
rootMutex()		x	Get the root mutex of this object, if one exists. Otherwise, return the orphan mutex. Used for thread synchronization.
orphanMutex()		x	Get an orphan mutex, a mutex used by all objects without a parent.
uiMutex()		x	Get a global UI object mutex.
key()		x	Get a key that uniquely identifies this object.
isValid()		x	Returns true if the object specified by the *address* and *key* exists and has the specified *root object* as a parent.
getGraphicProps()		x	Get the graphic properties of this object.
isTouched()		x	Always returns false. Derived classes can override and return true if the specified position touches its UI shape.

Table 50. Information retrieval functions

Saving a Manipulated Graphic

You use interactive graphics to allow a user to easily manipulate and change the graphics using the display and a mouse. Once these changes are made, usually you need to save the changed graphic for use later in your application. The following code is an example of how you could save a modified rectangle graphic.

```
ccRect myRect;
myRect = uiRect->rect();
```

Since the rectangle was added to the display in client coordinates, the rectangle returned is also in client coordinates.

Removing Interactive Graphics

To remove a graphic from the display, you can hide it by making it invisible:

```
uiRect->condVisible(false);
```

Or you can remove it from the display altogether:

```
display->removeShape(uiRect);
```

Note that interactive graphics added to a **ccDisplay** window will be automatically deleted when the window goes out of scope.

See *Drawing Interactive Graphics* on page 278 for ways to refresh the display after removing graphics.

Managing Interactive Events

Users interact with graphics displayed in a **ccDisplay** window using the mouse and keyboard. Actions such as mouse moves, clicks, and keyboard key strokes are called events. To manage these events and match the events to the intended graphic objects Cognex provides an event processor which reports mouse and keyboard activity without regard to meaning, and the UI shapes classes, which are interested in higher-level concepts like select and drag rather than button-down and mouse-move. The event manager classes include:

ccUIEventProcessor
ccKeyboardEvent
ccMouseEvent

The primary order-imposing event manager rules are *one button at a time* and *one UI shapes object at a time*. **ccUIEventProcessor::owner()** specifies the one **ccUIObject** and **ccUIEventProcessor::button()** specifies the one button. The owner and the button are set whenever a button-down event occurs. The front-most visible **ccUIObject** that is touched by the mouse, as determined by **ccUIObject::root()->whoIsTouched()** becomes the owner. All subsequent mouse events except idle moves (if the mouse moves and no button is down) are sent to that object until a new button-down occurs. Button events on inactive buttons are ignored.

Mouse Events

The UI Framework posts a mouse event whenever a button changes state and whenever the mouse moves. If the mouse moves and no button is down, it is called an *idle move*. You can also ask the event system to post periodic *dwell* events when a button is held down in one spot for a while.

Touch Zones

The *touch zone* (TZ) of a UI Shape is where you must click to select the object. When the mouse pointer is near a touch zone, the pointer changes shape and the object can be selected. If the object is invisible or behind another object, its touch zone is null.

The *family touch zone* (FTZ) of a UI Shape is the union of its touch zone and its children's family touch zones. Often an object's FTZ is the same as its TZ if it has no children.

Mouse Motion

Enabled UI shapes are notified whenever the mouse enters, leaves, or moves within its FTZ. To receive and process such a notification, an object overrides the protected **ccUIObject** virtuals **mouseEnter_()**, **mouseLeave_()**, and **mouseMove_(const cclPair&)**. For Cognex UI shapes **mouseEnter_()** and **mouseLeave_()** are overridden in the class **ccUIShapes**. If an enabled object is entered and becomes disabled before the mouse leaves, **mouseLeave_()** will be called on the next mouse event. We guarantee that **mouseEnter_()** and **mouseLeave_()** are balanced unless an object is deleted.

When the mouse is clicked on an object and that object becomes the mouse owner, the object's FTZ becomes the entire screen as long as the button is held down. The mouse owner in this state truly owns the mouse. It is notified of all mouse moves and no other object will get a **mouseEnter_()** notification. When the button is released, appropriate **mouseLeave_()** and **mouseEnter_()** notifications are given to reflect the new position of the mouse on the screen.

It is important to note that an object's FTZ includes the touch zones of all of its children. A parent's **mouseEnter_()** will always be called before its children's, and **mouseLeave_()** after its children's. **mouseMove_()** notification will always be given to the *youngest* object first, then to ancestors all the way up to **ccUIObject::root()**.

Mouse Clicking

If an object is **clickable()** and **enabled()** and the user clicks on it with the left mouse button, the object can receive and process notification by overriding the protected virtuals **mouseDown_()**, **mouseUp_()**, and **click_()**. Such an object will always receive **mouseDown_()** followed by zero or more **click_()**s followed by **mouseUp_()**. Typically mouse-down and mouse-up are used to animate buttons and click causes some response. Clients are not told what the user has done to cause these sequences of events, we simply guarantee the above sequence occurs.

Mouse-down notification is given for **clickable()** objects for either of the following:

1. The user pushes down on the left button while the mouse is within the object's TZ.
2. Condition 1 occurs followed by moving the mouse out of the TZ, followed by zero or more instances of moving the mouse back into the TZ and out again, followed by moving the mouse into the TZ (all with the button held down).

Mouse-up notification is given after any mouse-down condition immediately followed by any of the following:

1. Moving the mouse out of the TZ with the button still held down.
2. Releasing the left button.

3. For an object that is also **draggable()**, mouse-down condition 1 followed by any motion of the mouse. Note that releasing the button while the mouse is not within the TZ does not result in a mouse-up, because it does not immediately follow any mouse-down condition. (For example, mouse-up has already been given).

Click notification is given after any mouse-down condition immediately followed by either:

1. A dwell event, which results if dwelling is enabled on the object and the user holds the mouse button down for a certain interval.
2. Releasing the left button before any dwell events have happened. Note that click is always given before mouse-up.

Double Click

Objects can receive double-click (left mouse button) notification by overriding the virtual **dblClick_()**. Objects receive this notification if they are enabled regardless of whether they are **clickable()** or **draggable()**. If the object is **clickable()**, a double-click will cause the following events in this order: mouse-down, click, mouse-up, double-click.

Mouse Dragging

If an object is draggable, it can receive notification that dragging has begun or ended, or that during a drag the current position has changed, by overriding the protected virtuals **dragStart_(cclPair)**, **dragStop_(cclPair, cclPair)**, and **dragAnimate_(cclPair, cclPair)**. After dragging **dragStop_()** is used to update the object's state based on the final position of the drag. **dragStart_()** is rarely used.

Dragging begins when the left button is pressed down within the TZ of an **enabled()** and **draggable()** object. **dragAnimate_()** is called whenever the mouse moves while the button is still held down, and **dragStop_()** is called when the button is released. Note that, as described above, **mouseMove_()** will also be called during dragging.

One exception to these rules is that if an object is both **clickable()** and **draggable()**, pressing the left button down in the TZ does a mouse-down and not a **dragStart_()**. The choice between dragging and clicking is made by observing whether the next event is a move or an up-click. If the mouse moves, mouse-up is called to cancel the click and **dragStart_()**, followed immediately by **dragAnimate_()**, is called. If the button is released before the mouse moves, click followed by mouse-up is called.

Disabling Objects

If you disable an object, it will receive no events except possibly **mouseLeave_()** as described in *Mouse Motion* on page 283. If you disable after a **mouseDown_()**, you won't get a **mouseUp_()**. If you disable in the middle of dragging, you won't get a **dragStop_()**. This can be dangerous. Disabling after **click_()**, **dblClick_()**, **mouseUp_()**, and **dragStop_()** is safe.

Middle, Right Buttons

All of the GUI rules involving buttons are defined for the left mouse button only. Notification of activity on the other buttons can be received by overriding the protected virtuals **mouseMiddle()** and **mouseRight_()**. Note that the *one object, one button at a time* rules apply to all buttons. These notifications are given independent of **enabled()**.

Keyboard Events

Keyboard events are sent directly to the **ccUIObject** which is capturing mouse events. **ccUIObject**s interested in receiving keyboard events must override their **keyboard_()** member function.

Customizing Interactive Graphics Environments

Interactive graphics classes are designed to allow you to easily add functionality to interactive graphics applications. For example, if you find you would like to take some specific action when a **ccUIRectangle** object is selected, you can write your own code to accomplish this by doing the following:

1. Derive your own interactive rectangle class from **ccUIRectangle**. For example, derive **myUIRectangle** from **ccUIRectangle**.
2. In **myUIRectangle** override the protected function **select()** with your own routine that takes the specific action you desire.
3. In your application use the new **myUIRectangle** class instead of **ccUIRectangle**. Whenever a rectangle is selected, your new routine will execute.

If you wish to create a new graphic of your own, you will need to derive from one the of the UI base classes. See Figure 49 on page 271 to get an idea of how your new class would best fit into the hierarchy.

Customizing With Overrides

The **ccUIObject** and **ccUIShapes** interactive graphics framework classes provide a set of protected virtual functions that are place holders for overrides in derived classes. These place holders are called by the framework but perform no operations unless

overridden. Cognex adds some overrides in the UI shapes classes we provide. If you wish to develop your own custom interactive graphics classes or to modify the Cognex classes, you can provide overrides also in your derived classes. Table 51 summarizes these protected virtual functions and includes information about where Cognex has provided overrides.

Protected virtual function	Where overridden	Description
show()	ccUIShapes	Called when the **ccUIObject** becomes visible.
hide()	ccUIShapes	Called when the **ccUIObject** becomes invisible.
enable()		Called when the **ccUIObject** becomes enabled.
disable()		Called when the **ccUIObject** becomes disabled.
select()	ccUIShapes ccUIManShape	Called when the **ccUIObject** becomes selected.
deselect()	ccUIShapes ccUIManShape	Called when the **ccUIObject** becomes deselected.
mouseDown_()		Called when the left mouse button is pressed down while pointing to this **ccUIObject**. The **ccUIObject** must be **clickable()**.
mouseUp_()		Called after mouseDown when; 1) the left mouse button is released or, 2) the mouse pointer is moved off the **ccUIObject**.
mouseEnter_()	ccUIShapes	Called when the mouse pointer touches this **ccUIObject** when the left mouse button is down.
idleMouseEnter_()	ccUIShapes	Called when the mouse pointer touches this **ccUIObject** when the left mouse button is up.

Table 51. Protected virtual functions summary

Protected virtual function	Where overridden	Description
mouseLeave_()	ccUIShapes	Called when the mouse pointer is moved off this **ccUIObject** or, also called if an enabled object is entered and becomes disabled before the mouse leaves.
mouseMove_()		Called repeatedly when you move the mouse while it is touching this object. Each call provides the current mouse location.
click_()		Called following a mouseDown/mouseUP sequence. Also, following a mouseDown, if **dwell()** = true, **click()** is called repeatedly until mouseUP.
dblClick_()		Called when you double click the left mouse button while pointing to this object.
dragStart_()		Called when you begin to drag a **ccUIObject** with the mouse.
dragStop_()	ccUIShapes ccUIGenAnnulus ccUILine ccUILineSeg	Called when you stop dragging a **ccUIObject** with the mouse.
dragAnimate_()	ccUIShapes	Called repeatedly during dragging. The derived function will display an outline of the dragged graphic on each call to show the user the current graphic position.
mouseMiddle_()		Called when the middle mouse button is pressed down while the mouse is touching this object.

Table 51. Protected virtual functions summary

Protected virtual function	Where overridden	Description
mouseRight_()		Called when the right mouse button is pressed down, **rightButtonMode()** = *eClientRB,* and the mouse is touching this object. Your derived class must override **mouseRight_()** and implement some action of your choice.
front_()	ccUIShapes	Called when an object is moved to the front. For example, when it is selected.
back_()	ccUIShapes	Called when an object is moved to the back.
keyboard_()		Called when a keyboard event occurs and the mouse is touching this object.
draw_()	ccUIAffineRect ccUICoordAxes ccUIEllipseAnnulusSection ccUIGenAnnulus ccUIGenRect ccUILabel ccUILine ccUILineSeg ccUIPointIcon ccUIPointSet ccUIRLEBuffer ccUIRectangle	Call **ccUIShapes::draw()** to draw the UI shape into a specific tablet.
move_()		Call **ccUIShapes::move()** to move the object to a new location.

Table 51. Protected virtual functions summary

Protected virtual function	Where overridden	Description
pos_()	ccUIAffineRect ccUICoordAxes ccUIEllipseAnnulusSection ccUIGenAnnulus ccUIGenRect ccUILabel ccUILine ccUILineSeg ccUIPointSet ccUIPointShapeBase	Call **ccUIShapes::pos()** to get/set the objects position relative to its parent.
redim_()		Called when any of the UI shape's dimensions change. Has no effect unless overridden.

Table 51. Protected virtual functions summary

Example

The following scenario is typical for these protected member functions that you can override.

1. A user moves the mouse causing the UI Framework to call **mouseMove()** for the appropriate object.
2. **mouseMove()** performs some boiler plate bookkeeping and then calls **mouseMove_()**.
3. If **mouseMove_()** is not overridden in a user-derived class, **mouseMove_()** does nothing and returns. No additional action is taken.

   If **mouseMove_()** is overridden in a user-derived class, the override function is executed.

Working With Multiple Shapes

When graphic shapes have a parent/child relationship you can drag both the parent object and the child object by moving just the parent. This is discussed in *Working With Multiple Shapes* on page 289. In this section we discuss working with shapes that are not related.

Most applications involve multiple graphic shapes that are related. For example, when several shapes are all part of a single component, moving any shape will move all related shapes so that their distance and orientation relative to one another stays

constant. For dragging graphics around, the UI framework provides a simple way to handle this. You set all of the related objects to **uishape->multiSelectable(true)**, and then select each of the graphics in the group. Then when you move any one, all selected objects move together.

To make this more convenient, it would be nice to accomplish this by selecting only one shape in the group. You can do this by modifying each shape class to be notified when it is selected. Once notified, you can provide code that selects all of the other group members so they all appear selected together and can all be dragged as one. To accomplish this you need to the following:

1. Derive you own graphic classes from the Cognex supplied classes as described in *Customizing Interactive Graphics Environments* on page 285.
2. In each new class override **select()** and add code to select the other shapes in the group. For example:

   ```
   uishape->condSelected(true); // For each other shape
   ```

   When any shape is selected, they will all be selected.
3. In each new class override **deselect()** and add code to deselect the other shapes in the group. For example:

   ```
   uishape->condSelected(false); // For each other shape
   ```

   When any shape is deselected, they will all be deselected.

Notes

When you override **select()** and **deselect()** you must call **select()** and **deselect()** in the base class first, and then add you override code.

Resizing and Rotating Multiple Graphics

In addition to moving a group of shapes as described in the previous section, it is also possible to resize and rotate a group of shapes although this is more complex. Use the following procedure which is similar to the procedure above.

1. Derive you own graphic classes from the Cognex supplied classes as described in *Customizing Interactive Graphics Environments* on page 285.
2. In each new class override **redim_()** and add code to redimension the current shape as well as the other shapes in the group, and update the display (call **update()** for each redimensioned shape).

Redimensioning other shapes is the hard part. First you must obtain a copy of the changed graphic from the interactive graphic class. For example:

```
changed_rectangle = uirect->rect();
```

Knowing how it changed implies you know what it was before the change. Once you know the change that has taken place, you will then need to calculate the affect it has on the associated shapes. You then need to update the dimensions and orientation in each associated shape and redisplay them.

Displaying Result Graphics

Graphical representation of vision tool operations is often essential to determine vision tool performance and results. CVL provides several classes that simplify the display of vision tool results.

Tools Supported

In order to display result graphics, the vision tool must support the graphic list classes defined in *glist.h*. The following vision tools support these classes:

- Blob
- Caliper
- CNLSearch
- PMAlign

Result Graphics Overview

Result graphics are static graphics from vision tool operations that are wrapped in **ccGraphic**-derived classes and collected into an iterable **ccGraphicList** list before they are displayed. Each vision tool that supports result graphics contains one or more **draw()** functions.

After a tool is run, a **ccGraphicList** can be built from the result set using the **draw()** functions. Building a result graphic list can be as simple as calling the **draw()** function once, or you can iterate through the result set, modify individual items, and call the **draw()** functions multiple times.

Result Graphics Code Sample

The following example shows how result graphics can be created for the CNLSearch tool.

```
#include <ch_cvl/cnlsrch.h>
#include <ch_cvl/pelfunc.h>
#include <ch_cvl/windisp.h>
#include <ch_cvl/gui.h>
```

```
int cfSampleMain(int, TCHAR** const)
{
  // Create an image to work with:
  // Two white rectangles on a black background.
  ...

  // Create a window for model training:
  // Enlarge rectangle by 20 pels on every side.
  ...

  // Create appropriate parameters, set the origin, and train
  // the model.
  ...

  // Create a resultSet and runtime parameters.
  // Then run the model on the entire image.
  ccCnlSearchResultSet results;
  ccCnlSearchRunParams
    runParams(ccCnlSearchDefs::eNormalizedCnlpas)

  ...

  model.run (searchImage, runParams, results);

  // display run-time image
  ccDisplayConsole console(ccIPair(300, 300), cmT("Image"));
  console.image(searchImage, false);
  console.fit();

  // display result graphics
  ccGraphicList graphics;
  results.draw(graphics);
  console.drawSketch(graphics.sketch(),
    ccDisplay::eClientCoords);

  cfWaitForContinue();  // pause

  // Change the color of the result graphics to red
  // and remove labels
  cmStd vector<ccGraphicPtrh>& items = graphics.items();
  cmStd vector<ccGraphicPtrh>::iterator iter = items.begin();
  while (iter != items.end())
  {
    if (dynamic_cast<ccGraphicText*>(iter->rep()))
    // if the item is a text label, remove it
      iter = items.erase(iter);
    else
```

```
    {
     // otherwise, change the color to red
      iter->rep()->color(ccColor::redColor());
    ++iter;
    }
  }

  // erase and display new graphics
  console.eraseSketch(ccDisplay::eClientCoords);
  console.drawSketch(graphics.sketch(),
    ccDisplay::eClientCoords);
  cfWaitForContinue();  // pause
  return 0;
}
```

Building Graphic Lists

After a tool has been run, a list of graphic items can be created from the result set. In the example,

```
ccGraphicList graphics;
results.draw(graphics);
```

creates an empty graphic list and passes it to the results set's **draw()** function, which populates the list with graphic items representing the tool results. The contents of the graphic list can then be drawn to a sketch and displayed as usual. For example,

```
console.drawSketch(graphics.sketch(),
  ccDisplay::eClientCoords);
```

displays the contents of the graphic list on a display console in client coordinates.

Modifying Result Graphics

You can iterate through graphic lists and modify individual items. For example,

```
cmStd vector<ccGraphicPtrh>& items = graphics.items();
cmStd vector<ccGraphicPtrh>::iterator iter = items.begin();
while (iter != items.end())
{
  if (dynamic_cast<ccGraphicText*>(iter->rep()))
    // if the item is a text label, remove it
      iter = items.erase(iter);
  else
```

```
    {
     // otherwise, change the color to red
        iter->rep()->color(ccColor::redColor());
    ++iter;
    }
  }
```

creates a list of the result graphics and iterates through the list, removing graphic text items and changing the color of other graphic items to red. The dynamic cast

```
dynamic_cast<ccGraphicText*>(iter->rep())
```

is used to check if the given item is a text item. The use of **rep()** is required to convert CVL pointer handles to regular pointers. See *Pointer Handles* on page 46.

You can modify attributes of the graphic items including their color and size. Some of these attributes are defined in **ccGraphics**, and some are defined in the **ccGraphics-**derived classes The following table lists the attributes that you can set or get for graphic items.

Attribute	Meaning
color()	The color of the graphic item.
map()	The graphic item mapped by a transformation object.
clone()	A pointer to a duplicate of the graphic item.
offset()	Offset to draw label for **ccText** graphic items.
fill()	Fill property of graphic item for **ccGraphicWithFill**-derived graphic items.

Table 52. Graphic item attributes

To display the modified results, you just erase and redraw the sketch. For example,

```
console.eraseSketch(ccDisplay::eClientCoords);
console.drawSketch(graphics.sketch(),
  ccDisplay::eClientCoords);
```

erases and redraws the new contents of the graphics list on the display console.

ccGraphic Classes

Most of the classes used to display result graphics are wrapper classes derived from **ccGraphic** for the static graphic shape classes defined in *shapes.h*. The following table summarizes the classes and the corresponding static graphic shapes they support.

Class Name	Description
ccGraphic	Base class with common member functions for derived classes.
ccGraphicCross	Wrapper class for **ccCross.**
ccGraphicEllipseAnnulusSection	Wrapper class for **ccEllipseAnnulusSection.**
ccGraphicText	Wrapper class for **ccText.**
ccGraphicPointIcon	Wrapper class for **drawPointIcon()** in **ccUITablet.**
ccGraphicBuiltin	Parameterized base class for built in graphic items.
ccGraphicSimple	Parameterized class for built in graphic items with no additional parameters. Provides wrapper classes for **ccPoint**, **ccLineSeg**, **ccLine**, **ccEllipseArc2**, **ccGenAnnulus**, **ccCoordAxes**, **ccPointSet**, **ccAffineRectangle**, **ccPolyline**, **cc2Wireframe**.
ccGraphicWithFill	Parameterized class for built in graphic items with a fill parameter. Provides wrapper classes for **ccRect**, **ccGenRect**, **ccCircle**, **ccEllipse2**.

Table 53. ccGraphic Class Summary

Graphic Display Application Notes

The following sections describe special situations you may encounter in your graphic display applications.

Using Nonlinear Transforms

ccDisplay accepts images with either linear or nonlinear transforms. For example, when you display a pel buffer by calling **ccDisplay::image(***pb***)** the pel buffer you pass can contain either a linear transform or a nonlinear transform. If it contains a nonlinear transform, **ccDisplay** linearizes the transform using a first order polynomial fit over the entire image area.

When you use nonlinear transforms in your applications, allowing **ccDisplay** to automatically linearize your transform may not produce the best results depending on the amount of transform nonlinearity. For example, if you wish the display to be most accurate in one particular area, you will get the best results if you linearize the transform in that area. To do this you need to linearize the transform yourself and then replace the pel buffer nonlinear transform with your new linear transform before calling **ccDisplay**. There are several ways of obtaining a linear approximation and the most common are discussed in the next section, *Linearizing a Transform*.

Be aware that if your display contains graphics in client coordinate space, and you allow **ccDisplay** to automatically linearize the pel buffer transform, the displayed graphics will be only an approximation of the original graphics. If you choose to linearize the transform yourself using one of the methods described in the next section, you can also convert a nonlinear graphics list to a linear approximation using the method discussed in *Converting a Graphics List* on page 300. This method may produce a better graphics approximation than that created automatically by **ccDisplay**.

Linearizing a Transform

When you convert a nonlinear transform to a linear transform the transform is said to be *linearized*. The new transform is a linear approximation of the original. Depending on your application, a linearized transform may, or may not, be accurate enough for display purposes. The method of linearization you use can have a significant effect on the linear transform's accuracy.

If your display contains a single important graphic, use the nonlinear transform where the graphic is located and make that transform the linear approximation for the entire image. If you have many graphics scattered about the image, you may wish to use the nonlinear transform at the image center as the linear transform for the whole image. A

third approach is to select an image area (bounding box) where the important graphic features are located and take the average nonlinear transform in this area as the linear approximation. See Figure 52.

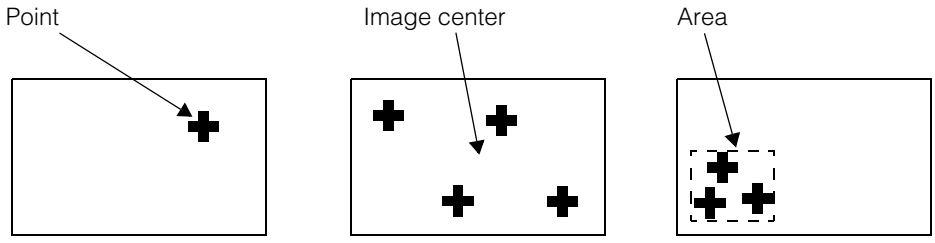


Figure 52. Linearization examples

The following is an example of a routine you can use to create a linear approximation of a nonlinear transform.

```
// Pass in a pel buffer containing a nonlinear transform.
// Return the same pel buffer now containing a linear
// approximation of the transform.
ccPelBuffer_const<c_UInt8> linearizePelBuffer(
                           const ccPelBuffer_const<c_UInt8> &pb)
{
    ccPelBuffer_const<c_UInt8> linPb = pb;
    //
    // Code to define new_xform
    //
    linPb.clientFromImageXform(new_xform);
    return linPb;
}
```

The code to define *new_xform* for the three examples in Figure 52 is shown below.

1. Point.

```
cc2Vect point = patMaxResult.location();
cc2Xform new_xform =
             pb.clientFromImageXformBase()->linearXform(point);
```

2. Image center.

```
ccPelRect rect = pb.window();
cc2Vect point(rect.ul().x() + 0.5*rect.width(),
              rect.ul().y() + 0.5*rect.height());
cc2Xform new_xform =
              pb.clientFromImageXformBase()->linearXform(point);
```

3. Area. Include a routine such as **linearXformArea()** (shown below).

   Call the routine as follows:

```
....
cc2Xform new_xform = linearXformArea(
             *pb.clientFromImageXformBase(), pb.window());
.....
.....
```

```
static cc2XformLinear linearXformArea(
   const cc2XformBase& xform,
   const ccPelRect& area)
{
   c_Int32 width = area.width(), height = area.height();

   // We need at least a 2x2 to have a well-defined fit.

   assert(width >= 2);
   assert(height >= 2);
   int nY = (height >= 8 ? 8 : height);
   int nX = (width  >= 8 ? 8 : width);
   int spacingX = width / nX, spacingY = height / nY;

   // Generate grid of points
   int nPoints = nX * nY;
   cmStd vector<cc2Vect> from(nPoints), to(nPoints);
   for (int i = 0; i < nY; i++)
   for (int j = 0; j < nX; j++)
   {
      int index = i * nX + j;
      from[index] = cc2Vect(area.ul().x() + j * spacingX,
                            area.ul().y() + i * spacingY);
      to[index]   = xform * from[index];
   }

   // Fit a first-order polynomial to the points
   cc2XformPoly firstOrderPoly(to, from, 1);

   // Return the linear xform.  Note that calling linearXform() on
   // our first-order polynomial can almost be thought of as a
   // cast, since no information is lost.  For a first-order
   // polynomial, the point at which the linearization is done
   // makes no difference [i.e. for this special case you would
   // get the same result no matter what point you passed in],
   // so just use the point (0, 0).
```

```
    return firstOrderPoly.linearXform(cc2Vect(0, 0));
}
```

Converting a Graphics List

The following is a routine you can include in your program and call to convert a nonlinear client coordinate **ccGraphicList** to a linear approximation client coordinate **ccGraphicList**. *clientFromImage* is the pel buffer nonlinear transform. *clientFromImageLinear* is the linear approximation of that transform you have decided to use. (See *Linearizing a Transform* on page 297 which describes three ways you can obtain an approximate linear transform).

```
void convertGraphicList(
    const ccGraphicList& glist,
    const cc2XformBasePtrh_const& clientFromImage,
    const cc2Xform& clientFromImageLinear,
    ccGraphicList& glistNew)
{
    cc2XformBasePtrh_const imageFromClient =
                                clientFromImage->inverseBase();
    cc2Xform imageFromClientLinearHere, convert;
    const cmStd vector<ccGraphicPtrh>& items = glist.items();
    c_Int32 nItems = items.size();
    for (c_Int32 i = 0; i < nItems; i++)
    {
        ccGraphicPtrh item = items[i];

        // Assume that the center of the enclosing rectangle is a
        // good place to linearize.
        ccRect encloseRect = item->encloseRect();
        cc2Vect centerClient = 0.5 * (encloseRect.ul() +
                                            encloseRect.lr());

        // Compute a transform to convert the graphic to image
        // coordinates.
        imageFromClientLinearHere =
                    imageFromClient->linearXform(centerClient);

        // Compute the full transform which can display in the
        // client coordinates that have been chosen for display.
```

```
        convert = clientFromImageLinear *
                                        imageFromClientLinearHere;

        // Convert the graphic from full nonlinear client coords to
        // the new approximate client coords.
        ccGraphicPtrh newItem = item->map(convert);
        glistNew.append(newItem, false);
    }
}
```

Summary

The following is an example of code you might use with a graphics list, whether the pel buffer transform is linear or nonlinear.

```
ccPMAlignResultSet resultSet;
ccPelBuffer<c_UInt8> pb;
ccGraphicList resultGraphics;
patMaxPattern.run(pb, runParams, resultSet);
result.draw(resultGraphics);
```

You then add the following code to linearize the pel buffer transform and display the result.

```
ccPelBuffer_const<c_UInt8> linPb = linearizePelBuffer(pb);
ccDisplayConsole *console =
                        new ccDisplayConsole(ccIPair(512, 512));
console->image(linPb, false);
ccGraphicList linResultGraphics;
convertGraphicList(resultGraphics,
                   pb.clientFromImageXformBase(),
                   linPb.clientFromImageXform(),
                   linResultGraphics);
console->drawSketch(linResultGraphics.sketch(),
                           ccDisplay::eClientCoords);
```

Display Examples

Several display code examples covering advanced topics are included with your CVL release in the sample code directory, *%VISION_ROOT%\sample\cvl*.

Sample application projects that illustrate CVL display concepts are discussed in *Display Sample Projects* on page 42.

Single file code samples that illustrate display concepts are shown in Table 54.

File name	Program description
disp.cpp	Shows how to use **ccDisplayConsole** windows to draw graphics.
dispprop.cpp	Shows how to use the **ccGraphicsProps** class.
disppump.cpp	Shows how to create a Windows message handler thread for a **ccDisplayConsole** (see *Windows XP and CVL Display* on page 302)
userbuf.cpp	Shows how to display the contents of a user-defined block of data in a pel buffer.

Table 54. Display-related single file code samples

See the complete list of single file code samples shipped with the current edition of CVL in that edition's *Getting Started* manual.

Windows XP and CVL Display

In Windows XP, it is important to construct a Windows message handler thread if you use a **ccDisplayConsole**. In earlier Microsoft operating systems (including Windows 98, NT, and 2000), it was possible to use a display console without a message handler thread. However, failing to provide a message handler thread for a display console in Windows XP may cause the display console to hang. See the sample code in *%VISION_ROOT%\sample\cvl\disppump.cpp* for an example of how to create a Windows message handler thread for a display console.

Using CVL Vision Tools 7

This chapter provides a brief overview of CVL vision tools. For each CVL vision tool described in this chapter, a section provides information about which CVL classes and functions you use to work with the vision tool. CVL vision tools are explained more thoroughly in the *CVL Vision Tools Guide.*

Using the PatMax and CNLSearch Tools tells how to use the CNLSearch and PatMax tools.

Using the Image Analysis Tools tells how to use the Blob and Caliper tools.

Using Vision Tools with Nonlinear Transforms describes how the CVL vision tools make use of nonlinear transformations (polynomial calibration).

Some Useful Definitions

CNLSearch Cognex Nonlinear Search, a pixel-oriented CVL vision tool that allows you to find the location of a particular feature within each of a series of search images, independent of nonlinear brightness changes in the successive search images.

PatMax A Cognex technology and a CVL vision tool for pattern location within images. PatMax uses a feature-based representation instead of a pixel grid representation, which allows the pattern to be located with very high accuracy even when the pattern is rotated or scaled in the target image.

PatInspect A CVL vision tool that uses PatMax technology to detect and report defects from the acquired image of an inspection object.

Using the PatMax and CNLSearch Tools

This section describes how to use the PatMax and CNLSearch vision tools. You follow the same basic steps to use both of these tools. CNLSearch finds instances of a trained model and PatMax finds instances of a trained shape or a trained image. For PatMax, both image training and shape training produce a trained pattern.

1. Obtain a training source similar to what you wish to locate in run-time images. For CNLSearch you need a training model, and for PatMax you need a training image or a shape model. You can use the Auto-select tool to locate appropriate portions of the PatMax pattern or CNLSearch model. See *Using the Auto-Select Tool* on page 311.
2. Train PatMax or CNLSearch.
3. Run the tool and search for the trained pattern or model in one or more run-time images.
4. Use the information returned by the tool about the pattern or model location in the run-time image.

As part of the process of training a pattern or model, specify the pattern or model origin. Both PatMax and CNLSearch return information about found instances of the pattern or model in terms of the pattern or model origin that you specify. The next section contains important information about specifying the pattern or model origin.

Pattern and Model Origin

When you train an image or shape as a PatMax pattern or a CNLSearch model, specify the model or pattern origin. When the tool locates the pattern or model in a run-time image, it returns the location of the origin of the pattern or model in the run-time image.

If you do not specify the origin explicitly, both PatMax and CNLSearch use the origin of the client coordinate system of the image or shape that you supply for training as the pattern or model origin. By default, the client coordinate system has its origin at the upper-left corner of the pixel with image coordinates (0, 0).

If you have created the training image as a subwindow of a larger image, the client coordinate system will still be associated with the image offset of the original image. The following figure illustrates the effect of this default behavior:

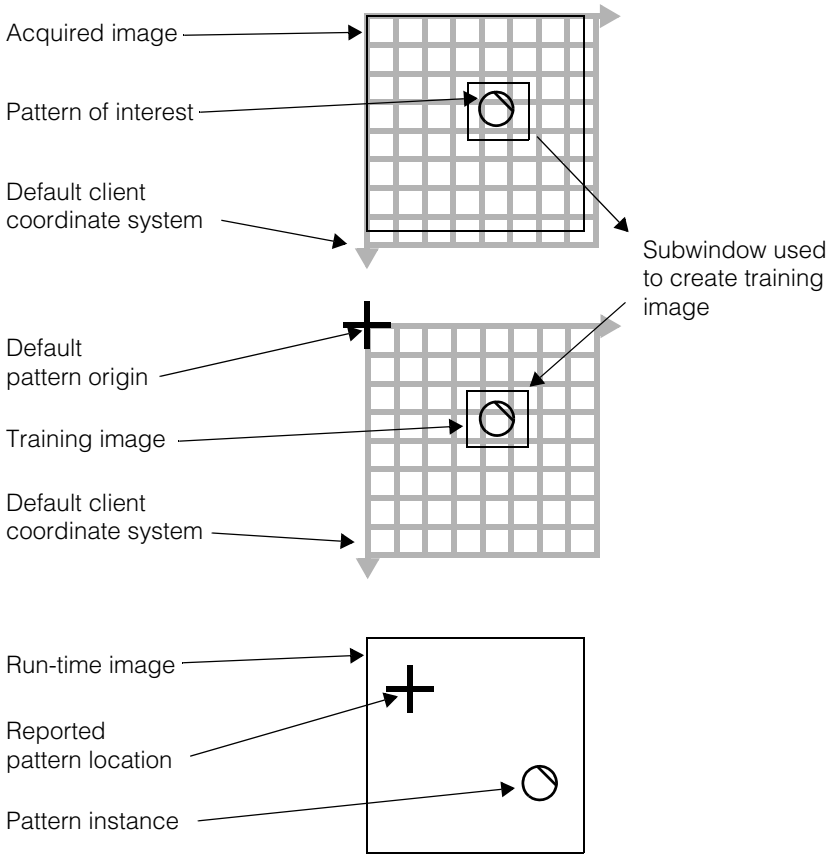


Figure 53. Effect of accepting default pattern or model origin on reported pattern locations

In almost all cases, explicitly set the pattern or model origin to refer to a location of interest within the pattern or model. The PatMax and CNLSearch sample code described in the following sections sets the pattern and model origin explicitly.

Using the CNLSearch Tool

This section describes how to use the CNLSearch vision tool.

1. Acquire an image to use for model training.
2. Create and configure a **ccCnlSearchTrainParams** specifying the model training parameters.

```
ccCnlSearchTrainParams
  trainParams(ccCnlSearchDefs::eNormalizedCnlpas);
```

3. Create and configure a **ccCnlSearchModel** and specify the model origin. In the following code, the origin is set to the center of the training image.

```
ccCnlSearchModel model;
cc2Vect modelCenter (
  modelImage.offset().x() + modelImage.width()/2.0,
  modelImage.offset().y() + modelImage.height()/2.0);

// Adjust origin for client coordinates
//
modelCenter = modelImage.clientFromImageXform()* modelCenter;
model.origin(modelCenter);
```

4. Train the **ccCnlSearchModel** using the parameters you specified.

```
model.train (modelImage, trainParams);
```

5. Create a **ccCnlSearchResultSet** to hold the search results.

```
ccCnlSearchResultSet results;
```

6. Create a **ccCnlSearchRunParams** that specifies the search parameters including the search algorithm, the accept and confusion thresholds, and the number of results to find.

```
ccCnlSearchRunParams runParams(
   ccCnlSearchDefs::eNormalizedCnlpas);
runParams.accept (0.8);
runParams.confusion (0.85);
runParams.maxNumResults (3);
```

7. Invoke the **ccCnlSearchModel::run()** function supplying it with a search image, the **ccCnlSearchResultSet** to hold the results, and the **ccCnlSearchRunParams** defining the search parameters.

```
model.run (searchImage, runParams, results);
```

8. Extract the results from the **ccCnlSearchResultSet**.

```
// Print the position and score for each result.
//
for (int i = 0; i < results.results().size(); i++)
  {
  cogOut << "Result " << i+1 << ": ";
  if (results.results()[i].found())
    {
    cogOut << "Score = " << results.results()[i].score() << ", "
         << "location = " << results.results()[i].location()
         << std::endl;
    }
  else
    cogOut << "Not Found" << std::endl;
  }
```

Using the PatMax Tool

This section describes how to use the PatMax vision tool. The following steps describe how to train PatMax using training image, and how to run PatMax to find instances of the trained pattern in a run-time image.

1. Acquire an image to use for pattern training.
2. Create a **ccPMAlignPattern** and train it using the training image.

   ```
   ccPMAlignPattern pat;
   pat.train(win);
   ```

3. Set the pattern origin to a meaningful location within the training image.

   ```
   pat.origin(cc2Vect(50,50));// In client coordinates
   ```

4. Create a **ccPMAlignResultSet** to contain the results of the search.

   ```
   ccPMAlignResultSet set;
   ```

5. Create a **ccPMAlignRunParams** and configure it with the number of results to find, the zone, and the degrees of freedom.

   ```
   ccPMAlignRunParams params;
   params.numToFind(3);
   params.zoneEnable(ccPMAlignDefs::kUniformScale);
   params.zone(ccPMAlignDefs::kUniformScale, 1, 1.2);
   ```

6. Perform the search by calling the **ccPMAlignPattern::run()** member function and supplying a run-time image, the **ccPMAlignRunParams** specifying the search parameters, and the **ccPMAlignResultSet** to contain the results of the search.

   ```
   pat.run(image, params, set);
   ```

7. Extract the results from the **ccPMAlignResultSet**.

```
cogOut << "number found = " << set.numFound() << std::endl;

int i;
for (i=0; i<set.numFound(); i++)
  {
  ccPMAlignResult r;
  set.getResult(i, r);

  cogOut << "result " << i << " = (x,y):(" << r.location().x()
  << "," << r.location().y() << ")" << "  "
  << (r.accepted() ? "accepted" : "not accepted") << std::endl;
  }
```

PatMax Shape Training

Shape training allows you to train PatMax directly from a shape model rather than from an image. Shape models are sometimes called geometric descriptions or synthetic models.

When you train PatMax using a shape model you pass it a pointer to a **ccShape** object that describes the shape. Cognex provides an extensive library of primitive shapes for you to use such as rectangle, circle, line, ellipse, point, and others. You can also create your own primitive shapes using the **ccGenPoly** class. It is a good practice to configure polarity and weights for shapes before training PatMax using **ccShape** objects. Polarity and weight is discussed in the **ccShapeModel** reference page and in the *Shape Models* chapter of the *CVL Vision Tools Guide*.

While all of these primitive shapes are available, most shapes you will need to train are more complex and require multiple primitives combined in various ways to represent the desired shape. To create these shapes Cognex provides shape trees, which are container classes that can hold many primitives related in various ways to represent one shape. The shape tree classes also derive from **ccShape**. You will normally pass your shape as a pointer to a shape tree rather than as a pointer to a primitive. The shape and shape tree classes are described in the *Shapes* chapter of the *CVL User's Guide*, where you will also find information about building shape trees.

Troubleshooting

After training a shape model, Patmax may sometimes fail to find any result. If this happens, perform the following actions:

1. Make sure that the shape models (at least roughly) the high contrast contours in the image. Training diagnostics can help you verify this.
2. Make sure polarity of the shape is correct. Training diagnostics can help you verify polarity.

3. Make sure the shape is defined in the same client coordinate space as the run-time image. Also, the *clientFromImage* transform supplied to **ccPMAlignPattern::train()** should have a similar scale as the *clientFromImage* transform of the run-time image.
4. Turn off PatMax temporarily and use PatQuick to locate the model. Look where the model does not match the image in the diagnostics following the alignment and modify the model accordingly. After the modifications try again with PatMax.
5. If 4 does not work, try lowering the accept threshold and see if this allows you to find the model in the correct position. If so, you can, then, adjust the model to maximize the matching score and raise the accept threshold accordingly.
6. If you still cannot locate the model, try increasing the elasticity parameter.
7. If you are using **ccGenPoly** objects, make sure corner rounding is set appropriately.

Using the Auto-Select Tool

This section describes how to use the Auto-select tool to locate good candidate regions in images for use as pattern or model training images.

1. Create a **ccAutoSelectParams** and configure it to specify the model size, number of candidates, and sub-sampling factor.

```
ccAutoSelectParams params;
params.sample(1);
params.maxNumResult(1);
params.modelSize(ccIPair(128, 128));
```

2. Create and configure a run-time search parameter object for the search method you want to use (CNLSearch or PatMax).

```
// Construct the run-time parameters structure for CnlSearch
//
ccCnlSearchRunParams
  (cnlsParamsccCnlSearchDefs::eNormalizedCnlpas);
```

3. Create a vector of **ccAutoSelectResult**s to hold the results.

```
cmStd vector<ccAutoSelectResult> results;
```

4. Call the **cfAutoSelect()** global function, supplying an input image, the **ccAutoSelectParams**, the **ccCnlSearchRunParams** (or **ccPMAlignRunParams**), and the vector of **ccAutoSelectResults**.

```
cfAutoSelect(source, params, cnlsParams, results);
```

5. Extract the results.

```
cogOut << "Result coordinates: ("
       << results[0].location().x()
       << ", " << results[0].location().y() << ")"
       << cmStd endl;
```

Using PatInspect

Structure of a PatInspect application

A PatInspect application consists of four main steps:

- Region selection
- Training
- Run-time inspection
- Obtaining inspection results

Region selection

This is the sequence of the basic actions that must be followed during this step:

1. Select the alignment region and the inspection regions within a reference image. To each inspection region assign an inspection mode.
2. Create a **ccPMInspectPattern** object.
3. Add the inspection regions that you have selected in step 1 to the **ccPMInspectPattern** object created in step 2 by calling **ccPMInspectPattern::addInspectRegion()**.

Training

This is the sequence of basic actions that must be followed during this step:

1. Start training by calling **ccPMInspectPattern::startTrain()**. Pass the alignment region as argument. If you have a pose use the no argument overload (see the *CVL Class Reference*).
2. Create a **ccPMInspectStatTrainParams()** object that controls the search for the alignment region within the training image.
3. Perform statistical training by calling **ccPMInspectPattern::statisticTrain()**. Pass the training images and the **ccPMInspectStatTrainParams()** object that you created in step 2 as arguments. If you have a pose, you can pass it now by calling the appropriate overload of **ccPMInspectPattern::statisticTrain()** (see the *CVL Class Reference*).
4. End train by calling **ccPMInspectPattern::endTrain()**.

Run-Time Inspection

This is the sequence of basic actions that must be followed during this step:

1. Create a **ccPMInspectResultSet** object to store the inspection results.
2. Create a **ccPMInspectRunParams** object that controls the search for the alignment region within the run-time inspection image.
3. Perform the inspection by calling **cPMInspectPattern::run()**. Pass the **ccPMInspectResultSet** and **ccPMInspectRunParams** objects created in steps 1 and 2 as arguments.

Get Inspection Results

Get inspection results by calling one of the functions in the following table:

Inspection mode	Function
Intensity difference	**ccPMInspectResult::diffImage()**
Feature difference	**ccPMInspectResult::getBoundaryDiff()**
Blank scene inspection	**ccPMInspectResult::getBlankSceneMeasurement()**

It is critical that all the steps and actions are performed in the order presented in this section. The next section provides a complete example of PatInspect application.

Example of PatInspect Application

This application shows you how to select regions within a reference image, how to perform statistical training, and how to run an inspection. The example also shows how to get and display inspection results for each inspection region.

1. Perform all the steps necessary for acquisition. The following code starts with live-video mode, acquires an image of the object you have positioned in front of the camera, and displays it on a display console. The image acquired is the reference image from which the alignment and inspection regions are selected.

```
cc8100 &frameGrabber=cc8100::get(0);

const ccStdVideoFormat &videoFormat =
       ccStdVideoFormat::getFormat(cmT("Sony XC75 640x480"))

ccStdGreyAcqFifoPtrh
  fifo=videoFormat.newAcqFifo(frameGrabber);
```

```
//Live Video
ccDisplayConsole display(ccIPair(640,480));

display.startLiveDisplay(fifo.rep());
cfWaitForContinue();
display.stopLiveDisplay();

//Acquire and display the reference image
fifo->start();
ccPelBuffer<c_UInt8> trainImage=fifo->complete();
display.image(trainImage);
```

2. Select Regions. The following code shows how to graphically select the alignment and inspection regions within the acquired reference image. In this example two inspection regions are selected.

```
////Select the Alignment Region/////

// Create a rectangle to display on the console
ccUIRectangle *myRect = new ccUIRectangle;
myRect->color(ccColor::red);
myRect->condVisible(true);
display.addShape(myRect,ccDisplayConsole::eImageCoords);
MessageBox(NULL, "Position rectangle to select alignment
  region.Hit OK to Continue","Alignment Region", MB_OK);
ccRect rectPos=myRect->rect();
delete myRect;

//Create a ccPelRect with dimensions of ccRect
ccPelRect trainWindow(rectPos.ul().x(),rectPos.ul().y(),
  rectPos.lr().x()-rectPos.ul().x(),
  rectPos.lr().y()-rectPos.ul().y());

//Create the ccPelBuffer that contains the alignment region
ccPelBuffer<c_UInt8> AlignRegion = trainImage;
AlignRegion.window(trainWindow);

//// Select Inspection Regions ///////////

//Create an array of inspection regions
ccPelBuffer<c_UInt8> InspectionRegion[2];

// Select inspection regions following the same steps
// used to select the alignment region
for(int i = 0; i < 2; i++){
  ccUIRectangle *myRect = new ccUIRectangle;
  myRect->color(ccColor::red);
  myRect->condVisible(true);
  display.addShape(myRect,ccDisplayConsole::eImageCoords);
```

```
  MessageBox(NULL, "Position rectangle to select inspection
    regions.Hit OK to Continue", "Inspection Region", MB_OK);
  ccRect rectPos=myRect->rect();
  delete myRect;
  ccPelRect Window(rectPos.ul().x(),rectPos.ul().y(),
    rectPos.lr().x()-rectPos.ul().x(),
    rectPos.lr().y()-rectPos.ul().y());
  InspectionRegion[i] = trainImage;
  InspectionRegion[i].window(Window);
}

// Create ccPMInspectPattern object
ccPMInspectPattern pattern;

//Add Inspection regions to ccPMInspectPattern object.
//Inspect region 0 in intensity difference mode and region 1
//in feature difference mode
pattern.addInspectRegion(InspectionRegion[0],
  ccPMInspectDefs::eIntensityDifference);
pattern.addInspectRegion(InspectionRegion[1],
  ccPMInspectDefs::eBPDifference);
```

3. Training. The following code shows how to perform statistical training on a sample of 3 images. The images are first acquired and then added to the sample of training images.

```
//Start Training
pattern.startTrain(AlignRegion);

//Create a ccPMInspectStatTrainParams object
ccPMInspectStatTrainParams trainParams;

//Select Degrees of freedom
trainParams.zoneEnable(ccPMInspectDefs::kAngle |
  ccPMInspectDefs::kUniformScale);
trainParams.zone(ccPMInspectDefs::kAngle, -90.0, 90.0);
trainParams.zone(ccPMInspectDefs::kUniformScale, 0.9, 1.1);

//Statistic train on 3 images
ccPelBuffer<c_UInt8> StatisticTrainImages[3];
for(i = 0; i < 3; i++){
  //Live Video
  display.startLiveDisplay(fifo.rep());
  cfWaitForContinue();
  display.stopLiveDisplay();
  //Acquire the training image
  fifo->start();
  StatisticTrainImages[i] = fifo->complete();
  display.image(StatisticTrainImages[i]);
```

```
  MessageBox(NULL, "Use this image for statistic train.Hit OK
    to Continue", "Statistic Training", MB_OK);
  //Add to sample
  pattern.statisticTrain(StatisticTrainImages[i],
    trainParams);
  if (i < 2)
    MessageBox(NULL, "Acquire next sample for statistic
      training.Hit OK to Continue","Statistic Training",
      MB_OK);
  else
    MessageBox(NULL, "Training is over.Hit OK to Continue",
      "Statistic Training", MB_OK);
}

//End training
pattern.endTrain(false);
```

4. Inspection. The following code shows how to perform an inspection on a run-time inspection image. The inspection image is first acquired and then inspected.

```
// Live Video
display.startLiveDisplay(fifo.rep());
cfWaitForContinue();
display.stopLiveDisplay();

//Acquire and display Image to inspect
fifo->start();
ccPelBuffer<c_UInt8> analysisImage=fifo->complete();
display.image(analysisImage);

//Create a ccPMInspectResultSet object to store inspection
//results
ccPMInspectResultSet resultset;

//Create a ccPMInspectRunParams object
ccPMInspectRunParams runParams;

//Set number of instances to find
runParams.numToFind(1);

//Set degrees of freedom
runParams.zoneEnable(ccPMInspectDefs::kAngle |
  ccPMInspectDefs::kUniformScale);
runParams.zone(ccPMInspectDefs::kAngle, -90.0, 90.0);
runParams.zone(ccPMInspectDefs::kUniformScale, 0.9, 1.1);

//Run PatInspect
pattern.run(analysisImage, runParams, resultset);
```

5. Get/view results. The following code shows how to get and display the inspection results.

```
//// View Boundary Difference Results ////

//Get intensity difference image for region "0" from results
//vector
  ccPelBuffer<c_UInt8> diffImage =
  (resultset.results())[0].diffImage(0);

// Create display console to display difference image
ccDisplayConsole diffConsole("Difference Image",
  ccIPair(400,400), ccIPair(300,300));

// Display difference image
diffConsole.image(diffImage);
diffConsole.fit();

//Pause to see difference region
cfWaitForContinue();

//// Get/View Boundary Difference Results ////

//Create a ccPMInspectBoundaryDiffData object
ccPMInspectBoundaryData diffData;

//Get simple boundary difference for region "1" from results
//vector
(resultset.results())[0].getBoundaryDiff(diffData, 1);

//Create a graphic list
ccGraphicList BoundList;

//Display features on graphic list
diffData.displayFeatures(BoundList);

//Create display consoles to display boundary differences
ccDisplayConsole boundConsole("Boundary difference Image",
  ccIPair(100,100), ccIPair(300,300));

//Sketch graphics on difference image display
boundConsole.drawSketch(BoundList.sketch(),
  ccDisplay::eClientCoords);
```

PatInspect Archiving

PatInspect does not allow you to archive partially trained patterns. A partially trained pattern is a **ccPMInspectPattern** object whose training has not been completed by calling **ccPMInspectPattern::endTrain()**.

Troubleshooting

If the throw *BadImage: unable to statisticTrain from image* occurs during blank scene training, you can try to adjust the *minimumFeatureSize* parameter. Leaving this parameter in default state may generate a throw at train time.

When inspecting in feature difference mode, occasionally result features contain a number of short (1 point) extra or missing edges. These edges usually are located at the corners. You can set *minimumFeatureSize* to greater than 1 to filter out these unwanted edges.

Using the Image Analysis Tools

This section describes how to use the Caliper and Blob tools.

Using the Caliper Tool

This section describes the steps you follow to use the Caliper tool.

1. Construct a **ccAffineRectangle** and **ccAffineSamplingParams** that specify the projection region for the caliper tool.

```
cc2Vect  po(20,20),
         px(40,20),
         py(20,40);
ccAffineRectangle affRect(po, px, py);
ccAffineSamplingParams sample(affRect, 20,20);
```

2. Construct a vector that contains pointers one or more scoring functions to determine how edge candidates in the image will be scored.

```
ccScoreContrast aScoringFunction;// Use default values
cmStd vector<ccCaliperScore *> scoringFunctions(1);
scoringFunctions[0] = & aScoringFunction;
```

3. Construct a **ccCaliperRunParams** that specifies the number and position of the expected edges. Add the scoring functions to the **ccCaliperRunParams** by calling the **scoringMethods()** function.

```
// Set the run parameters. Search for one edge, any polarity.
ccCaliperRunParams runParams(0.0, ceDontCare);
runParams.scoringMethods(scoringFunctions);
```

   If you do not specify any scoring function for the **ccCaliperRunParams**, the **ccCaliperRunParams** will use a single default-constructed **ccScoreContrast** to score results.

4. Construct a **ccCaliperResultSet** to hold the results.

```
ccCaliperResultSet rslt;
```

5. Call the **cfCaliperRun()** global function supplying the **ccAffineSamplingParams**, an input image, the **ccCaliperRunParams**, and the **ccCaliperResultSet**.

```
cfCaliperRun(searchImage, sample, runParams, rslt);
```

6. Extract the results.

```
for (int i = 0; i < rslt.results().size(); i++)
  {
  cogOut << "Caliper result "  << i+1 << ": "
         << "Score = "        << rslt.results()[i].score()
         << ", "
         << "Model Origin at " << rslt.results()[i].position()
         << std::endl;
  for (int j = 0;  j < rslt.results()[i].resultEdges().size();
       j++)
    {
    cogOut << "  Result edge " << j+1 << ": "
           << "location = "
           << rslt.results()[i].resultEdges()[j].position()
           << std::endl;
    }
  }
```

Enhanced Caliper Performance on Multiprocessor Systems

When using the Caliper tool on a multiprocessor PC, you can use the **ccCaliperRunParams::maxNumResults()** setter with the maximum number of expected individual results as its argument to reduce memory contention between multiple threads running on different processors. The result is improved Caliper performance on a multiprocessor PC over that expected of a single-processor PC. This setting is necessary to achieve the performance improvement on a multiprocessor PC, as *maxNumResults* is set to one by default.

Using the Blob Tool

This section describes the steps you follow to use the Blob tool.

1. Construct a **ccBlobParams** and configure it with the segmentation parameters that are appropriate for your image.

   ```
   ccBlobParams params;   // Use default parameters
   ```

2. Construct a **ccBlobResults** to hold the results of the blob analysis.

   ```
   ccBlobResults results;
   ```

3. Call the global function **cfBlobAnalysis()** supplying the image to analyze, the **ccBlobParams**, and the **ccBlobResults**.

   ```
   cfBlobAnalysis(image, params, results);
   ```

4. Obtain a pointer to the **ccBlobSceneDescription** contained in the **ccBlobResults**.

```
ccBlobSceneDescription* bsd = results.connectedBlobs();
```

5. Specify any sorting or filtering criteria for the blob scene description.

```
bsd->setSort(ccBlob::eArea,
        ccBlobSceneDescription::eDescending);
```

6. Extract the results of the analysis.

```
cmStd vector<const ccBlob*> blobs = bsd->allBlobs();
for (int i = 0; i < blobs.size(); i++)
  {
  cc2Vect com(blobs[i]->centerOfMass());
  cogOut << "Feature: " << std::setw(3) << i+1
        << " Label: " << std::setw(3)
        << (int)blobs[i]->label()
        << " Area: " << std::setw(3)
        << blobs[i]->area()
        << " Center of mass: (" << com.x() << ","
        << com.y() << ")" << std::endl;
  }
```

Using Vision Tools with Nonlinear Transforms

The client coordinate transform that is part of every pel buffer can be either linear (**cc2XformLinear**) or nonlinear (**cc2XformPoly**). All CVL vision tools support linear client coordinate transforms but only certain tools also support nonlinear client coordinate transforms.

The following CVL vision tools can make use of nonlinear transforms to return higher accuracy results than can be obtained using a linear transform:

- CNLSearch
- Blob
- Caliper
- PatMax

You can use the CVL Grid-of-Dots Calibration tool (described in the *CVL Vision Tools Guide*) to create a **cc2XformPoly** transformation (described in *Math Foundations of Transformations* on page 404). A polynomial nonlinear transformation uses a first, third, or fifth order polynomial equation to represent the slight nonlinear distortions introduced by many optical systems.

This section describes some general issues related to using nonlinear transformations with CVL vision tools, and it describes the specific approach taken by each of the tools that support the use of nonlinear transformations.

General Considerations

The following general guidelines apply to all uses of nonlinear client coordinate transformations by CVL Vision Tools:

- There is a minimal execution speed penalty for supplying an image with nonlinear client coordinates.
- The tools will return meaningful results only in cases where the nonlinear transformation is only slightly nonlinear. Specifically, if you are using a **cc2XformPoly** to correct distortion of more than about 2 pixels, using a nonlinear transformation may not improve vision tool accuracy.
- The CVL vision tools that support the use of nonlinear transformations do so by applying the nonlinear portion of the transformation to their results, not by transforming the input image using the nonlinear transformation.

  This approach provides a reasonable trade-off between speed and accuracy. While it attempts to approximate the results that would be achieved by transforming the entire input image before running the tool (which would take a prohibitive amount of time), it does not duplicate these results.

- No special actions are required to take advantage of nonlinear client coordinates. If you supply an input image that has a **cc2XformPoly** for its client coordinate transform, you will get more accurate tool results.

Tool-Specific Implementation Notes

The following sections describe how each of the tools listed above make use of nonlinear transformations.

CNLSearch

The CNLSearch tool is a pixel-based tool. It works by finding a close match between the pattern of pixels in the model image and the pattern of pixels in a portion of the run-time image.

When the run-time image has a nonlinear client coordinate system, the CNLSearch tool operates normally (using the pixels in the image). It then transforms the computed result location (the location of the model origin in the run-time image) through the nonlinear client coordinate system.

Note Nonlinear client coordinates are fully supported for the model training image as well. If you supply a model training image that has a nonlinear client coordinate system, CNLSearch interprets the model origin in the full, nonlinear client coordinates. The returned model origin location represents the transformation between the model origin (in model training image client coordinates) and the model location (in run-time image client coordinates).

Note that since the input image is not transformed before the search, you may see slightly lower scores for model instances found in areas of higher distortion, typically near the corners of the input image. While the scores may be reduced, the use of nonlinear client coordinates will increase the accuracy of the returned model location.

Blob Tool

The Blob tool is a pixel-based tool. It computes its results based on mathematical manipulations of the pixel values in the input image.

If you supply a run-time image that has a nonlinear client coordinate system, the Blob tool operates normally (using the pixels in the image). For each feature (blob or hole) detected in the image, the Blob tool computes the best linear approximation of the nonlinear client coordinate system at the center of mass of the feature. This local linearization is then used to map each of the feature's measurements (center of mass, area, perimeter, and so on) before returning them.

Keep in mind that any local linearization of a nonlinear client coordinate system is most accurate at the point for which the linearization is computed. In the case of a Blob feature, this point is the center of mass. Accordingly, the greatest improvement in Blob tool accuracy offered by this method is for the center of mass. Particularly in the case of larger features (those spanning more than 10 pixels), measures related to the whole feature (perimeter, area, acircularity, and the bounding box-related measures) may tend to show less improvement.

Caliper

The Caliper tool is a pixel-based tool. It computes its results by applying affine transformations and projections to the pixels in the input image, then detecting the locations of edges or edge pairs in the projection image. The Caliper tool makes use of nonlinear client coordinates in two ways: by adjusting the projection region and by adjusting the result locations. Each of these features is described in this section.

Projection Region Adjustment

If you supply the Caliper tool with a run-time image that has a nonlinear client coordinate system, the tool computes a local linearization of the nonlinear client coordinates across the projection region that you specify to the tool. The Caliper tool computes this linearization by performing a least-squares fit of five points (the four corners and the center) within the region.

The resulting linear transformation is used to construct the affine rectangle used for sampling.

This computation assumes that you have specified the input projection region in (nonlinear) client coordinates. Accordingly, if your application depends on an operator selecting a region using an image-coordinate-based region selector, adjust the operator's input region so that it reflects the nonlinear client coordinates associated with the input image. Figure 54 shows how this works.

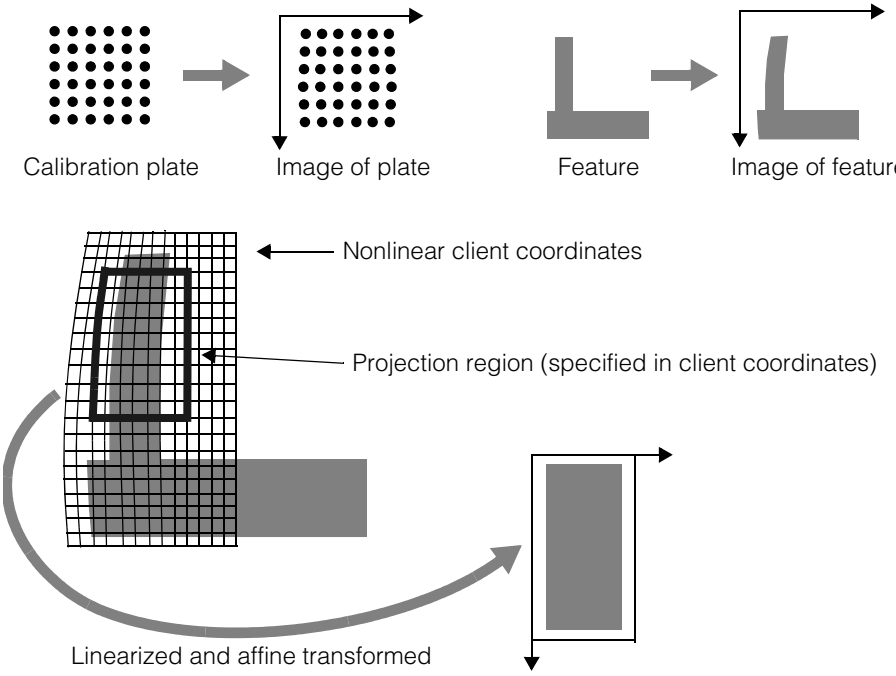


Figure 54. Caliper project region transformation

Note The degree of distortion shown in Figure 54 is exaggerated. Keep in mind that using a nonlinear client coordinate system with the Caliper tool only improves accuracy if the maximum corrected distortion is about 2 pixels.

Peak and Edge Location Reporting

The Caliper tool detects edge peaks in the linearized and transformed image produced as described in the previous section. Before the tool performs any scoring, however, it converts each edge and peak location back into the nonlinear client coordinate system. All Caliper results are reported in client coordinates.

PatMax

Unlike the other CVL vision tools that support nonlinear client coordinates, PatMax is a feature-based rather than pixel-based tool. PatMax works by locating a pattern of features in the run-time image that matches a trained pattern of features, then returning the transformation that describes the relationship between the trained pattern and the pattern in the run-time image.

If you supply a run-time image that has a nonlinear client coordinate transform, PatMax computes the best-fit linearization of the client coordinate transform across the entire input image, then applies that transformation to the features in the input image before locating the trained pattern.

Note Nonlinear client coordinates are fully supported for a pattern training image as well. If you supply a pattern training image that has a nonlinear client coordinate transform, PatMax computes the best-fit linearization of the transform across the entire input image, then applies that transformation to the features in the input image before training the pattern.

After PatMax locates the pattern instance or instances in the run-time image, it computes the best-fit linearization of the nonlinear client coordinate transform at the center of each found pattern instance (where the center is the center of the smallest rectangle that encloses all of the pattern's features), and it maps the returned pose through that local linearization before returning it.

Shapes

8

This chapter describes the design and use of shapes in CVL. It contains the following sections:

Some Useful Definitions defines some terms that you will encounter as you read this chapter.

Shapes Overview provides an introduction to shapes in CVL.

ccShape Interfaces gives an overview of the most important public interfaces of the **ccShape** base class.

Shape Hierarchies provides an overview of shape trees.

Rasterizing Shapes describes the global functions that rasterize shapes into pel buffers.

Shape Geometries describes all **ccShape**-derived classes that implement shapes in CVL.

Some Useful Definitions

closed contours Contours that do not have start and end points, such as circles, rectangles, and closed polygons. Infinite contours, such as lines, are also closed.

contours Shapes that are continuous paths in the plane. Contours may be drawn with a single stroke of a pen without lifting the pen or retracing. Most *primitive shapes* are contours.

handedness Non-intersecting, closed contours are either right handed or left handed. Handedness determines the direction in which points are generated when the curve is sampled.

hierarchical refinement Hierarchical refinement of a shape hierarchy preserves the structure of the original hierarchy up to the leaf nodes. The leaf nodes may themselves be expanded into full hierarchies in the refinement.

hole regions Regions that are children of solid regions in a region tree.

Non-Uniform Rational B-Splines (NURBs) Another name for the CVL implementation of de Boor splines. De Boor spline control points have associated positive scalar weights. Increasing the weight of any control point pulls the spline curve toward that point; decreasing the weight reduces the effect of that point on the curve.

open contours Continous curves with well-defined start and end points, which may coincide, through which they can be connected to other open contours. Examples of open contours are line segments, elliptical arcs, and open polygons.

perimeter positions Arbitrary and unique points along CVL shapes. Perimeter ranges can be used to delineate arbitrary sections of a shape.

perimeter ranges Pairings of a perimeter position and a signed distance that delineate arbitrary sections of a shape

phantom holes Holes at the root of a region tree used to group disjoint solid regions into a single region tree.

primitive shapes Shapes that are not made up of a shape tree, such as a line segment or a circle.

regions Shapes that partition the plane into an *inside* and an *outside*. The inside has a finite area and extent. The outside is the complement of the inside.

shape hierarchy A drawing usually comprises a shape hierarchy consisting of several component shapes. See also *shape tree*.

shape information objects	Objects that encapsulate detailed information about specific shapes. This information is required for the perimeter and parameterization functions.
shape tree	A composition of shapes organized in an hierarchical fashion, starting with a root node and ending at the leaf nodes. CVL has specialized shape tree classes to contain contour shapes and region shapes, and a general shape tree to contain any kind of shape.
solid regions	Regions that are children of hole regions in a region tree.
wireframes	Type of generalized polygon used to implement a synthetic model. To use a wireframe shape with PatMax, you must put the wireframe into a shape tree prior to training.

Shapes Overview

As the base class for most shapes, the **ccShape** class offers a common interface to support high-level tools that operate on shapes. By accessing shape information through the **ccShape** base class, these tools are able to work automatically on any kind of shape, without requiring separate versions of a tool for each type of shape. Two immediate examples are synthetic PatMax training and synthetic rendering, both of which were previously available only for **cc2Wireframe** shapes. The **ccShape** base class now provides methods that allow these tools to be implemented generically for all shapes.

A common shape base class also provides support for shape hierarchies (also known as *shape trees*), which often contain arbitrary collections of shapes. Providing pointers to a base class facilitates the handling of these complex shape hierarchies. See *Shape Hierarchies* on page 343.

Finally, a common base class facilitates adding new shapes. Some CAD files contain shapes such as splines. Classes representing these shapes have been added to support the importing of CAD files into CVL (for example, **ccBezierCurve** and **ccCubicSpline** and its derived types). As they all inherit from the **ccShape** base class, these new shape types are able to incorporate all of the existing functionality for shapes that operate through the base class.

Table 55 shows the possible origins of shape information. Whatever the origin, shape information placed in a shape tree can be used with a number of CVL vision tools.

Illustrated in sample file	Origin of shape information		Example uses for shape or shape tree
pmalign3.cpp	Drawn onscreen with **ccUI*** API		To create a mask for the Blob tool
pmalign4.cpp	Imported from DXF format CAD file	**ccShape**	Use with the Edgelet Filtering tool
pmalign5.cpp	Drawn in code with shapes API		As a PatMax model (uses shape models)

Table 55. Possible origins and uses for shape information

Shape Class Hierarchy

The class derivation hierarchy of CVL shapes, including the new shapes base class (**ccShape**) and the new hierarchical shapes base class (**ccShapeTree**) is shown in Figure 55.

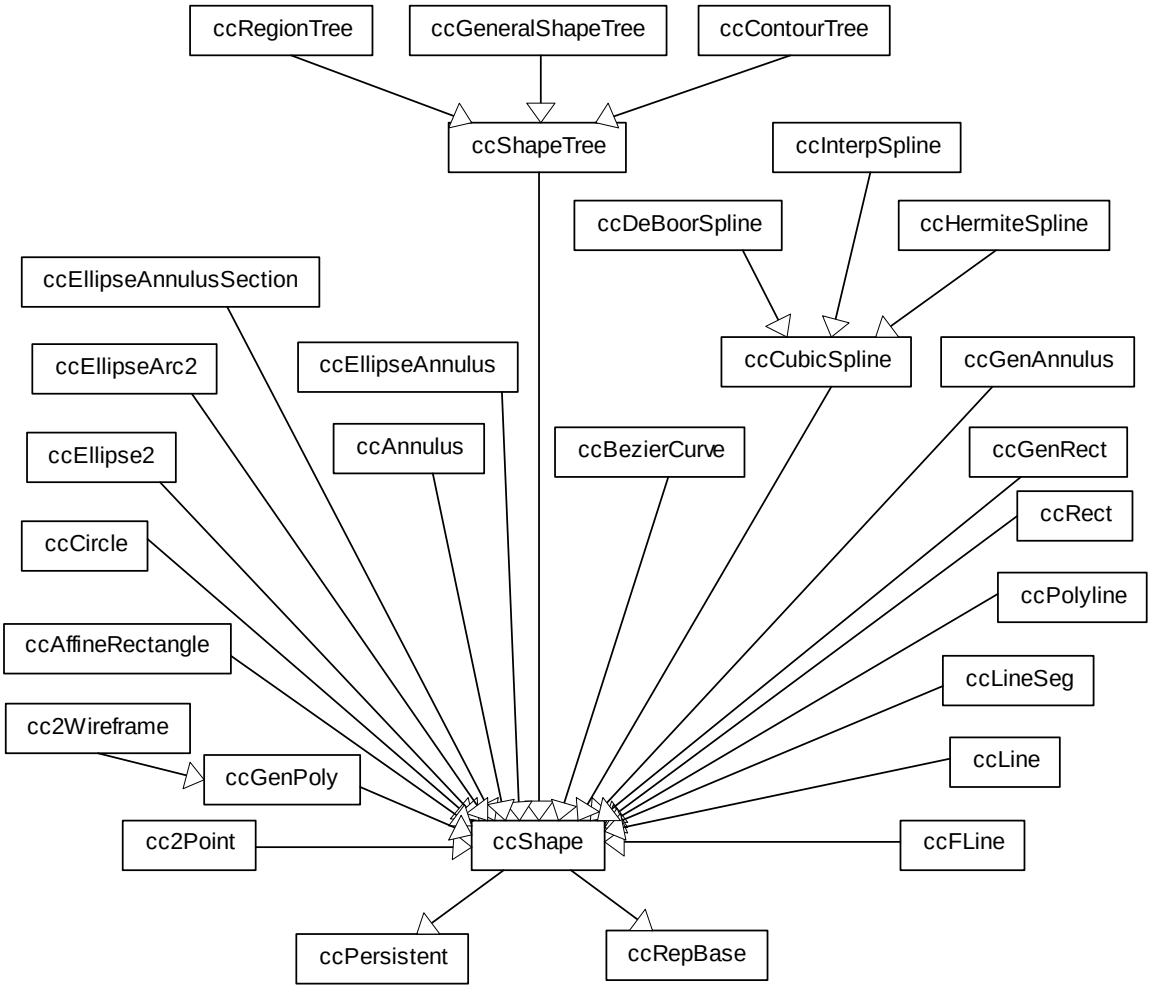


Figure 55. Shape class inheritance hierarchy

The **ccShapeTree** class is the abstract base class for all hierarchical shapes. A shape tree is also a shape, and algorithms that use the **ccShape** base class interface need not differentiate between operating on *primitive* (non-hierarchical) shapes and shape trees. See also *Shape Hierarchies* on page 343.

ccShape Base Class

The **ccShape** base class is a *lightweight* interface. It contains no data members other than those inherited from its own base classes, **ccPersistent** and **ccRepBase**. It provides several low-level virtual functions that derived classes implement for specific shapes.

One important new method that **ccShape** adds is a way to enumerate points and tangent directions along the contour of a shape in the form of the **sample()** method. See also *Sampling Shapes* on page 339.

Several methods of the **ccShape** base class are defined only for particular types of shapes. For example, **startPoint()** is valid only for open contours, while **within()** is valid only for regions. In many cases, the validity of invoking a particular shape method cannot be determined at compile time. For example, **within()** is defined for closed polylines but not open ones, and the open/closed state of a polyline can be dynamically changed during run time. For this reason, CVL performs most validity checks at run time. Thus, for example, **startPoint()** throws a *ccShapesError::NotOpenContour* exception when invoked on a shape that is not an open contour during run time.

ccShape Interfaces

This section gives a high-level overview of the interfaces of the **ccShape** base class. For detailed information, see the *CVL Class Reference*.

Classification Queries

Shapes can be classified according to a number of properties. Such properties group shapes according to whether they:

- Are open contours
- Are regions
- Are empty
- Have finite extent
- Have well-defined tangents along their boundaries
- Are decomposed
- Are reversible

The **ccShape** base class provides the following predicates for classifying shapes according to the above properties:

- The **isOpenContour()** query identifies open contours. A shape is an open contour if it is a continuous curve with identifiable start and end points that allow it to be connected to other open contours. See also *Contours and Regions* on page 335.
- The **isRegion()** query identifies region shapes. A shape is a region if it divides the plane into well-defined inside and outside regions. The inside region must be finite. See also *Methods Specific to Regions* on page 336.
- The **isFinite()** query identifies shapes that have finite extent. Empty shapes are also considered to be finite.
- The **isEmpty()** query identifies shapes that are empty. A shape is considered empty if the set of points that lie on its boundary is empty. For example, a **ccPolyline** with zero vertices is empty.
- The **hasTangent()** query identifies shapes that have have well-defined tangents at all but a finite number of points on their boundary, such as at corners. This method returns true for most shapes, other than points and line segments of zero length.
- The **isDecomposed()** query identifies shapes that have been decomposed into their component parts. All shapes can be decomposed into line segments, ellipse arcs, or Bezier curves, or a combination of these shapes, possibly arranged into shape trees. See also *Decomposing Shapes* on page 338.

- The **isReversible()** query identifies shapes that can be reversed. All open contour shapes are reversible, while most closed contour shapes (including circles and rectangles) are not. Directional lines (**ccLine**s) are reversible, while normal lines (**ccFLine**s) are not.

Geometric Queries

The **ccShape** base class provides the following predicates for identifying certain geometric properties of shapes:

- The **boundingBox()** query identifies the smallest rectangle that encloses a shape. A number of algorithms that cull shapes based on their extent use this information.
- The **nearestPoint()** query locates the nearest point on the boundary of a shape to a given point.
- The **distanceToPoint()** query returns the distance between the nearest point on the shape to a given point.

Contours and Regions

Contours are shapes that are continuous paths in the plane. They may be drawn with a single stroke of a pen, without lifting the pen or retracing. Most primitive shapes are contours. Exceptions are classes such as **ccPointSet** and **ccCross**, as well as all of the annulus classes, each of which comprise two contours.

Contours are considered *open* if they possess well-defined start and end points, both of which must be points not located in the interior of the contour. Examples include line segments, elliptical arcs, and open polygons. There is an implied direction along an open contour from the start point to the end point. Open contours are important because they form the building blocks of **ccContourTree**s. This hierarchical shape describes a complex contour obtained by connecting the end point of one open contour to the start point of the next.

Contours that do not have start and end points are considered *closed*. Examples of closed contours are circles, rectangles, and closed polygons. Infinite contours, such as lines, do not have start and end points and are, therefore, also closed.

Regions are shapes that partition the plane into an *inside* and an *outside*. Points exactly on the curve are neither inside nor outside. The inside has a finite area and extent, and the outside is the complement of the inside. Neither set is required to be connected. Regions include all non-intersecting closed contours as well as the annulus classes.

Methods Specific to Open Contours

The **startPoint()**, **endPoint()**, **startAngle()**, **endAngle()**, **tangentRotation()**, and **windingAngle()** methods are applicable only to open contour shapes. They all throw *ccShapesError::NotOpenContour* if invoked on shapes that are not open contours.

These methods return the following information for open contour shapes:

- The **startPoint()** and **endPoint()** methods return the starting and ending points of an open contour.
- The **startAngle()** and **endAngle()** methods return the starting and ending angles of an open contour.
- The **tangentRotation()** method returns the net angle through which the tangent vector rotates as the curve is traversed from the start point to the end point. This angle is used to determine the handedness of a **ccContourTree**.
- The **windingAngle()** method returns an angle that is relative to a given point. If *q* is a point that moves along the open contour shape, **windingAngle(p)** returns the net angle through which the segment *pq* rotates as *q* moves from the start point to the end point. This angle is used to determine whether a point is inside or outside of a closed **ccContourTree**.

Methods Specific to Regions

The **within()** and **isRightHanded()** methods are applicable only to region shapes. They all throw *ccShapesError::NotRegion* if invoked on shapes that are not regions.

These methods return the following information for region shapes:

- The **within()** method returns true if a given point is in the inside portion of a region. See also *Determining Point Containment in Regions* on page 353.
- The **isRightHanded()** method returns true for shapes that are right handed and false for shapes that are left handed. Handedness presumes a direction along the shape. Direction is implicitly defined for some primitive shapes.

Handedness

Non-intersecting, closed contours are either right-handed or left-handed. Some shapes have fixed handedness. For example, **ccCircle**s are always right handed. Others, such as **ccEllipse2**, can be either right or left handed, depending on the internal parameters of the object. In all cases, however, **isRightHanded()** returns true if and only if points are sampled in a right-handed manner along the curve (see *Sampling Shapes* on page 339).

The following criteria are all equivalent ways of defining handedness:

- A non-intersecting, closed contour is right- (respectively, left-) handed if and only if the tangent vector rotates through a net angle of $+2\pi$ (respectively, -2π) radians along the curve.
- A non-intersecting, closed contour is right- (respectively, left-) handed if and only if the inward normal vector (the normal pointing toward the enclosed area of the plane) is rotated $+\pi/2$ (respectively, $-\pi/2$) radians from the tangent vector.
- A non-intersecting closed polygon with vertices p_0, p_1, …, p_{n-1} is right handed if and only if the sum of the cross product of successive points (shown at right) is positive, and left handed if the sum of the cross product is negative.

  $$\sum_{i=0}^{n-1} p_i \times p_{i+1}$$

  Handedness presumes a tangent vector and, therefore, an implied direction along the contour. As closed contours do not have start and end points, the implied direction is the direction in which points are generated when the curve is sampled (see *Sampling Shapes* on page 339). Some of the criteria for defining handedness involve the signs of angles. For these, the positive direction is taken as the direction in which *x* rotates into *y*. Equivalently, the sign of the angle change as unit vector *u* rotates into unit vector *v* is the same as the sign of the cross-product $\overline{u \times v}$.
- Handedness is implicit in the shape itself and not dependent on the coordinate system in which the shape is displayed. Thus, the terms *clockwise* and *counter-clockwise* are not equivalent to handedness, as they are dependent on a coordinate system. If the coordinate system is restricted, however, clockwise and counter-clockwise do become equivalent to handedness. For example, in a coordinate system in which the x-axis is pointing right and the y-axis is pointing up, a non-intersecting, closed contour is right handed if and only if it traces around the enclosed area in a counter-clockwise direction, and left handed if it traces the enclosed area in a clockwise direction.

Modifying Shapes

The **mapShape()**, **reverse()**, **clip()**, and **decompose()** methods are all used to generate modified shapes.

- **mapShape()** maps a shape by an affine transform. This is similar to the **map()** method of primitive shapes. **mapShape()** returns a pointer to a new shape that is in many cases of the same class as the original shape. In some cases, however, the returned shape is of a different class. For example, for a **ccGenPoly** this method returns a **ccContourTree**.
- **reverse()** returns a new contour that describes the same locus of points as the original traversed in the opposite direction. Thus, the start point and end point are swapped in the reversed shape, and the implied direction is reversed.

- **clip()** clips a shape against a rectangle or convex polygonal region. Clipping against general (non-convex) polygons is not supported. The shape being clipped is not required to be convex. See also *Clipping Shapes* on page 338.
- **decompose()** decomposes a shape into its component parts. If the shape is already decomposed, such as is the case for line segments, ellipse arcs, and Bezier curves, a clone of that shape is returned. See also *Decomposing Shapes* on page 338.

Clipping Shapes

Clipping is done differently depending on whether the shape being clipped is to be treated as a contour or a region.

- **Contour clipping** involves returning only the portions of a contour within the clipping region. A single contour may give rise to an arbitrary number of individual contours when clipped. Any point on a clipped contour must have been a point on an original contour. If the original contour bounded a region of the plane, information regarding this region may be lost upon contour clipping.
- **Region clipping** preserves relationships among regions. Clipping a region gives rise to a set of new shapes that are always regions themselves. The contours bounding such regions may come from the contours bounding the original region, but they may also come from portions of the clipping polygon itself.

Region clipping is more complex then contour clipping. The only **ccShape**s that are clipped as regions are **ccRegionTree**s. To clip a primitive region shape, such as a **ccCircle**, as a region, you must first create a **ccRegionTree** from the primitive shape and then clip that. See also *Clipping Region Trees* on page 353.

Decomposing Shapes

Decomposed shapes contain only line segments, ellipse arcs, and Bezier curves. These primitives may be arranged into **ccGeneralShapeTree**s and **ccContourTree**s. A decomposed shape describes the same locus of points as the original shape, and contains the same connectivity and tangent direction information. For example, a **ccRect** can be decomposed into four line segments grouped into a right-handed **ccContourTree**.

Decomposition is a many-to-one transformation. It is not possible to infer from the decomposition alone the structure of the original shape.

Shapes for which **isEmpty()** is true or **isFinite()** is false decompose into **ccGeneralShapeTree**s or **ccContourTree**s without any children. Points decompose into zero-length line segments.

Sampling Shapes

The **sample()** method provides a way to enumerate points and tangent directions along the contour of a shape. An individual sample comprises a point on the shape, and possibly the associated tangent vector at that point. Parameters control what to do at places where the tangent vector is not well defined, such as at the corners of polylines.

Several algorithms use the **sample()** method to process shapes generically, including:

- Synthetic PatMax training
- Synthetic rendering
- Geometric fitting

Because of the complexity of the sampling operation, two auxiliary classes are used in the interface to the **sample()** method:

- **ccSampleParams** specifies details of how the sampling should be performed (see *Sample Parameters* on page 340).
- **ccSampleResult** stores the results of the sampling operation (see *Sample Results* on page 340).

Both of these classes are defined within the **ccShape** class, however they are not data members of **ccShape**.

The following is an example of how to set your own tolerance and spacing parameters for the **sample()** operation:

```
// First, instantiate a ccSampleParams object
// (see also Sample Parameters on page 340)
ccShape::ccSampleParams params;
params.tolerance(0.1);
params.spacing(2.0);

// Next, instantiate a ccSampleResult object
// (see also Sample Results on page 340)
ccShape::ccSampleResult result;

// Instantiate a shape and sample it using the params and result
// objects you just created
ccCircle circle(cc2Vect(0.0, 0.0), 3.0);
circle.sample(params, result);

// Now you can query the result object, for example to determine
// the number of distinct chains in the result
c_Int32 num = result.numChains();
```

Sample Parameters

The following parameters are part of a **ccSampleParams** object:

- The *tolerance* parameter sets a maximum distance by which the approximating polygon, formed by connecting the sampled positions with line segments, may deviate from the true shape. This is useful for adaptively controlling sampling along curved shapes.
- The *spacing* parameter sets a maximum distance between successive sample points. This is an upper bound on the arc length along the shape between two sample points.
- The *max points* parameter specifies a cap on the number of sample points allowed along a single curved component of a primitive shape. If the number of sample points required to meet the tolerance and spacing bounds exceeds this cap, the **sample()** operation throws an exception.
- The *compute tangents* parameter specifies whether to compute tangent vectors to the shape at each of the sample points.
- The *duplicate corners* parameter specifies how to sample points and tangents where two distinct components of a shape are joined, such as at the corner of a rectangle.

Sample Results

The results of a **sample()** operation are returned in a **ccSampleResult** object. You can query this object to determine the sampled points and tangents along the shape using the **positions()** and **tangents()** methods of the **ccSampleResult** object.

If a shape comprises multiple contours, as in an annulus, the returned sample result structure separates the samples from the individual contours into *chains*. The *breaks* vector of the result object describes how the composite *positions* and *tangents* vectors are partitioned into individual chains: each element of *breaks* indexes the starting element of a distinct chain in the *positions* and *tangents* vectors. Thus, the size of the *breaks* vector equals the number of distinct chains. This scheme facilitates effective preallocation of storage for position and tangent samples.

The *closedFlags* vector of the result object always has the same number of elements as the *breaks* vector. The *i*th element of *closedFlags* indicates whether the chain indexed by the *i*th element of *breaks* is an open or closed chain.

Perimeter Positions

Perimeter positions specify arbitrary and unique points along a **ccShape**. Perimeter positions are implemented through the **ccPerimPos** class, which represents a perimeter position on a shape as specified by a distance (arc length) along the shape from the *first point* on the shape.

Both the *first point* on a shape and the manner in which the shape is traversed, as a function of arc length, are consistent with the way in which **sample()** operations work on the shape. That is, the first point is the point at which a **sample()** operation would begin, and shape traversal is defined by the ordering of the sample chains and of the points within those chains along the perimeter of the shape. For open contours, the first point is the point returned by **ccShape::startPoint()**.

A **ccPerimPos** object defines a position on the perimeter of a particular shape instance. It is not valid for any other shape, whether obtained from the original through mapping, reversing, clipping, decomposing, or any other type of modification.

It is possible to specify points along an infinite shape with perimeter positions. An infinite primitive shape defines a canonical reference point and tangent direction, both required by perimeter positions. For infinite primitives only, the arc length component of a perimeter position is a signed value, specifying a position relative to the reference point. For all finite shapes, the arc length component of a perimeter position is non-negative. Some operations are possible on infinite primitive shapes, but not on infinite non-primitive shapes. These are noted where appropriate.

Perimeter ranges can be used to delineate arbitrary sections of a shape. Perimeter ranges are implemented through the **ccPerimRange** type definition.

Perimeter and Parameterization Functions

The perimeter and parameterization functions of **ccShape** include the following:

- **perimeter()** returns the perimeter of a shape.
- **nearestPerimPos()** returns the nearest perimeter position on a shape to a given point, as determined by **nearestPoint()**.
- **perimPoint()** returns the point along a shape described by a given perimeter position.
- **perimPointAndTangent()** returns the point along a shape described by a given perimeter position, and the tangent angle at that point.
- **subShape()** returns the shape describing a portion of a shape over a given perimeter range.

The perimeter and parameterization functions are used to inspect the boundaries of objects in an image. See the *Boundary Inspection Tool* chapter of the *CVL Vision Tools Guide* for more information.

Shape Information Objects

The perimeter and parameterization functions of **ccShape** require certain information about a shape to be explicitly precalculated. This ensures that repeated calls to those functions on an unchanging shape are maximally efficient.

The information stored in **ccShapeInfo** is specific to a particular shape instance. It is not valid for any other shape, whether obtained from the original through mapping, reversing, clipping, decomposing, or any other type of modification.

A **ccShapeInfo** object can be constructed for any primitive shape, finite or infinite, but not for a non-primitive shape. Similarly, shape operations that require the information stored in a **ccShapeInfo** object are possible on primitive shapes, finite or infinite, but not on non-primitive shapes.

Shape Hierarchies

Shape hierarchies provide a means to deal with complex geometry, such as that created with CAD programs. It is often desirable to access particular submodels of a large geometric model, and hierarchies provide the means to do this. Examples of operations that can be applied to submodels are:

- Mapping by an affine transform
- Training a vision tool
- Rendering (rasterizing)
- Selection through a GUI

Hierarchies can also be used to make certain algorithms more efficient. For example, computing the closest point to a complex geometric model is faster if the geometry is arranged in a bounding box hierarchy.

Closed curves may define regions of the plane, these regions may have holes defined by other closed curves, these holes might have solid islands within them, and so on. Hierarchies provide a way to capture the salient relationships among nested regions, and to facilitate functionality such as inside/outside queries.

Hierarchies also provide a way to express the results of certain operations on primitive shapes. For example, clipping an annulus may generate a different annulus, or a C-shaped object without holes, or several disconnected regions. In general, clipping a single shape can lead to an arbitrary number of new shapes, possibly of different types than the original. Hierarchies provide a way to model the significant geometric modifications that can result from operations such as clipping. Mapping by an affine transform is another operation that may lead to a hierarchy of shapes when applied to a primitive shape.

Finally, shape hierarchies provide a general framework to represent boundaries and regions generated, consumed, and returned by vision tools. Having a common representation for describing boundaries and regions makes it easier to write code that uses these tools.

Shape Trees

A **ccShapeTree** is basically nothing more than a vector of pointers to **ccShape**s, its children. There are several different kinds of trees, based on additional requirements placed on valid structures in certain uses. For example, a **cc2Point** is not a region and therefore does not belong in a shape tree that represents a nested hierarchy of regions.

Because of differences like these, **ccShapeTree** is actually an abstract base class with at least three derived classes. The current implementation includes the following derived shape tree classes:

- **ccGeneralShapeTree** is essentially a non-abstract synonym for the base class **ccShapeTree**. It adds no functionality or structure beyond that provided by the base class. This class supports completely arbitrary shape hierarchies, with arbitrary relationships between the components.
- **ccContourTree** describes connected contours comprising different types of connectable shapes. Line segments, arcs, open polygons, splines, and the like can be connected into a single composite contour, which itself may be open or closed. This class is a more general form of **ccGenPoly** and **cc2Wireframe**. The added generality comes at the price of less structure, and therefore less powerful manipulation methods. This class provides no native methods for vertex rounding, for example.
- **ccRegionTree** describes hierarchies of regions of the plane. Solid regions may have holes, these holes may have solid regions within them, and so on. This class supports some region-specific operations, such as synthetic rendering, and a query that indicates whether a point is inside or outside of the region.

Const Only Access to Non-Root Shapes

Once a shape is inserted into a shape hierarchy as a child of some other shape, CVL provides only `const` access to it. This makes it possible to maintain certain constraints over shape hierarchies. For example, the primitive shapes in a **ccContourTree** must be open contours. If non-`const` access were provided, the application could close the contours of individual primitives, violating this constraint placed on shapes in **ccContourTree**s. Note that `const`-only access does not restrict what trees can be built, it only means that trees should be built in bottom-up fashion. You can gain non-`const` access to an internal node of a tree by copying a subtree, so that the internal node becomes the root of a new tree.

Copying and Transforming Trees with Shared Children

Shape trees and classes derived from them are implemented with internal pointer handles to children. These pointer handles are `const` (specifically, **ccShapePtrh_const**) so it is impossible to modify children objects through their parent, even if the parent is not `const`. Therefore, it is not possible to modify an object once it has been placed into the tree.

Copying and cloning operations are always shallow copies when performed on children. The copy is achieved by copying pointer handles, and after the copy the original and the copy point to the same set of children objects. This means that trees cannot be

modified in place. Instead, a tree modification is achieved by creating a new tree using parts of the old one, possibly rearranged and modified. Every modification of a tree node generates a new tree node. Nodes that do not change are simply copied or cloned. Being shallow, copying and cloning are fast and efficient operations.

For example, consider the following algorithm for mapping an entire tree by an affine transform. This function takes an original tree *T* and an affine transform *X*, and returns a new tree *T'* that is the transform of the original.

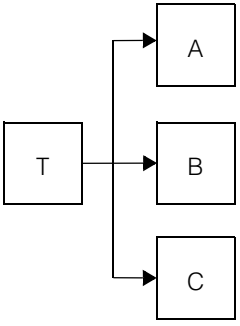
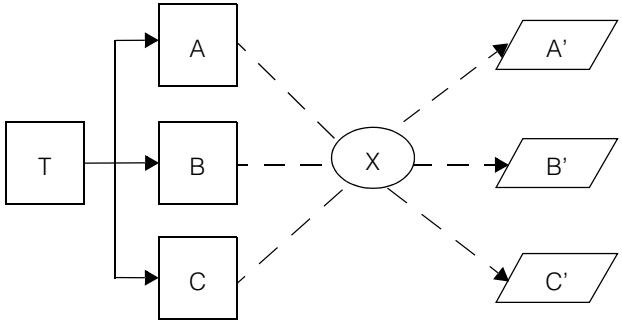


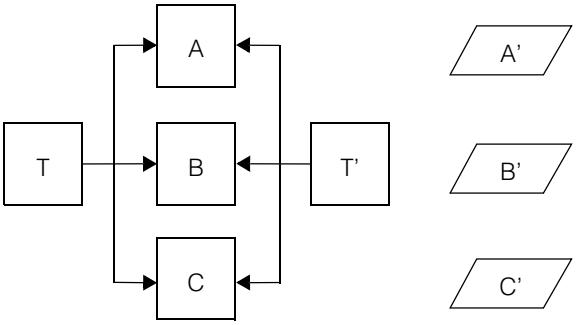
Figure 56. Tree T with three children

The steps are as follows:

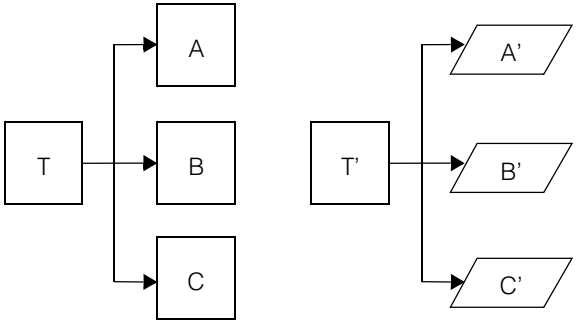
1. Recursively map the children of *T* by *X*, obtaining a set of new, transformed children.



2. Set *T'* to be a clone of *T*. This is a fast operation as it involves a shallow copy. After this operation, *T* and *T'* share the same children.



3. Replace the children of *T'* with the set of transformed children computed in step 1. The children of *T* remain unchanged.



4. Return *T'*.

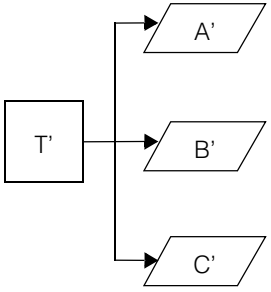


Figure 57. Mapped tree T'

Assumption of Correct Topology

A closed **ccContourTree** is assumed to be a region shape; no check is made to verify that the contour does not self-intersect. A **ccRegionTree** is built from individual region shapes, and it is assumed that the children of a **ccRegionTree** describe regions lying completely in the interior of their parent. The latter assumption stems from the fact that a child of a solid **ccRegionTree** describes a hole within that solid region, and the child of a hole **ccRegionTree** describes a solid within that hole region. These self-intersection and containment properties are complicated to check, and the added checking would excessively slow down the common case of correct usage. Therefore, the onus is on the user application to use the class sensibly.

ccShapeTree Interfaces

The **ccShapeTree** base class provides methods for building and querying the hierarchy of shapes, without regard to the kinds of shapes that are in the hierarchy or how they are related.

Querying the Shape of the Tree

The **numChildren()**, **size()**, and **height()** methods are non-virtual, `const` functions that return standard information about the shape of the hierarchy rooted at this **ccShapeTree**.

Accessing and Manipulating Children of a Shape Tree

The following methods allow you to access and modify the set of children of a **ccShapeTree**:

- **child()** provides *const*-only access to a specified child of a given **ccShapeTree**. No non-`const` version is provided.
- **insertChild()** inserts one child at a given index in the tree, while **insertChildren()** inserts a contiguous set of children at a given index.
- **addChild()** adds one child at the end of the shape tree, while **addChildren()** adds a contiguous set of children at the end of the shape tree.
- **replaceChild()** replaces one child at a given index in the shape tree, while **replaceChildren()** replaces all children of the shape tree.
- **removeChild()** deletes one child at a given index in the shape tree, while **removeChildren()** removes all children of the shape tree. The latter may cause derived shape trees that are required to have at least one child to throw *ccShapesError::BadGeom*.

The above methods are all virtual. Derived classes may impose their own rules governing what constitutes a valid set of children.

Flattening Trees

The **flatten()** method returns a flattened version of a shape tree. A flattened tree contains all of the primitive shapes of the original tree, including those defining the boundaries of region trees. The primitives in a flattened tree are all arranged as direct children of the root node, which is the only shape tree node in the flattened tree. If the original tree is a contour tree, then the returned tree is also a contour tree. Otherwise, the returned tree is a general shape tree.

The order of primitives in a flattened tree is the order in which they are encountered in a standard depth-first, pre-order traversal of the tree. The distinction between pre- and post-order traversals is significant only when primitives are present as internal nodes as is the case, for example, with region trees.

ccGeneralShapeTree Interfaces

The **ccGeneralShapeTree** class is the simplest concrete class derived from **ccShapeTree**. It adds only a single method beyond those of the base class: **connect()** (see *Connecting Shapes in a General Shape Tree* on page 348).

ccGeneralShapeTree is useful in cases where it is necessary to represent application-specific hierarchies. It is also used as a general container for results returned by certain methods of the other **ccShape** classes. For example, when a circle is clipped against a convex polygonal region, the result is an arbitrary number of disjoint circular arcs. This result is neither a **ccContourTree** nor a **ccRegionTree**, and must therefore be returned as a **ccGeneralShapeTree**. The **ccGeneralShapeTree** class also serves as the container for imported DXF files as there are no implied relationships among the individual components.

Connecting Shapes in a General Shape Tree

The **connect()** method of **ccGeneralShapeTree** connects the immediate children of a general shape tree into longer contours. Flattening (see *Flattening Trees* on page 348) and connecting the resulting children shapes is necessary, for example, after importing a DXF file into a shape tree. DXF files often contain no explicit information regarding the connectivity of shapes. For example, a rectangle might be represented as four line segments whose endpoints happen to be coincident. The flattening and connecting operations interpret coincident endpoints to make the connections explicit. In the case of a rectangle, flattening and connecting create a contour tree containing the four line segments. PatMax can use the resulting explicit information when training a pattern, which generally results in better alignment performance. See the *CAD File Import* chapter of this *User's Guide* for more information on importing DXF files into CVL.

ccContourTree Interfaces

The **ccContourTree** class represents a single contour built by linking an arbitrary number of open contours together. **ccContourTree** generalizes the approach of **ccGenPoly** and **cc2Wireframe**. Whereas these classes allow you to link together circular arcs, line segments, and elliptical arcs in a restricted sense, **ccContourTree**s allow you to link together these shapes in unrestricted ways. Also, you can insert open **ccPolygon**s, open **ccGenPoly**s, and open **cc2Wireframe**s into a **ccContourTree**.

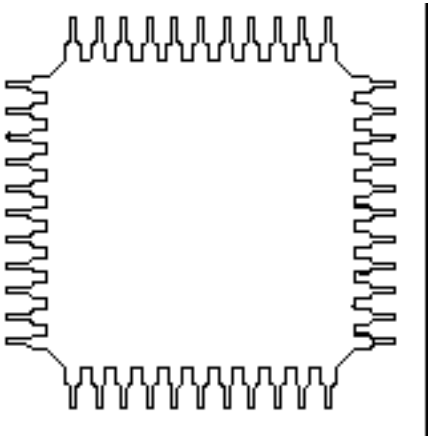
There is an implied ordering of primitives in a **ccContourTree**, defined as the order in which the primitives, which are necessarily all leaf nodes, are encountered during a standard traversal of the tree. Two primitives are adjacent if they are adjacent in the ordered list of primitives.

ccContourTree assumes that the end point of one child is approximately equal to the start point of the next child. If the **ccContourTree** is closed, the end point of the last child is assumed to be approximately equal to the start point of the first child. Because of floating point inaccuracies, exact equality is not enforced. For example, decomposing a **ccGenRect** yields a valid **ccContourTree**, even though the rounded corner endpoints may not exactly coincide with the straight side end points. Small discrepancies such as these do not pose a problem. However, since coincidence of endpoints is not enforced, it is possible, though not advised, to create **ccContourTree**s with gaps between the endpoints of adjacent children that are large relative to the scale of the children themselves. This may produce unpredictable results, as many **ccContourTree** methods (for example, **sample()** and **isRightHanded()**) assume that gaps are of negligible size.

ccContourTrees may be open or closed. Closed **ccContourTree**s have an implied connection from the end point of the last primitive to the start point of the first primitive, while open ones do not. The open/closed state is maintained in a boolean flag, which is the only member that **ccContourTree** adds beyond those in the **ccShapeTree** base class. A **ccContourTree** is considered to be an open contour shape if and only if it is open; it is considered a region shape if and only if it is closed.

Why maintain an ordered sequence of primitives in a hierarchy rather than as a simple list or vector? A list or vector representation is reasonable for contours, but there are at least three reasons for using hierarchies:

- The hierarchy machinery is already needed for **ccRegionTree**s and **ccGeneralShapeTree**s, so few modifications are needed for **ccContourTree**s.
- Many contours, particularly models, are hierarchical in nature. For example, the boundary of a chip model may be broken down into rows of pins, and each row may be broken down into individual pins (see figure below). Hierarchies make such models easier to construct, and easier to interpret after they have been modified, for example through affine transforms or clipping.



- Hierarchies provide a framework for more efficient algorithms. For example, a function to return the closest point on a complex contour to a given point can be made more efficient if the contour is arranged in a bounding box hierarchy with bounding box information cached at internal nodes of the hierarchy. A bounding box hierarchy shape can be derived from **ccContourTree**.

Querying the State of a Contour Tree

The **isClosed()** method gets or sets the state of the closed flag of a **ccContourTree**. This controls whether the last primitive descendant of the **ccContourTree** is connected to the first primitive descendant through a bridge connection.

Region Tree Queries and Operations

The **ccRegionTree** base class provides the following predicates for identifying properties of and manipulating region trees:

- The **isHole()** query indicates whether the root boundary of a region tree encloses a solid or hole region.
- The **isRoot()** query indicates whether a region tree is at the root of a hierarchy (has no parent).
- The **boundary()** method sets or gets the boundary of a region tree. The setter does not change the solid/hole state of the boundary. Any shape that is a region, other than another **ccRegionTree**, can be a valid boundary shape.
- The **flip()** method returns a **ccRegionTree** that is identical to the original except that its solid/hole state is flipped.

Manipulating Children of a Contour Tree

The following methods of **ccContourTree** override the virtual methods declared in **ccShapeTree** to ensure that only open contour shapes are added as children. If an attempt is made to add another type of shape as a child, these methods throw. This implies that a **ccContourTree** can also have other **ccContourTree**s as children, but only if they are open.

- **insertChild()** inserts one child at a given index in the tree, while **insertChIdren()** inserts a contiguous set of children at a given index. The children must be open contours.
- **addChild()** adds one child at the end of the shape tree, while **addChildren()** adds a contiguous set of children at the end of the shape tree. The children must be open contours.
- **replaceChild()** replaces one child at a given index in the shape tree, while **replaceChildren()** replaces all children of the shape tree. The children must be open contours.

ccRegionTree Interfaces

The **ccRegionTree** class describes a hierarchy of nested regions. Each region is bounded by a region shape, and is classified as a solid region or a hole region. Solid regions may have hole regions within them, which are stored as children of the solid region, and vice versa. Figure 58 shows an example of a region shape, with the solid portions shaded grey, and the corresponding region hierarchy.

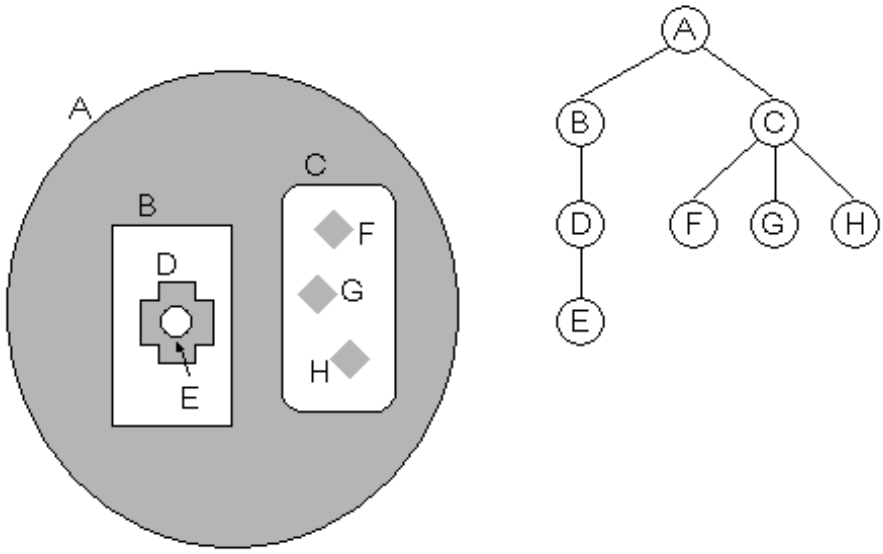


Figure 58. Region shape and associated region hierarchy

The inside/outside status of subregions alternates along any path from the root to a leaf of the tree. The only allowable children of a **ccRegionTree** are other **ccRegionTree**s. Unlike all of the other hierarchical shapes, **ccRegionTree**s must associate actual shapes with each internal node of the hierarchy, not simply with each leaf node. These shapes describe the boundary corresponding to each node. The boundary may be any region shape other than a **ccRegionTree**. Thus, either a primitive region shape or a closed **ccContourTree** is a valid boundary. This boundary shape is an added data member of **ccRegionTree**, and is required to construct one.

Multiple disjoint solid regions can be represented in a single **ccRegionTree** by enclosing all of them with a hole region forming the root of the hierarchy. This phantom hole region is not included in sampling or geometric operations, such as **boundingBox()**.

Accessing and Manipulating Children of a Region Tree

The following method of **ccRegionTree** shadows, rather than overrides, the following non-virtual method declared in **ccShapeTree**:

- **child()** provides *const*-only access to a child of a **ccRegionTree** at a given index. This method returns a more derived reference to a region tree child, which is always another region tree. This can be used anywhere there is a reference to a **ccShape**.

The following methods of **ccRegionTree** override the virtual methods declared in **ccShapeTree** to ensure that only other region trees are added as children. If an attempt is made to add some other type of shape as a child, these methods throw. This implies that a **ccRegionTree** can only have other **ccRegionTree**s as children.

- **insertChild()** inserts one child at a given index in the tree, while **insertChIdren()** inserts a contiguous set of children at a given index. The children must be region trees.
- **addChild()** adds one child at the end of the shape tree, while **addChildren()** adds a contiguous set of children at the end of the shape tree. The children must be region trees.
- **replaceChild()** replaces one child at a given index in the shape tree, while **replaceChildren()** replaces all children of the shape tree. The children must be region trees.

Determining Point Containment in Regions

The **within()** method of **ccRegionTree** overrides the method of the **ccShape** base class to descend into the hierarchy, level by level, until it can show that the smallest region enclosing a given point is a solid or a hole. This method returns true if the point lies within a solid region, and false if the point lies within a hole region.

Clipping Region Trees

The **clip()** method of **ccRegionTree** performs region clipping, which produces a set of new **ccRegionTree**s and not the open contours produced by contour clipping.

Contour clipping produces a set of contours inside the clipping region. Information regarding whether the contours originally came from the same continuous boundary, and their ordering along such a boundary, is lost. Contour clipping is appropriate for applications such as PatMax training, where the contours themselves, and not the area that they enclose, are significant.

Region clipping produces a set of regions that are bounded by closed contours, inside the clipping region. The regions are produced by appropriately joining the contours produced by contour clipping, and adding new portions to the clipping region boundary when necessary. Region clipping is appropriate for applications such as rasterization, where the area enclosed by the contours is significant.

For example, consider a region that has been clipped into a set of contours as shown on the lower left in Figure 59.

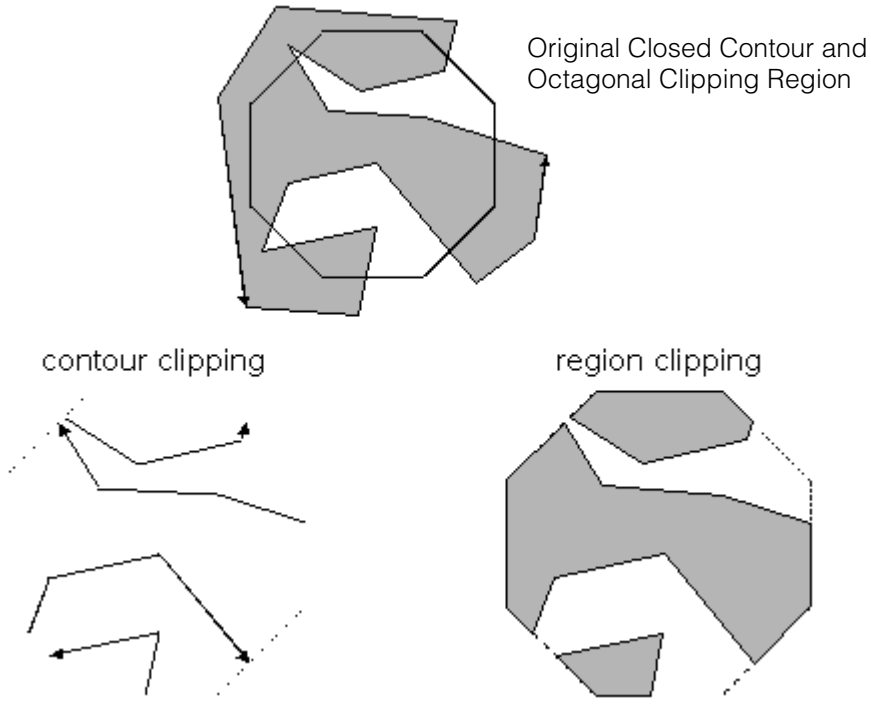


Figure 59. Region clipping a closed contour

Region clipping depends on children having the opposite handedness to their parents. If the boundary of a solid region is right-handed, the boundary of any holes within it must be left-handed. If the handedness of a child is the same as that of its parent, then any open contours produced from contour clipping are reversed using the **reverse()** method before the children are linked.

Figure 60 shows an example of region clipping. The original regions are synthetically rendered in dark grey, while the clipped regions are synthetically rendered in light grey.

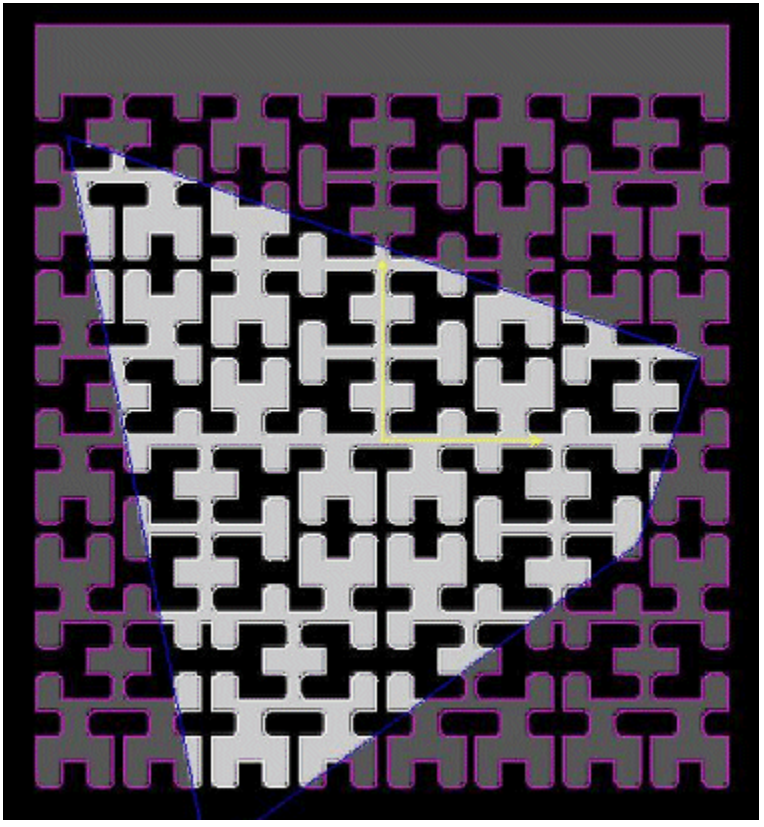


Figure 60. Result of region clipping a complex region

See also *Clipping Shapes* on page 338.

Rasterizing Shapes

CVL provides global functions to rasterize shapes into a pel buffer. Rasterizing is different from drawing in the following ways:

- A rasterized shape is filled in. That is, more than just its boundary is shown.
- A rasterized shape is converted into a form that may be displayed on a grid of pixels. This can produce digitization errors near the edges of the shape because pixels have finite size.

The **cfRasterizeContour()** global function can rasterize any type of **ccShape**-derived object into a pel buffer, including **ccRegionTree**s, **ccContourTree**s, and **ccGeneralShapeTree**s. The **cfRasterize()** global function rasterizes only **ccRegionTree**s into a pel buffer.

Rasterizing Contours

The **cfRasterizeContour()** global function rasterizes the contour of a shape into a pel buffer. Contour pixels are first sampled with the **sample()** method, and the returned pixels are rendered into the pel buffer in the specified grey level and thickness. This function creates a circular stencil with a diameter of the specified thickness centered around each sampled point on the pel buffer and slides it along the line between this point and the next sampled point. Any pixels that are touching this stencil are assigned the foreground color, while remaining pixels are not changed. Any pixels that are outside the pel buffer window and touching the circular stencil are clipped. See the *CVL Class Reference* for more information.

Rasterizing Regions

The **cfRasterize()** global function rasterizes a **ccRegionTree** into a pel buffer. Solid pixels are rendered in the specified grey level while hole pixels are left unchanged. Pixels that are on the boundary between solid and hole regions are calculated according to the *boundaryFillMode* flag setting. Phantom holes (that is, hole regions that are not children of some enclosing solid region) are not rasterized.

See Figure 60 on page 355 for an example of the result of invoking **cfRasterize()** on a complex shape. See the *CVL Class Reference* for more information on the **cfRasterize()** function.

Shape Geometries

The following sections give a high-level overview of the **ccShape**-derived geometric shapes implemented in CVL.

Points, Lines, and Line Segments

The **cc2Point** class implements a point shape. The **ccFLine**, **ccLine**, and **ccLineSeg** classes implement lines, directional lines, and line segments, respectively, in CVL.

cc2Point

The **cc2Point** class represents a single point as (*x*,*y*) coordinates in a two-dimensional Cartesian plane. **cc2Point** implements the **ccShape** base class methods for points.

Note **cc2Point** replaces **ccPoint**, which is now deprecated.

The **cc2Vect** class is also often used to represent a point in (*x*,*y*) coordinates. As **cc2Vect** does not inherit from **ccShape**, it does not implement the shape functionality required by vision tools. **cc2Vect** objects also do not contain the extra data members associated with the base classes of **ccShape** and, as a result, may use significantly less memory (in a relative sense) than **cc2Point** objects.

The **ccPointSet** class represents a set of points. This class also does not inherit from **ccShape**.

ccFLine

The **ccFLine** class represents a 2D line. A line has infinite length and is bidirectional. **ccFLine** implements the **ccShape** base class methods for lines.

The **ccFLine** class defines a line in θ/distance coordinates. These coordinates are interpreted as though the line contained the X axis of a coordinate frame. The *line frame* is first rotated by θ and then translated along the line-frame's new y-axis by the distance

coordinate. Thus, **distance()** is the shortest distance from the origin to the line. The distance can be positive or negative, although it can always be kept positive by adding 180° to the angle if needed.

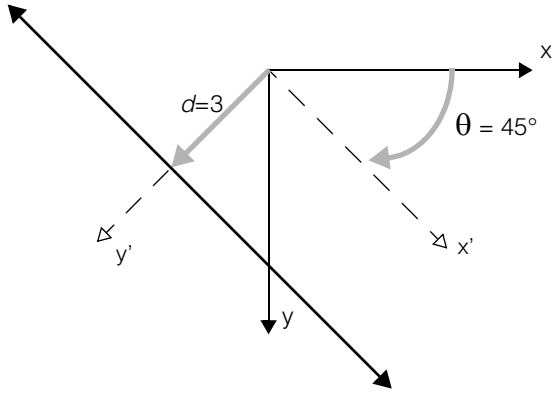


*Figure 61. **ccFLine** with rotation angle 45° and distance 3*

There are an infinite number of coordinate representations of a line. For any integer *n*, adding 360 * *n* to the *angle* does not change the line. Adding 360 * *n* + 180 to the *angle* and negating the *distance* also does not change the line.

ccLine

The **ccLine** class represents a 2D line with a given direction (in radians), passing through a given point. **ccLine** implements the **ccShape** base class methods for directional lines.

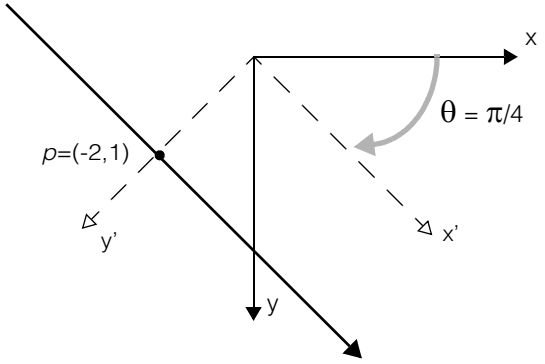


*Figure 62. **ccLine** with direction π/4 passing through point (-2,1)*

ccLineSeg

The **ccLineSeg** class represents a line segment. This class implements the **ccShape** base class methods for line segments.

You construct a line segment by specifying a start point and an end point.

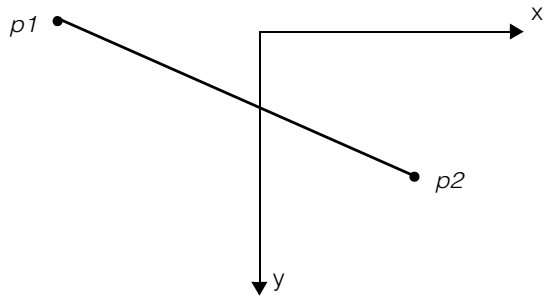


*Figure 63. **ccLineSeg** with start point p1 and end point p2*

Rectangles

The **ccRect** and **ccAffineRectangle** classes implement rectangles and affine rectangles, respectively, in CVL.

ccRect

The **ccRect** class represents a rectangle. This class implements the **ccShape** base class methods for rectangles.

You construct a rectangle by specifying an upper left corner and a size.

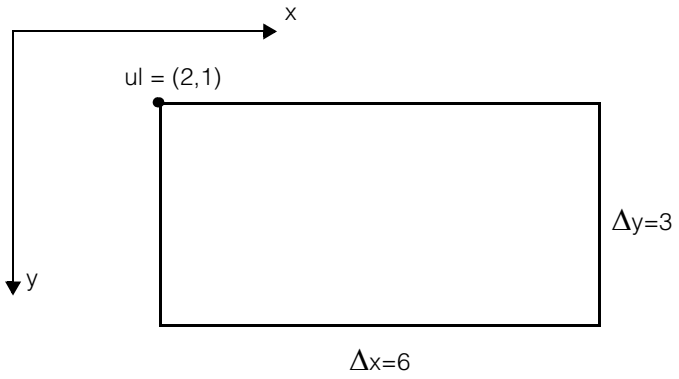


*Figure 64. **ccRect** with upper left corner at (2,1) and size 6x3*

ccGenRect

The **ccGenRect** represents a generalized rectangle. This class implements the **ccShape** base class methods for generalized rectangles.

Whereas the corners of a **ccRect** shape are always right-angled, the corners of a **ccGenRect** shape can be rounded. Furthermore, while a **ccRect** is always aligned with the coordinate axes, a **ccGenRect** may be arbitrarily oriented relative to the coordinate axes.

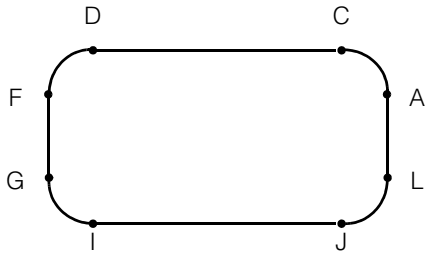


*Figure 65. **ccGenRect** with rounded corners*

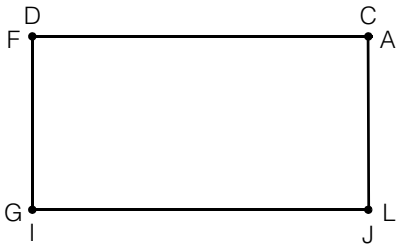


*Figure 66. **ccGenRect** with right-angled corners*

You construct a generalized rectangle by specifying any of the following:

- A center point, a 2D matrix, and a corner rounding value
- A center point, two radii, an orientation angle, and a corner rounding value
- A rectangle
- An ellipse

ccAffineRectangle

The **ccAffineRectangle** class represents an affine rectangle. This class implements the **ccShape** base class methods for affine rectangles.

An affine rectangle is a labeled parallelogram, or a quadrilateral with parallel opposite sides that are of equal length, and vertices that are uniquely labeled. You can use affine rectangles to specify regions of an image for further processing, for example, a region of pels to be projected.

Unique vertex labels define directions within the affine rectangle, much like a local coordinate system. For example, you can think of p_0 as the origin of a system where the x-axis extends through the point p_x and the y-axis extends through the point p_y. The remaining corner, p_{opp}, is diagonally opposite the origin.

You construct an affine rectangle by specifying any of the following:

- A 2D transform
- Three corner points: p_0, p_x, and p_y
- A rectangle center point, side lengths, rotation of the rectangle sides, and skew

For example, you can create an affine rectangle by specifying three corners, and then access the shape by querying the length of its sides. Each way of accessing the rectangle is different from, but mathematically related to, the others.

Another way to represent an affine rectangle is as a 2D transform that maps the unit square to the desired rectangle. The 2D transform maps the following points:

- The origin O to the point p_0
- The point (1,0) to p_x
- The point (0,1) to p_y
- The point (1,1) to p_{opp}

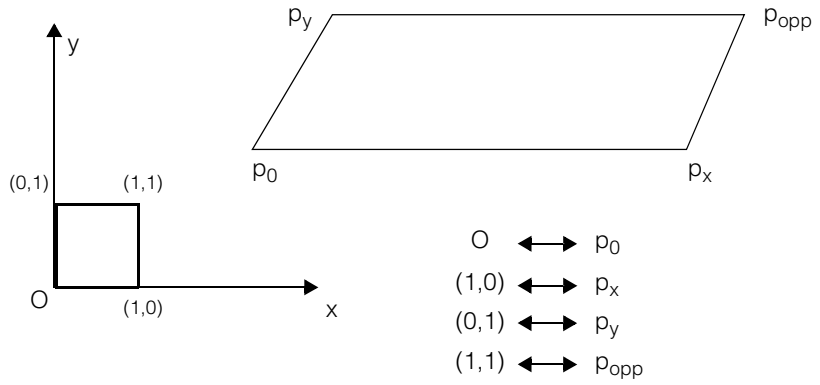


Figure 67. **ccAffineRectangle** *mapping the unit square to an affine rectangle*

See also Figure 71 on page 368 depicting an ellipse annulus section with similar mapping.

This mapping can also be interpreted as a sequence of scaling, rotation, and translation operations that transform the unit square into the affine rectangle as follows:

1. First, the dimensions of the unit square are scaled to become *xLength* and *yLength*, where *xLength* is the distance between p_0 and p_x, and *yLength* is the distance between p_0 and p_y.
2. Next, the sides of the expanded square are rotated through the angles R_x (*xRotation*) and R_y (*yRotation*). If *xRotation* is equal to *yRotation*, the shape undergoes a pure rotation. Otherwise it is skewed by an angle *K*, where:

   $K = R_y - R_x$

   K can be as large as 180°.
3. Finally, the scaled and rotated square is translated so that its origin moves to p_0. This motion also moves the other corners to the positions shown.

You will typically create an affine rectangle using a single, simple interface that is natural for your application. The tools that use the rectangle can then access it in the way best suited to that tool.

Circles and Ellipses

The **ccCircle** class represents a circle in CVL. The **ccEllipse2** and **ccEllipseArc2** classes represent ellipses and ellipse arcs, respectively, in CVL.

ccCircle

The **ccCircle** class represents a circle. This class implements the **ccShape** base class methods for circles.

You construct a circle by specifying a center point and a radius.

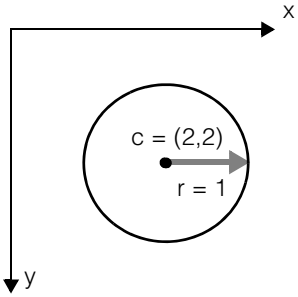


*Figure 68. **ccCircle** with center at (2,2) and radius of 1*

ccEllipse2

The **ccEllipse2** class represents an arbitrary 2D ellipse, or more precisely, an arbitrary 2D affine transformation of the unit circle (a circle with a radius of 1.0).

The **ccEllipse2** class implements the **ccShape** base class methods for ellipses.

Note **ccEllipse2** replaces **ccEllipse**, which is now deprecated.

The affine transformation definition allows the ellipse to be parameterized by an angle in a well-defined way, and either in a right-handed or left-handed fashion.

You construct an ellipse by specifying any of the following:

- The lengths of both radii, an orientation angle, a center point, whether the ellipse is right-handed, and a phase angle
- An affine transform that maps the unit circle to an ellipse
- Six constant coefficients (*a*, *b*, *c*, *d*, *e*, and *f*) that are substituted into the equation for an ellipse:

$$ax^2 + bxy + cy^2 + dx + ey + f = 0$$

ccEllipseArc2

The **ccEllipseArc2** class represents arbitrary 2D elliptical arcs. This class implements the **ccShape** base class methods for ellipse arcs.

Note **ccEllipseArc2** replaces the **ccEllipseArc**, which is now deprecated.

An elliptical arc comprises an underlying ellipse, plus a parameter range that specifies the portion of the ellipse that is included in the arc. The parameter range, specified by a start angle and an angular span, determines the range of Φ values of the underlying ellipse that lie on the elliptical arc. The phase and handedness of the underlying ellipse can affect both of these values.

The *angular span* may be positive or negative. Changing the sign of the angle range changes the direction along the ellipse in which the ellipse arc lies relative to the start point; flipping the **isRightHanded()** field of the underlying ellipse has the same effect. The angular span may have magnitude greater than 2π radians, making it possible to specify multiple revolutions around the ellipse. **ccEllipse2** provides methods for manipulating the underlying ellipse as well as the angle range of the arc.

You construct a **ccEllipseArc2** by specifying any of the following:

- An ellipse (**ccEllipse2**)
- An ellipse, a start angle, and an angular span

Annuli

The **ccAnnulus**, **ccEllipseAnnulus**, **ccEllipseAnnulusSection**, and **ccGenAnnulus** classes implement annuli, ellipse annuli, ellipse annulus sections, and generalized annuli in CVL.

ccAnnulus

The **ccAnnulus** class implements the **ccShape** base class methods for annuli.

An annulus is a ring shape. You construct an annulus by specifying either a center point and two radii, or two concentric circles.

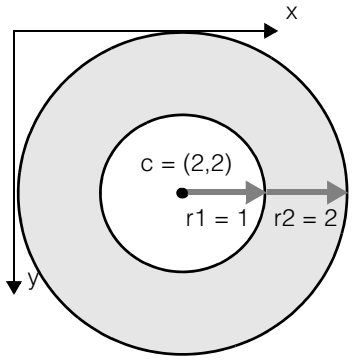


*Figure 69. **ccAnnulus** with center at (2,2), radius r1 of 1, and radius r2 of 2*

ccEllipseAnnulus

The **ccEllipseAnnulus** class implements the **ccShape** base class methods for ellipse annuli.

An ellipse annulus is an ellipse-shaped ring. You construct an ellipse annulus by specifying any of the following:

- A center point and two radii
- Two concentric ellipses with the same orientation angle and aspect ratio
- Two concentric circles

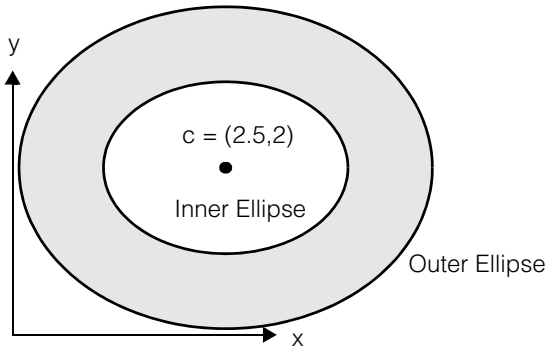


*Figure 70. **ccEllipseAnnulus** with center at (2.5,2) and inner and outer ellipses*

ccEllipseAnnulusSection

The **ccEllipseAnnulusSection** class implements the **ccShape** base class methods for ellipse annulus sections.

An ellipse annulus section can be represented as a transform that maps the unit square to the desired annular section, as shown in Figure 71. The unit square in the lower left corner is mapped to the ellipse annulus section in the upper right.

The transform maps the points of the unit square to the following points of the ellipse annulus section in the figure:

- The origin O to the point p_0
- The point (1,0) to p_x
- The point (0,1) to p_y
- The point (1,1) to p_{opp}.

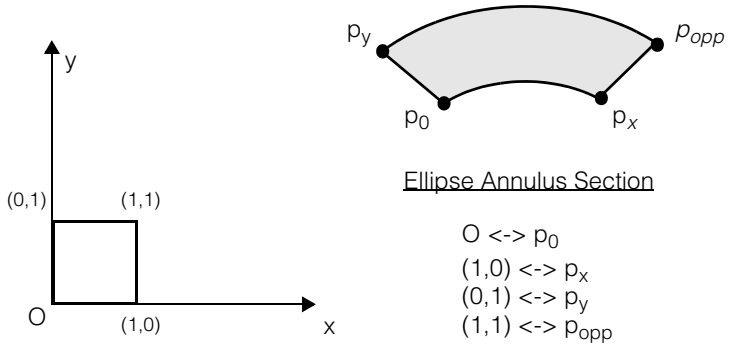


*Figure 71. **ccEllipseAnnulusSection** mapping the unit square to a section of an ellipse annulus*

ccGenAnnulus

The **ccGenAnnulus** class implements the **ccShape** base class methods for generalized annuli. A generalized annulus is an annulus formed by two concentric generalized rectangles.

You construct a generalized annulus by specifying any of the following:

- A center point, two radii, an orientation angle, and a corner rounding value
- A center point, two radii, an orientation angle, and two different corner rounding values
- Two generalized rectangles
- An annulus

Curves and Splines

CVL supports Bezier curves and cubic splines. The **ccBezierCurve** class implements Bezier curves. The **ccCubicSpline** class is an abstract base class that encapsulates common cubic spline functions. The **ccDeBoorSpline**, **ccInterpSpline**, and **ccHermiteSpline** classes are specializations of **ccCubicSpline** and implement its methods for de Boor splines, interpolated splines, and Hermite splines, respectively.

A good reference for further information on curves and splines is *Curves and Surfaces for Computer Aided Geometric Design*, Second Edition, Academic Press, 1990, ISBN 0-12-249051-7, by Gerald Farin.

ccBezierCurve

The **ccBezierCurve** class implements the **ccShape** base class methods for cubic, weighted Bezier curves.

Bezier curves have four control points. The curve lies within the convex hull of the control points, interpolating (passing through) the first and last control points. The middle control points pull the curve towards themselves: a control point with a higher weight exerts more pull than one with a lower weight.

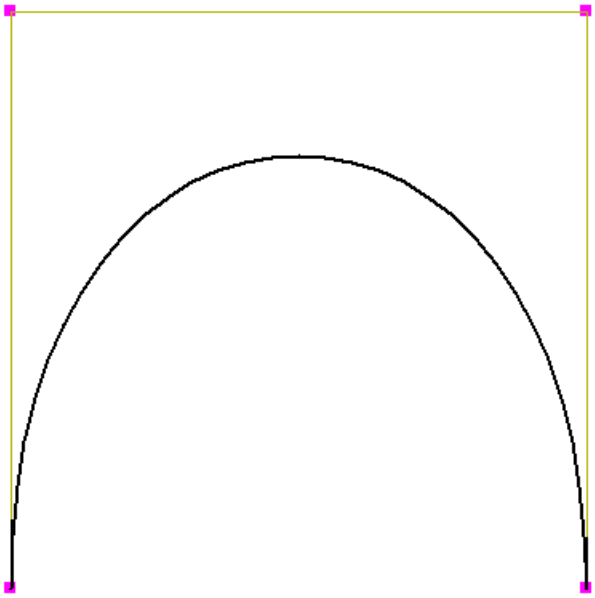


*Figure 72. **ccBezierCurve** with four control points*

Bezier curves cannot be decomposed. They form the component parts of splines. Bezier curves are always open contours, and therefore have start and end points.

Although Bezier curves may have loops and cusps, they are smooth in parameter space. They are always open contours, and it is generally not possible to join the first and last control points to form a closed curve with a smooth tangent. It is not possible to model a circle, for example, with a single Bezier curve, although this can be done with multiple Bezier curves.

Bezier Curve Parameterization

Points along a Bezier curve can be parameterized by (*x(t)*,*y(t)*), where *x(t)* and *y(t)* are cubic polynomials and *t* is in the range 0 through 1.

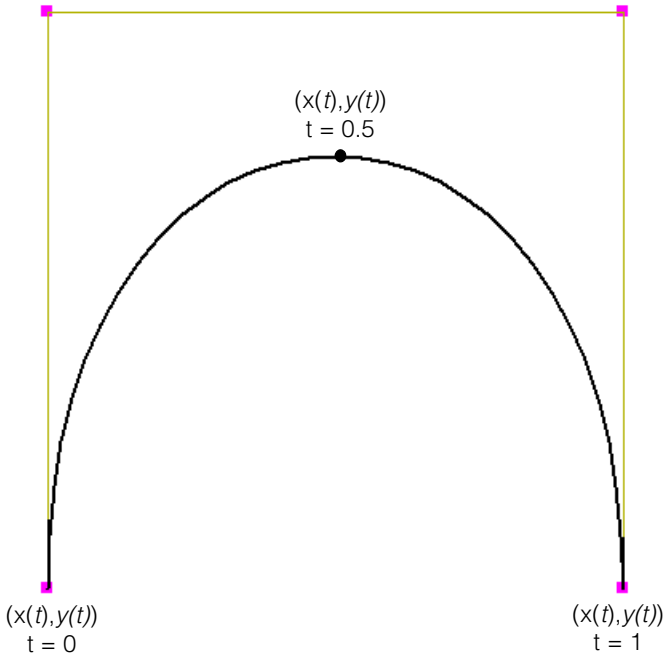


Figure 73. Parameterization of points along a Bezier curve

ccCubicSpline

The **ccCubicSpline** class is the abstract base class for all cubic splines in CVL. This class implements the **ccShape** base class methods for cubic splines.

While Bezier curves are always open, cubic splines can be either open or closed.

A cubic spline is the concatenation of several Bezier curves. When a cubic spline is decomposed using the **decompose()** method, a set of Bezier curves is returned. Cubic splines impose constraints on the component Bezier curves, reducing the number of control points to approximately one per curve. Various types of splines differ in the way they process control points and other knobs to obtain the Bezier control points.

The **ccDeBoorSpline**, **ccInterpSpline**, and **ccHermiteSpline** classes are specializations of **ccCubicSpline**. They implement de Boor splines, interpolation splines, and Hermite splines, respectively, in CVL.

Cubic Spline Parameterization

Spline parameterization determines how much relative time the point (*x(t),y(t)*) spends on each component Bezier curve. The parameterization, specified in **ccCubicSpline** as the *IntervalMode*, affects the shape of the curve when you move control points.

Different interval modes supported for cubic splines in CVL include the following:

- **Uniform mode:** intervals between control points are all equal to 1.0. This is the only parameterization that is invariant under affine transformation of the control points. This has advantages when mapping splines. This is the default parameterization.
- **Chord length mode:** based on the geometry of the control points. Intervals between control points are proportional to the distance between control points. This sometimes produces a better-shaped spline than uniform parameterization, although it can produce somewhat *loopy* splines.
- **Centripetal mode:** also based on the geometry of the control points. Intervals are proportional to the square root of the distance between control points. This produces even less loopy splines than chord length parameterization, and is often a compromise between the uniform and chord length parameterizations. This mode often produces the best-shapes splines.
- **Fixed mode:** intervals are not modified when control points are moved.

The parameterization modes described above only determine how intervals are adjusted when control points are added, deleted, or moved. Through appropriate setters, you can individually adjust each of the intervals to achieve custom parameterizations.

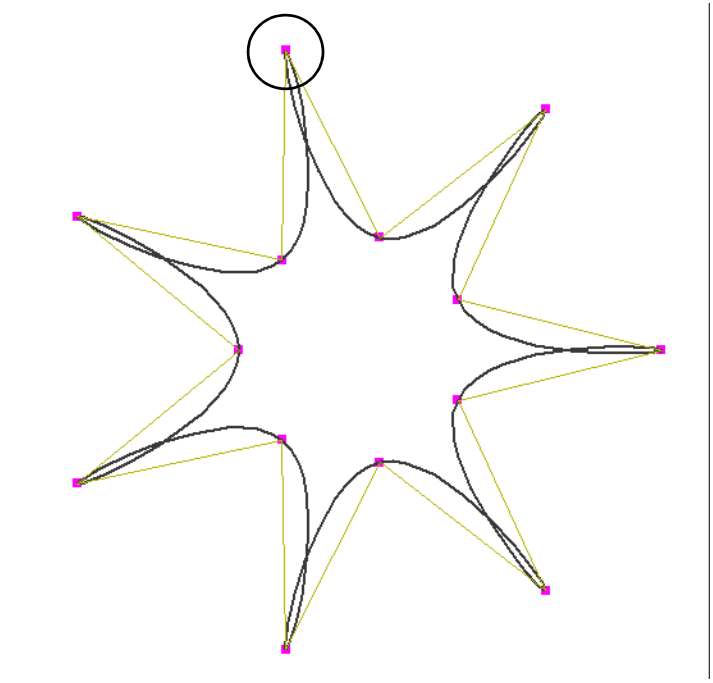


Figure 74. Cubic interpolation spline in the shape of a star

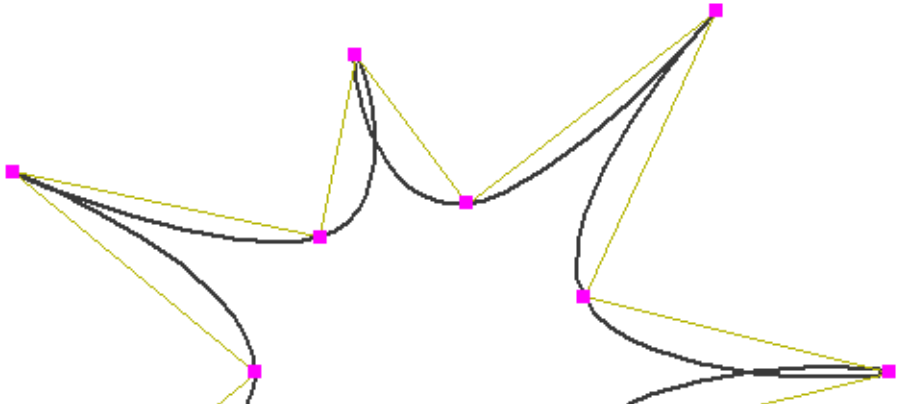


Figure 75. Effect of moving control point under uniform mode

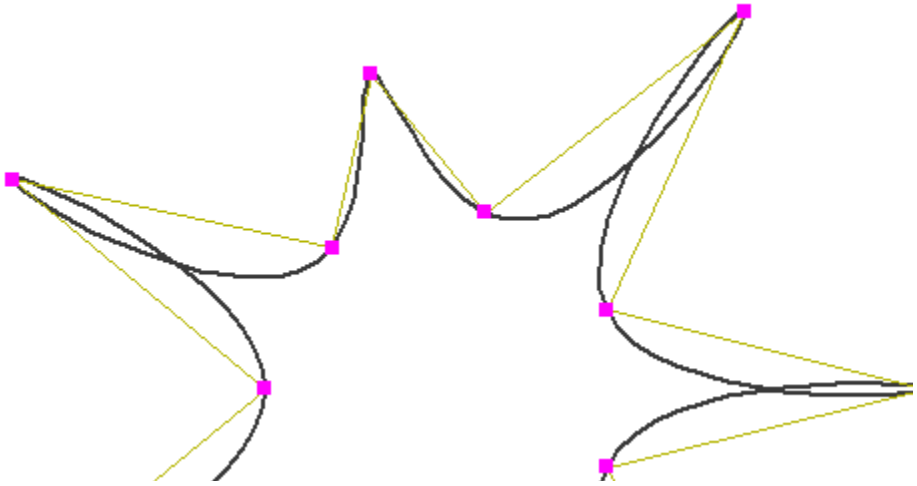


Figure 76. Effect of moving control point under chord length mode

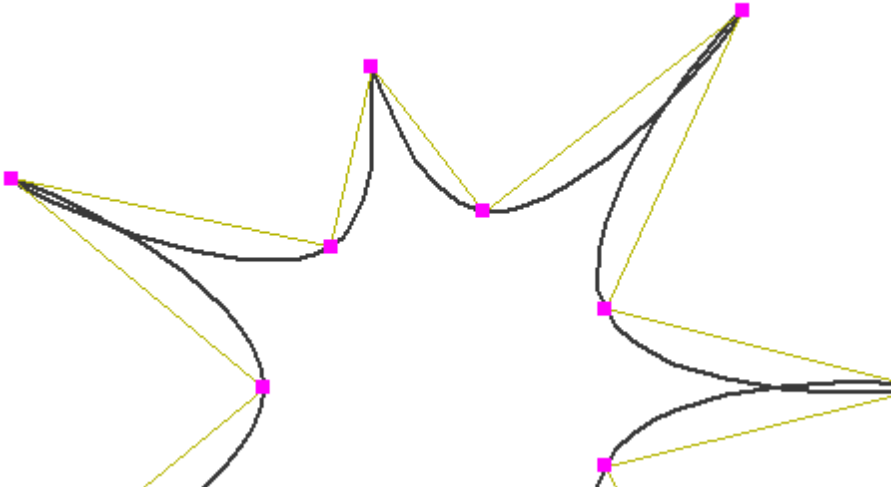


Figure 77. Effect of moving control point under centripetal mode

Figure 75 through Figure 77 show the effects of moving the control point highlighted in Figure 74 for different interval modes. The uniform mode (shown in Figure 75) produces a curve that most closely resembles the original. The chord length mode (shown in Figure 76) produces the loopiest curve. The centripetal mode (shown in Figure 77) often produces the best-shaped curve.

ccDeBoorSpline

The **ccDeBoorSpline** class implements a de Boor spline in CVL. This class implements the **ccShape** and **ccCubicSpline** base class methods for de Boor splines.

De Boor splines share many properties with Bezier curves: they maintain the convex hull property of Bezier curves, control points of de Boor splines also have weights, and the curve, if open, interpolates only the first and last spline control points. De Boor splines provide local control of the curve; that is, moving a control point affects only a small number of Bezier curves near that control point. As de Boor splines are B-splines, the junctions between Bezier curves have continuous second derivatives.

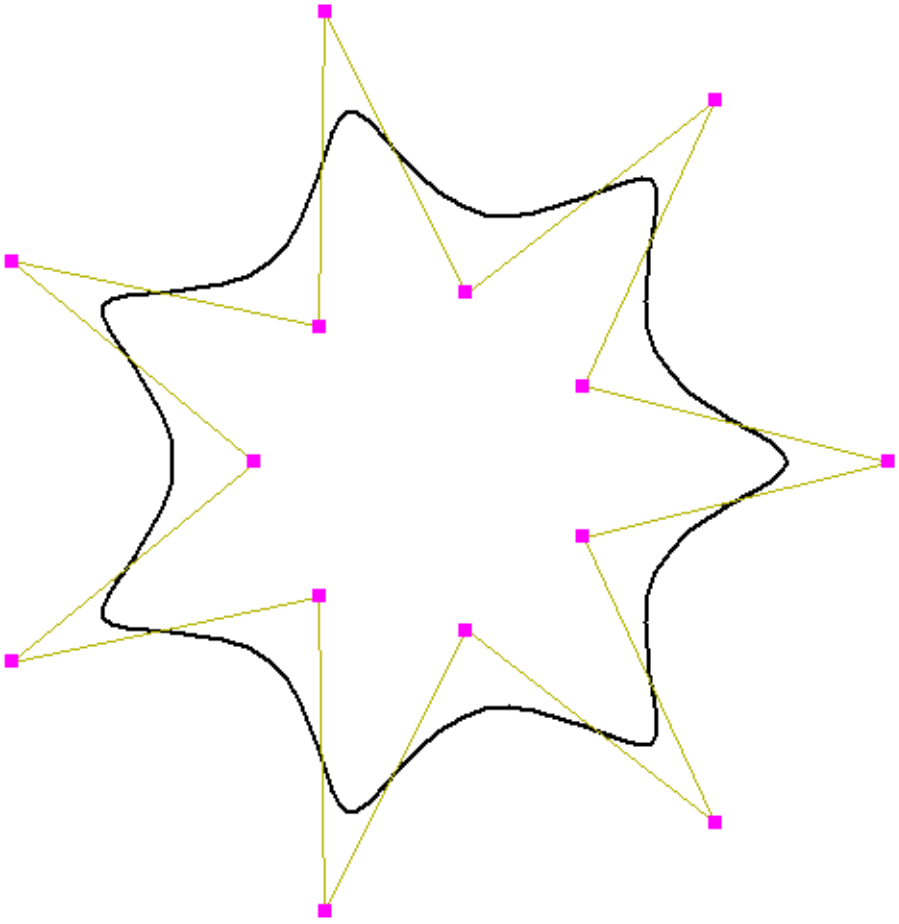


Figure 78. de Boor spline

ccInterpSpline

The **ccInterpSpline** class implements an interpolation spline in CVL. This class implements the **ccShape** and **ccCubicSpline** base class methods for interpolation splines.

Interpolation splines interpolate all control points. They do not stay within the convex hull of the control points. Control points do not have weights. Interpolation splines do not provide local control of the curve; that is, moving a control point affects the entire spline. As interpolation splines are B-splines, the junctions between the component Bezier curves have continuous second derivatives.

Interpolation splines are the slowest to compute.

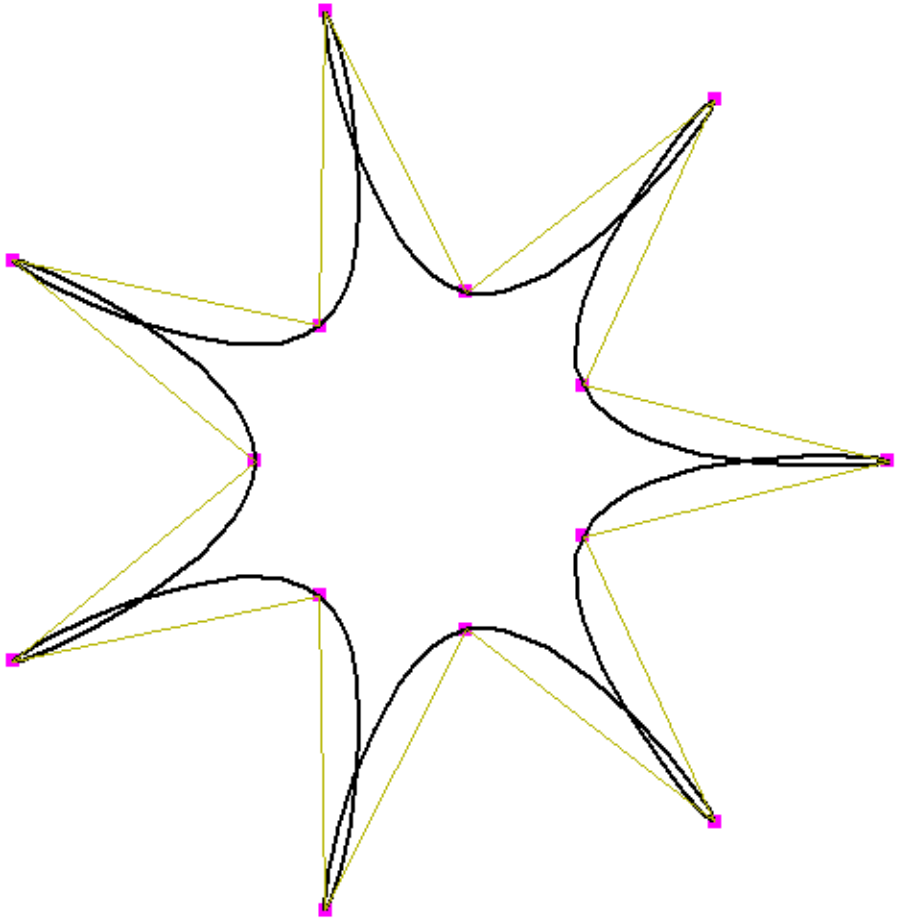


Figure 79. Interpolation spline

ccHermiteSpline

The **ccHermiteSpline** class implements a Hermite spline in CVL. This class implements the **ccShape** and **ccCubicSpline** base class methods for Hermite splines.

Hermite splines interpolate all control points. They support manipulation of tangent vectors at the control points. They do not stay within the convex hull of the control points. The control points do not have weights. Hermite splines provide local control of the curve. As Hermite splines are not B-splines, they do not necessarily have continuous second derivatives at the junctions of component Bezier curves.

Hermite splines are the fastest to compute.

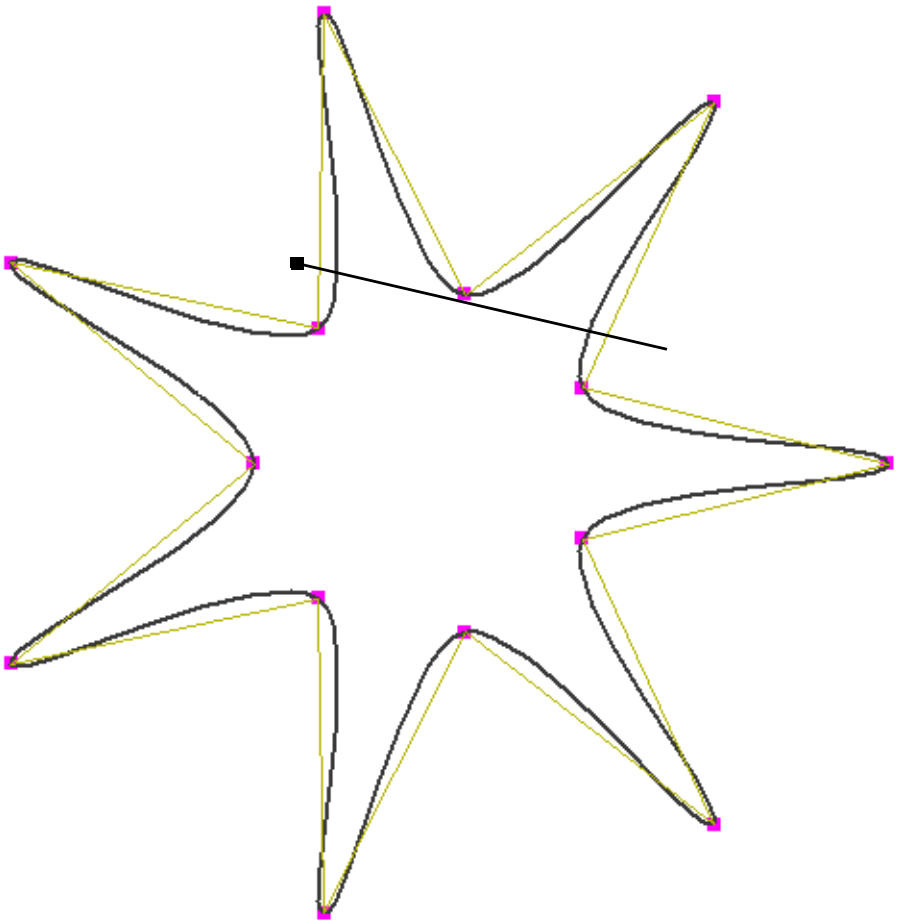


Figure 80. Hermite spline with manipulable tangent vector at a control point

Polygons and Wireframes

The **ccPolyline**, **ccGenPoly**, and **cc2Wireframe** classes implement polygons, generalized polygons, and wireframes, respectively, in CVL.

ccPolyline

The **ccPolyline** class represents a polygon shape. This class implements the **ccShape** base class methods for polygons.

Note **ccPolyline** replaces **ccPolygon**, which is now deprecated.

You construct a **ccPolyline** by specifying an initial set of vertices, and optionally whether the **ccPolyline** is open or closed (the default is open). The open/closed state is controlled through a flag, and is independent of the vertex positions. For example, a **ccPolyline** can be open even though its first and last vertex coincide, or can be closed when they do not. In the latter case, there is an implied line segment connecting the last and first vertex.

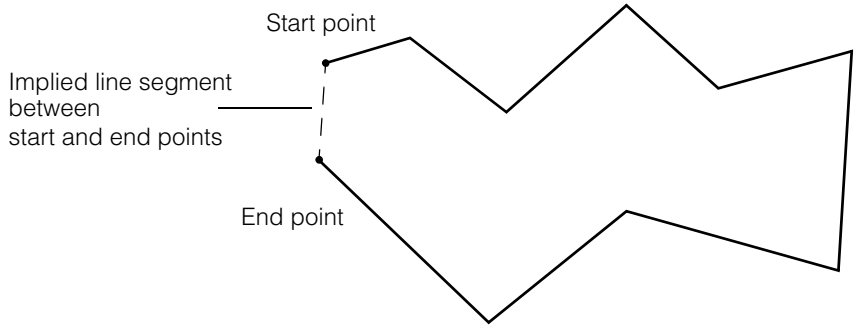


*Figure 81. Closed **ccPolyline** whose start and end points do not coincide*

Computing Standard Geometric Data

The **ccPolyline** class includes several methods to compute standard geometric data for polygons. Among the most commonly used are the following:

- The **perimeter()** method computes the length of the perimeter of the polygon. This method applies to both open and closed polygons.
- The **area()** method returns the area enclosed by the polygon. This method applies only to closed polygons
- The **centerArea()** method returns the point that is the center of area (centroid) of a closed polygon.

- The **principalMomentsArea()** method computes the principal moments of area of a closed, non-empty polygon.
- The **principalMomentsArcLength()** method computes the principal moments of arc length of a polygon.

The latter two methods also compute the transform from the coordinate system in which the principal moments are computed to the coordinate system in which the polygon's vertices are defined. Both of these methods relate to a principal coordinate system: an orthogonal coordinate system with origin at the centroid and an orientation that causes the products of area to vanish.

The results of all of the above methods are exact for polygons.

ccGenPoly

The **ccGenPoly** class implements the **ccShape** base class methods for generalized polygons.

The **ccGenPoly** class encapsulates a general polygon shape consisting of sequentially connected 2D arc and line segments connected at vertices. Each vertex is specified by a point and a rounding size. The vertex point specifies the coordinate of the vertex. The rounding size specifies the circular radius of rounding of the vertex. The general polygon object is created incrementally by adding vertices one at a time, and adjusting the curvature of the resulting segments for rounding.

When running PatMax, you will generally use a **ccGenPoly** (or **ccGenPolyModel**) shape. You would only use a **cc2Wireframe** in cases where you need segment length tolerance information.

Vertex Rounding

The rounding size for any vertex of a **ccGenPoly** may be 0, a positive number, or a negative number:

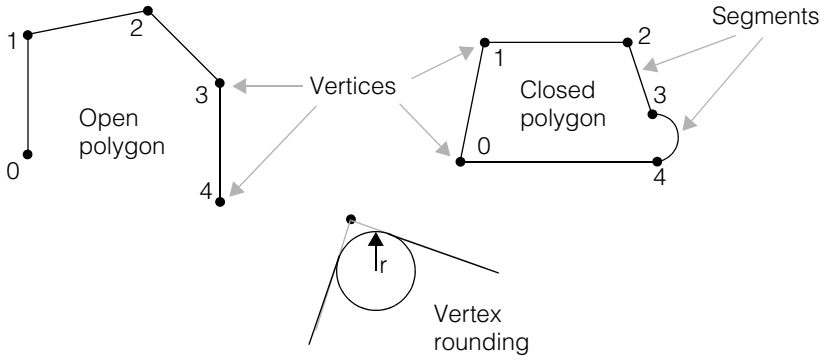
- A corner rounding value of 0 (the default) causes the corner to be drawn as a sharp intersection of the two segments.
- A positive corner rounding value causes the corner to be drawn as the radius of a circular arc, or *fillet*, which smoothly blends one segment into the other.
- A negative corner rounding value tells pattern matching tools to ignore what would be the circular arc, or *fillet*, if the value were positive.

The *showVertex* argument to the **ccUITablet::draw()** overload for **ccGenPoly** controls only how vertices with negative corner rounding values are drawn. It is ignored when the rounding value is 0 or positive.

Though the shape characteristics are exactly the same for positive and negative sizes (the absolute value is used), the meaning of the sign may differ when a general polygon is used by a particular vision tool. Note that a rounding size is considered invalid if more than half the length of the segment to either side of the vertex is altered by the rounding.

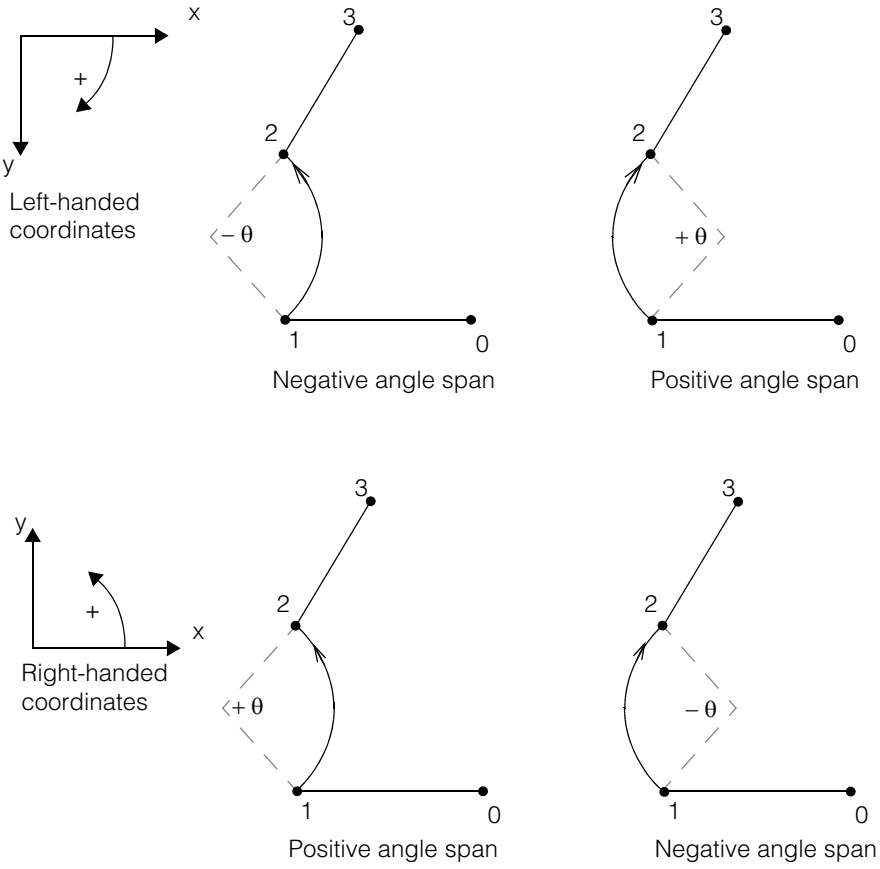
Vertex and Segment Indices

Vertices in the general polygon have indices that increase by one as you traverse each segment of the general polygon in the direction that it was originally created. Vertex 0 refers to the first vertex, vertex 1 refers to the second vertex, and so on. For existing polygons, vertices can be inserted between any two vertices, before the first vertex, or after the last vertex. Vertices may also be deleted. Insertion or deletion will cause vertex indices to change. Therefore, it is typical but not true in general, that vertex 0 is the first inserted vertex. See the following examples:



Segments also have indices. Segment N is defined as the segment between vertex N and vertex ((N+1) modulo M), where M is the number of vertices in the general polygon. The curvature of each segment is indicated by its *segmentAngleSpan*, which is the angle subtended by that segment between the two vertex points. A span of 0 specifies a straight line segment. A positive span for segment N specifies an arc that arcs in the direction of increasing angle when moving along the arc from vertex N to vertex N+1,

while a negative span indicates an arc that arcs in the direction of decreasing angle when moving along the arc from vertex N to vertex N+1. The following are examples of segment angle spans.



A general polygon object may be open or closed. However, none of the segments or vertices in the general polygon may intersect. Operations that might cause such crossings to occur result in an appropriate throw, and leave the general polygon unaffected.

Immutable General Polygons

There are certain operations which result in an immutable general polygon object (one which cannot be subsequently changed); specifically, any operation which causes the general polygon to contain elliptical arc segments or vertex roundings. These operations include mapping the shape by a linear transform, or constructing a shape from another shape containing elliptical components. Because of this you should set up

general polygons completely in a single space before transforming them into other spaces; for example, when drawing or for training vision tools. Further, you currently cannot directly construct general polygons comprising multiple elliptical segments.

cc2Wireframe

The **cc2Wireframe** class implements the **ccShape** and **ccGenPoly** base class methods for wireframes. A special type of generalized polygon (**ccGenPoly**), a wireframe shape consists of a set of vertices sequentially connected by circular arcs and line segments.

The following components comprise a wireframe:

- The vertices that make up the wireframe
- The segments that connect the vertices (straight lines or circular arcs)
- The polarity of the wireframe

Figure 82 shows an example of wireframe.

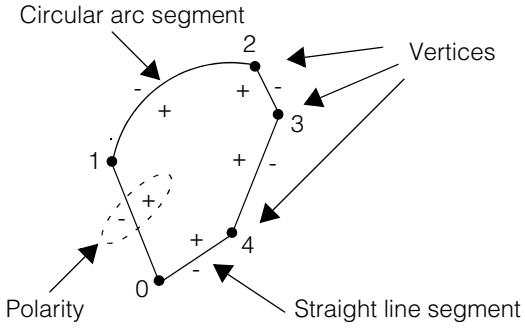


Figure 82. A wireframe

Vertices

The vertices that make up a wireframe object are defined by two parameters:

- The coordinates (v_x, v_y) of the point where the vertex is located
- The rounding parameter r that determines the roundness of the vertex

Figure 83 illustrates how the rounding parameter affects the roundness of a vertex:

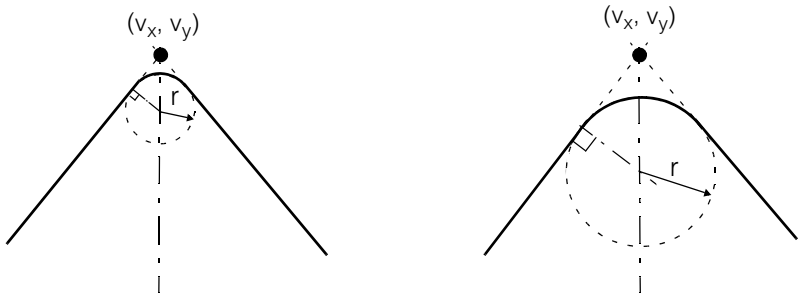


Figure 83. Effect of the parameter r on the roundness of the vertex.

As shown in Figure 83, the larger the rounding parameter, the larger the curvature of the vertex.

Vertices are identified by an index that increases by one following the order in which they were created. The index of the first vertex is always 0.

Segments

A segment joining two vertices is defined by two parameters:

- The angle span θ that determines the curvature of the segment
- The length tolerances of the segment

Angle Span

Figure 84 illustrates how the curvature of a segment is affected by the angle span parameter θ.

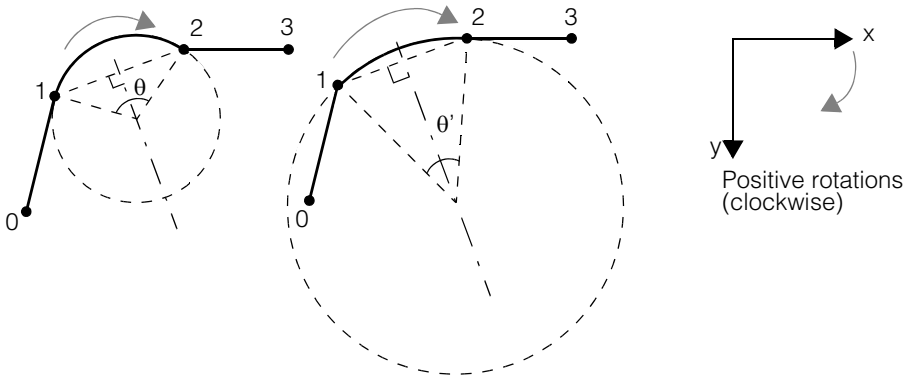


Figure 84. Effect of angle span on the curvature of the segment joining two vertices.

As shown in Figure 84, the smaller the angle span, the smaller the curvature of the circular arc that joins the two vertices. When the angle span is 0°, the arc degenerates to a straight segment.

Segments in a wireframe are also identified by indices: the segment joining the vertex N to N+1 is indexed by N. For example, the index of the segment between the vertex 0 and 1 is 0.

The angle span parameters in Figure 84 are both positive. The angle span of a segment N is defined positive if the arc that joins the vertex N to N+1 is in the direction of positive rotations. The sign of the span angle determines the concavity of the segment. In Figure 84 the coordinate frame is left-handed and the direction of positive rotations is clockwise. If you specify a negative angle span in that coordinate frame, the arc segments reverse their concavity as shown in Figure 85.

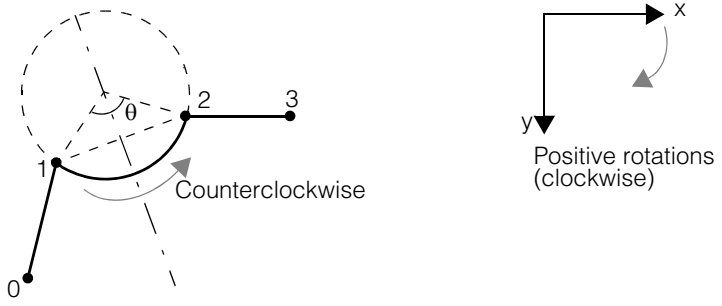


Figure 85. The effect of negative angle span on the concavity of the segment

Length Tolerances

Each segment of a wireframe object has associated tolerance information that includes the segment's nominal dimension, the minimum dimension, and the maximum dimension.

You can specify the tolerances on a segment dimension in three different ways:

- As absolute limits on the nominal dimension of the segment. In this case the length of the segment is limited between the interval MinDim and MaxDim, where MaxDim = maximum segment dimension and MinDim = minimum segment dimension. The nominal dimension of the segment must be between MinDim and MaxDim as shown in Figure 86.

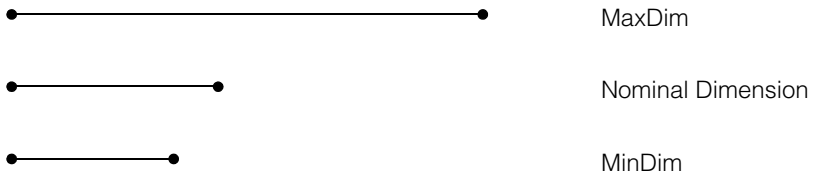


Figure 86. Absolute limits on the nominal dimension of a segment

- As relative tolerances on the nominal dimension of the segment. In this case the length of the segment is limited between the interval(NomDim-MinOff) and (NomDim+MaxOff) where: MaxOff= maximum offset, MinOff=minimum offset, NomDim = nominal dimension. The limits on the dimension of the segment are illustrated in Figure 87.

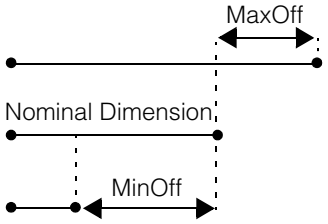


Figure 87. Relative tolerances on the nominal dimension of a segment

- As percentage tolerances on the nominal dimension of the segment. In this case the length of the segment is limited between [NomDim*(1-MinPerc)] and [NomDim*(1+MaxPerc)] where: MinPerc = minimum percentage tolerance, MaxPerc = maximum percentage tolerance and NomDim = nominal dimension. The limits on the dimension of the segment are illustrated in Figure 88.

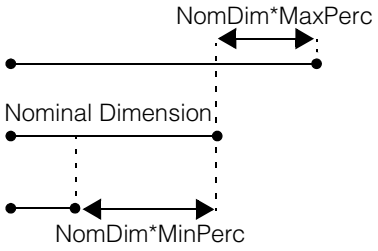


Figure 88. Percent tolerances on the nominal dimension of a segment

Notes

PatMax ignores tolerances. If you train PatMax using a wireframe shape, PatMax trains uses only the nominal dimensions of the segments that make up the wireframe, but no tolerance information.

Polarity

Each segment of a wireframe object has a positive and negative side associated with it. This allows you to encode polarity information in your synthetic model. When you specify a wireframe object for synthetic training, the default polarity is determined according to the following rule: the positive side of the segment N is the side the segment would move toward if it were to rotate in the direction of positive angles with center of rotation at vertex N. Figure 89 shows the polarity of a wireframe object when the coordinate frame is left-handed.

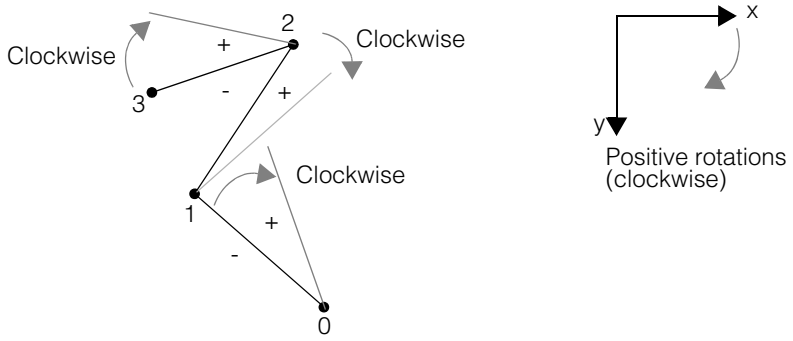


Figure 89. Polarity of a wireframe in a left-handed coordinate frame

Constraints on the Geometry of Wireframes

The geometry of wireframe shapes is subject to the following constraints:

a. The number of vertices N_v must be greater than or equal to the number of segments N_S.

b. There can be only one vertex at a specific position of the coordinate frame.

c. Any closed polygon must end at vertex 0.

d. Wireframe segments must not cross.

The examples shown in Figure 90 are valid instantiations of wireframe shapes.

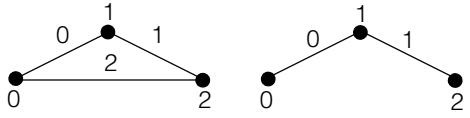


Figure 90. Valid examples of wireframes

The example in Figure 91 cannot be a wireframe since it has more segments than vertices (it violates requirement a). As shown in Figure 91, the segment that closes the wireframe is undefined.

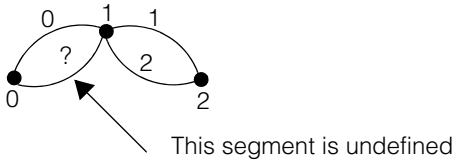


Figure 91. invalid representation of the pattern as a wireframe

If you want to create a wireframe representation of the pattern in Figure 91, you need to add an extra vertex at a position very close to the position of vertex 1 as shown in Figure 92 (notice that the new vertex cannot be at the same position as vertex 1, this would violate requirement b). The extra vertex allows you to close the wireframe properly.

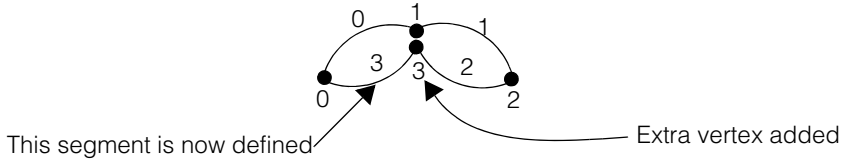


Figure 92. Valid wireframe representation of the pattern in Figure 91

The example in Figure 93 is not a wireframe because the polygon does not close at vertex 0 (it violates requirement c).

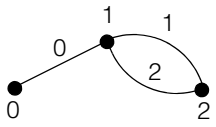


Figure 93. Invalid representation of the pattern as a wireframe

To create a valid wireframe representation of the pattern in Figure 93, you must add a fourth vertex at a position very close to the position of vertex 1 as illustrated in Figure 94.

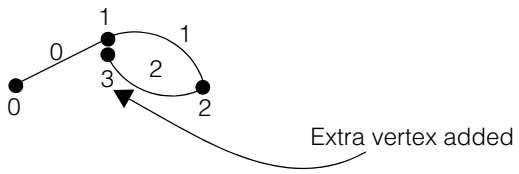


Figure 94. Valid wireframe representation of the pattern in Figure 93

A common closed shape you may need to model with a wireframe is the circle. To do this you only need two vertices and two segments as illustrated in Figure 95:

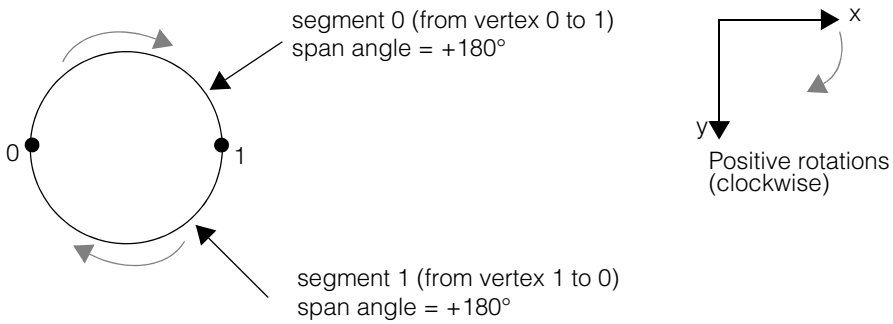


Figure 95. Wireframe representation of a circle

Creating Wireframe Shapes

You can create a wireframe shape by calling the member functions of the **cc2Wireframe** class. The following code shows how to create the wireframe representation of an L-shaped polygon and a circle.

1. Create a **cc2Wireframe** object and initialize it.

```
// constructs a wireframe shape with no vertices
cc2Wireframe w;
```

2. Add vertices to the wireframe. The following code shows two examples: an L-shaped polygon and a circle.

```
// Example 1: wireframe representation of an L-shaped polygon
w.insertVertex(cc2Vect(0, 0));
w.insertVertex(cc2Vect(100, 0));
w.insertVertex(cc2Vect(100, 100));
w.insertVertex(cc2Vect(200, 100));
w.insertVertex(cc2Vect(200, 200));
w.insertVertex(cc2Vect(0, 200));
w.close();

// Example 2: wireframe representation of a circle centered at
//(150, 100)and radius = 50
w.insertVertex(cc2Vect(100, 100), 0.0);
w.insertVertex(cc2Vect(200, 100), 0.0, ccDegree(180.0));
w.close(ccDegree(180.0));
```

3. If you want to display the wireframe in a console for graphical editing, create a **ccUIGenPoly** version of the wireframe and add it to the console:

```
ccUIGenPoly *uiPoly = new ccUIGenPoly;
uiPoly->shape(w);
uiPoly->color(ccColor::red);
uiPoly->condVisible(true);
console.addShape(uiPoly, ccDisplay::eClientCoords);
```

After editing, use **cc2Wireframe::map()** to map the coordinates of the vertices from wireframe to client coordinates. For example, if the client coordinate system is just a scaled version of the image coordinate system you can use the following code to create a wireframe expressed in client coordinates.

```
cc2Wireframe w_cc = w.map(cc2Xform(cc2Vect(0,0),ccDegree(0),
  ccDegree(0), 0.1, 0.1));
```

Converting a Shape into a cc2Wireframe Object

You can easily convert a CVL shape into a wireframe object. The following code shows how to create the wireframe representation of a circle centered at (200, 200) with radius of 100 (remember that **cc2Wireframe** is derived from **cc2GenPoly**).

```
cc2Wireframe w(ccCircle(cc2Vect(200, 200), 100.0));
```

Creating a Wireframe Graphically

You can create wireframe objects graphically by using **ccUIGenPoly** objects. You can use the various drawing and editing modalities provided by this class to create and edit wireframe shapes.

Drawing Wireframes

The following code shows the basic steps to draw a polygon from within a **ccDisplayConsole** and make it into a wireframe object.

1. Create a **ccUIGenPoly** object and adjust the settings.

   ```
   ccUIGenPoly *uiPoly = new ccUIGenPoly;
   uiPoly->color(ccColor::red);
   uiPoly->condVisible(true);
   ```

2. Add the **ccUIGenPoly** object to a display console.

   ```
   console.addShape(uiPoly, ccDisplay::eClientCoords);
   ```

3. Set the drawing mode.

   ```
   uiPoly->drawMode(ccUIGenPoly::eDrawPolygonMode);
   ```

4. Draw the polygon in the display console.

5. Convert the polygon you draw into a wireframe.

   ```
   cc2Wireframe wrfrm = uiPoly->shape();
   ```

6. The wireframe object `wrfrm` is represented in wireframe coordinates (when you display the wireframe shape, the wireframe coordinate axes are shown in yellow). If you want the wireframe to be represented in client coordinates you need to map the wireframe from its coordinate system to the client coordinate system. To do this you use **ccUIGenPoly::modelFromShape()** to get the transformation between wireframe coordinates and client coordinates and then map the wireframe:

   ```
   cc2Wireframe wrfrm_cc = wrfrm.map(uiPoly->modelFromShape());
   ```

   You can also obtain the wireframe in client coordinates directly from `uiPoly` by calling **ccUIGenPoly::wireframe()**:

   cc2Wireframe wrfrm_cc = uiPoly->wireframe();

The following table shows all the drawing modalities of **ccUIGenPoly** and how to modify step 3 on page 390 to implement them:

Drawing Mode	Step 3 on page 390
Circle	`uiPoly->drawMode(ccUIGenPoly::eDrawCircleMode);`
Rectangle	`uiPoly->drawMode(ccUIGenPoly::eDrawRectMode);`
Triangle	`uiPoly->drawMode(ccUIGenPoly::eDrawTriangleMode);`
Corner	`uiPoly->drawMode(ccUIGenPoly::eDrawCornerMode);`
Polygon	`uiPoly->drawMode(ccUIGenPoly::eDrawPolygonMode);`

You can create wireframe objects by drawing them directly on top of an acquired image. The following code shows how to do this.

1. Perform all the steps necessary for acquisition. The following code starts with live-video mode, acquires the image of the object you have positioned in front of the camera and displays it on a display console.

```
// get a reference to a frame grabber
cc8100 &frameGrabber = cc8100::get(0);
// create a video format
const ccStdVideoFormat &videoFormat =
       ccStdVideoFormat::getFormat(cmT("Sony XC75 640x480"));
// create a pointer handle to a fifo
ccStdGreyAcqFifoPtrh fifo =
  videoFormat.newAcqFifo(frameGrabber);
// create a display console
ccDisplayConsole console(ccIPair(640, 480),
  cmT("Poly Editor"));
console.closeAction(ccDisplayConsole::eCloseDelete);
// start displaying live video
console.startLiveDisplay(fifo.rep());
// use message box to pause the program
MessageBox(NULL, "Live Video\n\n1)  Position the object under
  the camera\n2)  Hit OK to acquire image", "Hit OK to
  Continue", MB_OK);
// stop displaying live image
console.stopLiveDisplay();
// start acquisition
fifo->start();
// complete acquisition
ccPelBuffer<c_UInt8> acquiredImage = fifo->complete();
```

2. Define the client coordinate system (in the following code the client coordinate system is a simple scaled version of the image coordinate system).

```
acquiredImage.clientFromImageXform(cc2Xform(cc2Vect(0, 0),
  ccDegree(0), ccDegree(0), 0.5, 0.5));
```

3. Draw on top of the acquired image and create the wireframe following the steps 1 through 5 described in *Drawing Wireframes* on page 390.

```
ccUIGenPoly *uiPoly = new ccUIGenPoly;
uiPoly->color(ccColor::red);
uiPoly->condVisible(true);
console.addShape(uiPoly, ccDisplay::eClientCoords);
uiPoly->drawMode(ccUIGenPoly::eDrawPolygonMode);
MessageBox(NULL, cmT("Click OK when finished drawing
  polygon."),cmT("Poly Editor"), MB_OK);
// Get the wireframe from uiGenPoly
cc2Wireframe wrfrm = uiPoly->shape();
```

4. If you want the wireframe to be represented in the client coordinates defined in step 2, use **ccUIGenPoly::modelFromShape()** to get the transformation between wireframe coordinates and client coordinates and then map the wireframe.

```
cc2Wireframe wrfrm_cc = wrfrm.map(uiPoly->modelFromShape());
```

   You can also obtain the wireframe in client coordinates directly from `uiPoly` by calling **ccUIGenPoly::wireframe()**:

```
cc2Wireframe wrfrm_cc = uiPoly->wireframe();
```

Editing Wireframes

The **ccUIGenPoly** class includes also several editing modalities. The following code shows you how to display an existing wireframe object and edit its vertices.

1. Create a **ccUIGenPoly** object.

```
ccUIGenPoly *uiPoly = new ccUIGenPoly;
```

2. Add an already existing wireframe object (*wrfrm*) to the display console (or draw one following the steps described in *Drawing Wireframes* on page 390):

```
uiPoly->shape(wrfrm);
uiPoly->color(ccColor::blue);
uiPoly->condVisible(true);
console.addShape(uiPoly, ccDisplay::eClientCoords);
```

3. Set the editing mode.

```
uiPoly->editMode(ccUIGenPoly::eVertexMode);
```

4. Edit the vertices of the wireframe in the display console.

The following table shows all the editing modalities of **ccUIGenPoly** and how to change step 3 on page 392 to implement them:

Editing Mode	Step 3 on page 392
New vertex	uiPoly->editMode(ccUIGenPoly::eNewVerticesMode);
Delete vertex	uiPoly->editMode(ccUIGenPoly::eVertDelMode);
Edit vertex	uiPoly->editMode(ccUIGenPoly::eVertexMode);
Delete segment	uiPoly->editMode(ccUIGenPoly::eSegDelMode);
Edit rounding	uiPoly->editMode(ccUIGenPoly::eRoundingMode);

CAD File Import

9

This chapter describes how to import a CAD drawing in standard DXF format into a CVL general shape tree. You can then use the shape tree for several purposes within CVL.

This chapter has two sections:

Some Useful Definitions, which defines terms useful in reading this chapter.

Importing DXF Files to Shape Models describes the importing of DXF format files into CVL shape trees.

Some Useful Definitions

The following terms may be useful in reading this chapter.

AutoCAD® CAD software from AutoDesk, Inc., considered by many to be the benchmark for CAD software.

CAD Computer aided drawing, a category of software for drawing and maintaining technical drawings for industries such as architecture, engineering, and manufacturing.

DXF Document exchange format, a proprietary but widely used file format for exchanging technical drawing information between CAD programs.

Shape tree A hierarchy of shape objects grouped together to describe a complex geometric model. See *Shape Hierarchies* on page 343.

Volo View Express Freely available DXF file viewing software from Autodesk, Inc.

Importing DXF Files to Shape Models

The DXF™ file format is a standard format for the exchange of technical drawings between different drafting and CAD programs. The DXF format is developed and maintained by Autodesk, Inc. as a means of exchanging technical drawings between their AutoCAD® product line and other programs.

CVL provides the **ccCADFile** API that lets you import the basic two-dimensional geometric shape information from a DXF format file, and import its drawing shape primitives into **ccShape** objects within a **ccGeneralShapeTree** object.

Once a DXF file is imported into a shape tree, you can use it for several purposes in CVL, including:

- As a template that can be used to rasterize run-time masks for various vision tools, such as the Blob tool, the PatMax tool, or edge detection. (See **cfRasterize()** and **cfRasterizeContour()** in *Rasterizing Shapes* on page 356.)
- As a model for Edgelet Chain Filtering tool (see **cfFilterEdgeletChains()** in the *CVL Class Reference*).
- As the trained model for PatMax pattern location.
- To display diagnostic graphics.

DXF Versions Supported

The DXF format is an evolving standard, closely tracking released versions of AutoCAD software. CVL supports text-based DXF files in formats corresponding to the versions of AutoCAD shown in Table 56. CVL does not support the binary version of the DXF file format.

DXF version ID	AutoCAD Version	Comments
AC1009	R11	Same DXF version as R12
AC1009	R12	Supported by CVL
AC1012	R13	Supported by CVL
AC1014	R14	Supported by CVL
AC1015	2000	Supported by CVL
AC1015	2000i	Same DXF version as 2000
AC1015	2002	Same DXF version as 2000

Table 56. DXF versions supported by CVL for import

DXF File Format

The DXF file format is divided into sections. Table 57 shows the DXF section names and how they are parsed by **ccCADFile.**

DXF Section	Use when importing into ccCADFile object
HEADER	Parsed to determine the units, handedness, and base angle of the drawing
TABLES	Parsed for information about the drawing's layers
BLOCKS	Parsed for information about any custom objects and groups
ENTITIES	Parsed for information about basic drawing shapes and primitives
CLASSES	Not used by **ccCADFile**
OBJECTS	Parsed for group information

Table 57. DXF sections used by CVL

The DXF ENTITIES section contains one or more entities, which correspond to CAD drawing primitive shapes. Table 58 shows the DXF entities recognized by **ccCADFile**, and shows the mapping between each entity and its corresponding CVL shape object.

DXF Entities	Use when importing into ccCADFile object
ARC	Converted into a **ccEllipseArc2** object
CIRCLE	Converted into a **ccEllipse2** object
ELLIPSE	Converted into a **ccEllipseArc2** object, including start and stop angles
INSERT	Inserted custom object converted into a **ccGeneralShapeTree**
LINE	Converted into a **ccLineSeg** object
LWPOLYLINE	Converted into a **ccPolyline** or **ccCountourTree** object (see notes below)
MLINE	Converted into a **ccPolyline** object

Table 58. DXF entities and corresponding CVL shape objects

DXF Entities	Use when importing into ccCADFile object
POINT	Converted into a **cc2Point** object. Any Z position information is ignored.
POLYLINE	Converted into a **ccPolyline** or **ccCountourTree** object (see notes below)
RAY	Converted into a **ccLine** object
SOLID	Boundary converted into a closed **ccPolyline** object
SPLINE	Converted into a **ccDeBoorSpline** object
XLINE	Converted into a **ccLine** object

Table 58. DXF entities and corresponding CVL shape objects

ccCADFile ignores the following DXF entities:

3DFACE	HATCH	SEQEND
3DSOLID	IMAGE	SHAPE
ACAD_PROXY_ENTITY	LEADER	TEXT
ATTDEF	MTEXT	TOLERANCE
ATTRIB	OLEFRAME	TRACE
BODY	OLE2FRAME	VERTEX
DIMENSION	REGION	VIEWPORT

The following DXF entity information is ignored:

- All three dimensional information
- Fill and style information
- Rounded corner and spline representations using POLYLINE

POLYLINE and LWPOLYLINE entities are represented as either **ccPolyline** or **ccCountourTree** objects. If the DXF entity consists of straight line segments, a **ccPolyline** object is returned. If the DXF entity contains any arcs, a **ccCountourTree** object consisting of **ccEllipseArc2** and **ccLineSeg** objects is returned.

The INSERT entity provides a mechanism for displaying a transformed version of a custom object from the BLOCKS section of the DXF file. Its **ccShape** is a **ccGeneralShapeTree** whose children consist of the transformed **ccShape** versions of its defining entities (which can include other INSERTs or trees). If a BLOCK is used more than once, it is replicated into a new **ccGeneralShapeTree**.

Saving DXF Files

The DXF format is supported as a standard file format by all major CAD programs, and by technical drawing programs such as Microsoft Visio®.

Most CAD programs allow you to rotate the drawn object in three-dimensional space, and to save DXF format snapshots of the object in various positions. For importation into CVL as a **ccGeneralShapeTree** object, save the DXF file when the object is in a flat, two-dimensional view.

Importing DXF Files

The CVL sample code file *pmalign4.cpp* demonstrates the steps to prepare and import a DXF drawing into a general shape tree. The overall steps are described in this section.

1. Specify the name and path to your DXF file, construct an empty **ccCADFile** object, and open the DXF file into the object.

   ```
   ccCvlString cad_filename = cmT("plate.dxf");
   ccCADFile cadFile;
   cadFile.open(cad_filename);
   ```

2. At this point, the recognized DXF primitives are imported into the specified **ccCADFile** object, but are not yet useful to CVL vision tools. Make them useful by declaring a **ccGeneralShapeTree** object and adding the imported DXF primitives as nodes on the shape tree:

   ```
   ccGeneralShapeTree importedShapeTree;
   cadFile.shapeTree(importedShapeTree);
   ```

   Any unrecognized DXF features are ignored.

3. Now that the DXF objects are in the shape tree, you can close the DXF file.

   ```
   cadFile.close();
   ```

If you plan to use this shape tree as a PatMax model, you will have better results if you flatten and connect the imported shapes. This is because DXF files contain no explicit information about the connectivity of shapes. For example, a rectangle might be represented as four individual line segments whose endpoints are coincident. The flattening and connecting operations interpret coincident endpoints in order to make

connections explicit. For example, these operations would take the four individual line segments and create a contour tree containing four line segments. PatMax then uses this contour explicit information when training the pattern, which in general results in better alignment performance.

4. Run the **flatten()** method on your imported shape tree like this example. (For more on **flatten()**, see **ccShapeTree** in the *CVL Class Reference.*)

```
ccShapeTreePtrh flattenedShapeTree =
    importedShapeTree.flatten();
```

5. Run **connect()** on your flattened shape tree, as shown in this example. **connect()** takes one argument, a `double` value to specify the tolerance, which is the maximum distance between endpoints of different open contours that can be connected to each other in contour trees. This example shows a tolerance value of 10 microns. (For more on **connect()**, see **ccGeneralShapeTree** in the *CVL Class Reference.*)

```
ccGeneralShapeTreePtrh connectedShapeTree
  = (dynamic_cast<ccGeneralShapeTree
  &>(*flattenedShapeTree)).connect(1.0e-3);
```

Determining the DXF Layer to Import

CAD drawings are usually constructed in layers. For example, one drawing layer might contain the lines and arcs of the drawing itself, while another layer contains text annotations. You can specify which drawing layer to convert when importing a DXF file into a **ccCADFile** object.

To determine the names of the layers in the DXF file you want to import, you can

- Refer to the CAD program that originated your DXF file.
- Use the free Volo View Express program from AutoDesk, Inc. to display the DXF file and its layer names.
- Use **ccCADFile::layerNames()** to retrieve a vector containing the names of all layers in the DXF file.

Using **layerNames()** is useful if the layers have names that describe their contents. For example, in a drawing with layers named DIMENSIONS, NOTES, DRAWING, and LABELS, you can guess that the layer containing the supported DXF entities is the DRAWING layer.

Using Volo View Express

AutoDesk, Inc. publishes a DXF file viewer named Volo View Express, and makes it freely available for download from their web site. For information and support, see http://www.autodesk.com/voloviewexpress.

When you have opened your DXF file in Volo View Express, use **View -> Layers** to show the Layers popup window. This window lets you enable and disable the display of each layer independently. Figure 96 shows the Layers window with the names of the layers in the example DXF file *plate.dxf*, which is shipped with CVL in the *%VISION_ROOT%\samples\cvl* directory.

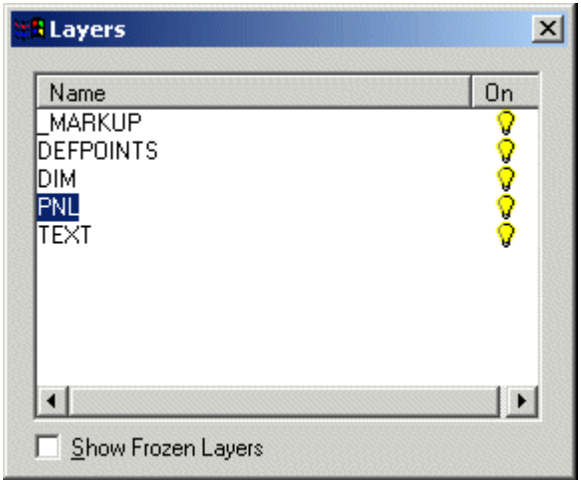


Figure 96. Layer window of Volo View Express showing layers in plate.dxf

Figure 97 shows portions of the Volo View Express main window as it displays various layers of *plate.dxf*.

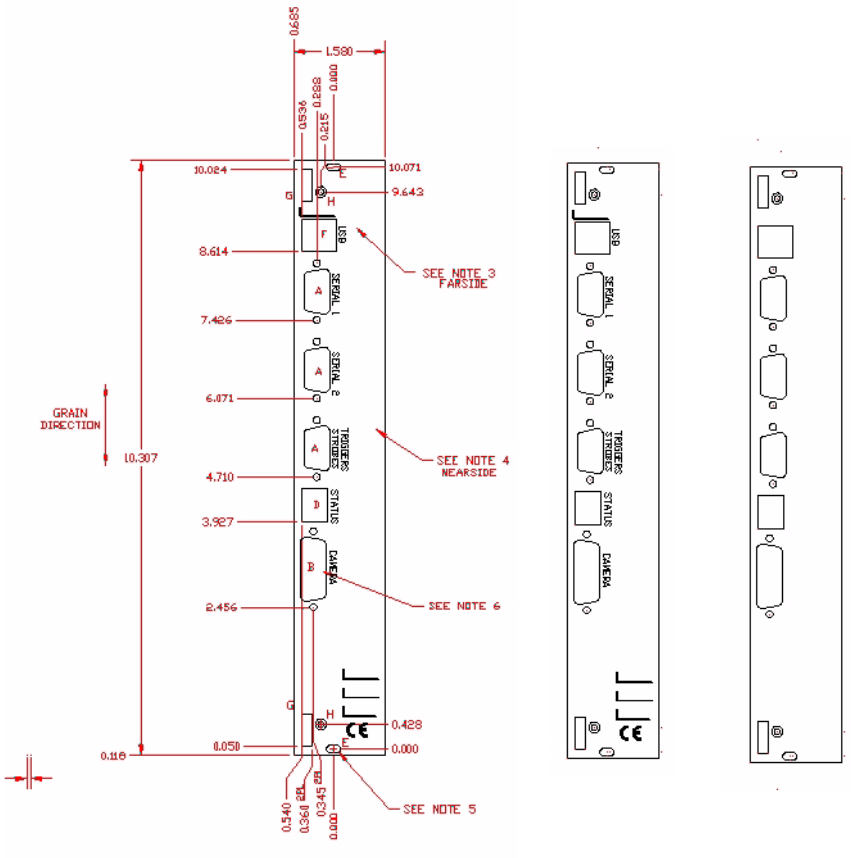


Figure 97. Volo View Express showing various layers of plate.dxf

As shown in Figure 97, in the example file *plate.dxf*, only the layer named PNL contains drawing primitive entities that are supported by **ccCADFile**. All information in the DIM and TEXT layers is ignored by **ccCADFile**.

Importing a DXF Layer

Once you have determined which layer or layers in your DXF file contain the drawing shapes you want to import. use the **layerShapeTree()** method to import that layer into a general shape tree. You can specify the layer by name or by its zero-based index in a vector of layer names returned by **layerNames()**.

For example, to import the PNL layer of the *plate.dxf* sample file, use code like this fragment:

```
ccCvlString cad_filename = cmT("plate.dxf");
ccCADFile cadFile;
cadFile.open(cad_filename);
ccGeneralShapeTree importedShapeTree;
cadFile.layerShapeTree("PNL", importedShapeTree);
```

or

```
cadFile.layerShapeTree(3, importedShapeTree);
```

The contents of the layer are added as a separate branch on the specified shape tree.

Importing AutoCAD Groups

AutoCAD software supports a feature called *groups* that allows you to create a named group of objects that have a feature in common, independent of their hierarchy in the drawing. For example, you could group together all of the components of an assembly drawing that are sourced from the same vendor. Grouped objects can be on any drawing layer, and can thus cross layer boundaries.

If your DXF file has a named group that contains DXF drawing entities that **ccCADFile** supports, you can import that group into your **ccCADFile** object. The contents of the specified group are added as a separate branch of the imported shape tree.

You can specify the group by name or by its zero-based index in a vector of group names returned by **groupNames()**. Use code like these fragments:

```
cadFile.groupShapeTree("drill_holes", importedShapeTree);
```

or

```
cadFile.groupShapeTree(0, importedShapeTree);
```

The contents of the group are added as a separate branch on the specified shape tree.

Math Foundations of Transformations

10

This chapter provides a brief summary of the basic mathematical definitions and conventions used in CVL coordinate system transformations.

Some Useful Definitions defines certain terms you will encounter as you read.

Points and Vectors reviews the basic notions of points and vectors.

2D Transformations provides a discussion on 2D transformations and how they are implemented in CVL.

This chapter assumes familiarity with basic matrix algebra.

Some Useful Definitions

2D linear transformation A 2D transformation defined by a (2x2) matrix and a (2x1) vector.

point A position in the coordinate system.

polynomial transformation A nonlinear transformation defined by a polynomial function.

vector A length and direction in the coordinate system.

Overview of CVL Transforms

CVL provides transform classes that encapsulate functions which allow you to transform points from one coordinate space to another. The classes include 1D, 2D, and 3D linear transforms and 2D classes that support nonlinear transforms. In a linear transform, all points are transformed in the same way regardless of the point location. Nonlinear transforms map points differently depending on the point location. Nonlinear transforms allow you to correct distortions that are not uniform throughout the coordinate space.

The most common transform you will find in CVL is the 2D client coordinate transform that is part of every pel buffer. From this transform object you can obtain the *clientFromImage* transform that maps points from image space (pixels) to client coordinate space, a space you typically define in physical units. You can also obtain the *imageFromClient* transform that maps points from client coordinate space to image space.

The following are transforms you will find when using CVL.

Class	Transformation
cc1Xform	A one-dimensional linear transform used by CVL to map the scale and offset of points along one axis.
cc2Xform	A two-dimensional linear transform that maps points in one 2D space into another 2D space. The transform provides six degrees of freedom; x- and y-scale, x- and y-translation, x- and y-rotation. An alternative use of the same transform allows you to specify the six degrees of freedom in terms of x- and y-scale, x- and y-rotation, aspect ratio of the x to y-axis, and shear angle.
cc2Rigid	A two-dimensional transform similar to **cc2Xform** except that it does not correct for scale or skew.
cc3Xform	A three-dimensional linear transform that maps points in one 3D space into another 3D space.
cc2XformBase	A base class used to derive both linear and nonlinear 2D transform classes. See the derivation hierarchy in Figure 98 below.

Table 59. CVL transform summary

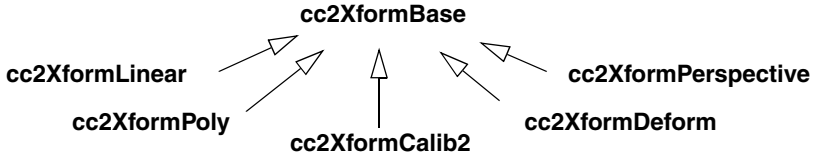


Figure 98. 2D transform class derivation hierarchy

A description of these 2D derived classes is given in Table 60 below.

Class	Transformation
cc2XformLinear	A linear 2D transform returned by the Grid-of-Dots calibration tool when a linear calibration is specified. This transform is typically used in pel buffers as the client coordinate transform. Note that **cc2XformLinear** is equivalent to **cc2Xform**. **cc2Xform** is the older version and **cc2XformLinear** the newer version that derives from **cc2XformBase**.
cc2XformPoly	A nonlinear 2D transform returned by the Grid-of-Dots calibration tool when a nonlinear calibration is specified. This transform is typically used in pel buffers as the client coordinate transform. Be aware that when you use a nonlinear client coordinate transform in your pel buffers, the vision tools you use on these images must support nonlinear client coordinate transforms. See *Using Vision Tools with Nonlinear Transforms* on page 323.
cc2XformCalib2	This is the class used for the Feature Correspondence Calibration tool. You call member functions of this class to create a nonlinear transform from corresponding points in image space and client space. The resulting transform is encapsulated in the class. In CVL, this transform is used by the Image Warp tool.
cc2XformPerspective	This perspective transform is computed by PatMax when it runs the PatFlex algorithm with the perspective transform specified (or when it runs the PatPersp algorithm). You can apply this transform to the PatMax run-time image to transform it into an image that matches the trained model. You can use the global function **cfSampledImageWarp()** to do this transformation.
cc2XformDeform	This deformation transform is computed by PatMax when it runs the PatFlex algorithm with the deformation transform specified. You can apply this transform to the PatMax run-time image to transform it into an image that matches the trained model. You can use the global function **cfSampledIMageWarp()** to do this transformation.

Table 60. Derived 2D transforms

Points and Vectors

Although both points and vectors are represented in CVL by the **cc2Vect** data type, they represent two conceptually distinct notions.

Point

A point represents a specific position in the coordinate frame. It is identified by a pair of numbers that specify the x- and y-coordinates of the point (see Figure 99).

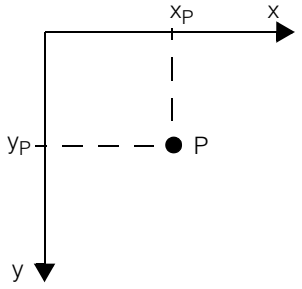


Figure 99. A point in the coordinate frame

The position of the point P in the coordinate frame is specified by (x_P, y_P) (see also *Pixels and Coordinate Grids* on page 226).

Vector

A vector specifies a length and a direction in the coordinate frame but has *no* fixed location. Vectors are depicted as arrows. In the following figure, the four arrows represent the same vector since they all share the same length L and direction θ (see Figure 100).

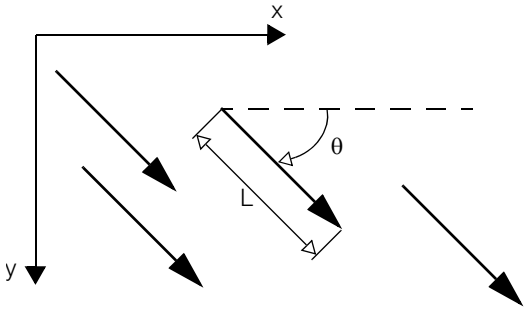


Figure 100. The four arrows represent the same vector in the coordinate frame

Vectors in CVL are defined in terms of their components (x_v, y_v) in the coordinate frame, where $x_v = L\cos\theta$ and $y_v = L\sin\theta$. This is an alternative way to specify the length and direction (see Figure 101).

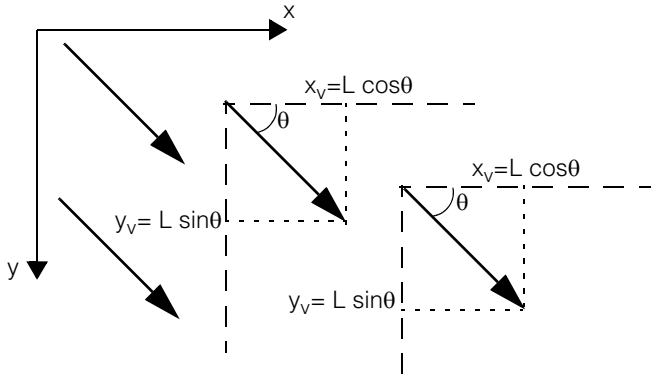


Figure 101. Vector components

Vectors and Points can share the **cc2Vect** data type because they are both represented by an x-y pair. A point can be thought of as the tip position of a vector whose tail is anchored at the origin. However, points and vectors are treated differently when they are

transformed between coordinate frames. To clarify this issue, consider Figure 102 that shows the coordinates of the point P and the components of the vector v in two different coordinate frames (Frame 1 and Frame 2).

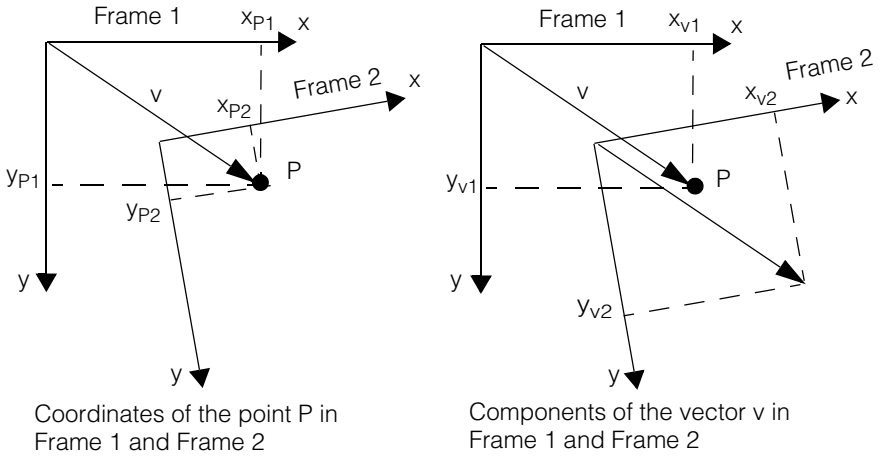


Figure 102. Point coordinates and vector components in two coordinate frames

Notice that the components of the vector and the coordinates of the point are the same in Frame 1 because the example intentionally represent the vector with an arrow whose tail is at the origin and whose tip is at the point P. In Frame 2 the vector components and the point coordinates are different because the point has a fixed position in space while the vector has a fixed length and direction.

The member functions **mapPoint()** and **mapVector()** in **cc2Xform**, **cc2XformLinear** and **cc2Rigid,** are used to map point coordinates and vector components from one frame to the other. In the preceding figure, **mapPoint()** would map (x_{P1}, y_{P1}) to (x_{P2}, y_{P2}) while **mapVector()** would map (x_{v1}, y_{v1}) to (x_{v2}, y_{v2}).

2D Transformations

A typical problem in machine vision is the one of mapping image features between coordinate frames. Suppose, for example, that the task of your machine vision application is to inform a robot manipulator of the position of a device as illustrated in Figure 103.

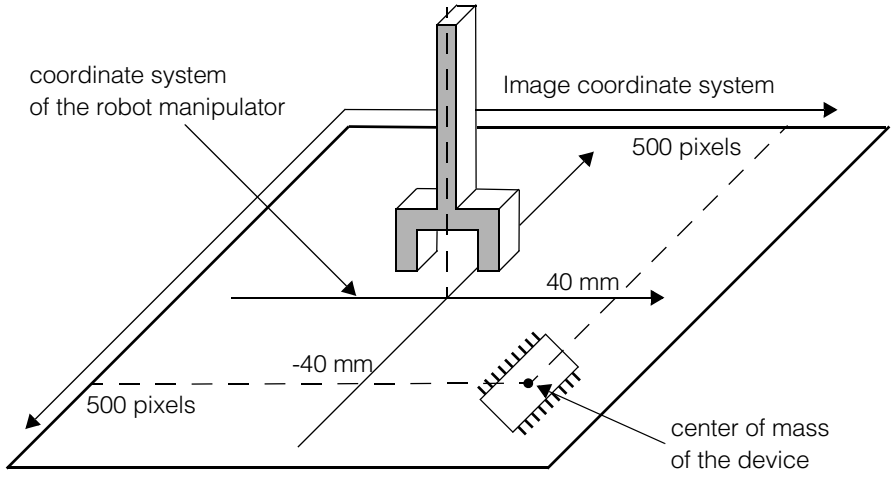


Figure 103. Mapping the center of mass of the device from image coordinate system to robot coordinate system

To accomplish this task the camera acquires an image of the platform and the machine vision tools locate the center of mass (CM) of the device. Before passing the information to the robot manipulator the application must solve one more problem: the CM coordinates are expressed in pixels in the coordinate frame of the machine vision application (the image coordinate system in the figure) while the robot manipulator has a coordinate system which is calibrated in millimeters. Furthermore, the vertical axis of the manipulator is not coincident with the vertical axis of the image coordinate system. In Figure 103 the image coordinates of CM are (500, 500), while its manipulator coordinates are (40, -40). In order to describe the position of the device in units meaningful to the manipulator the application must transform the CM coordinates from the image coordinate system to the coordinate system of the robot manipulator. Such mapping is performed by a 2D transformation.

Ways to Use 2D Transformations

In general, a 2D transformation is a mathematical mapping that enables you to map points from one place to another. 2D transformations can be used (and thought of) in a number of different ways. Each of these ways are mathematically equivalent; the difference is one of interpretation.

1. A transformation can be used to transform data; that is, to change the 2D coordinates of some geometrical entity within an existing coordinate frame. For example, imagine a planar shape held beneath a camera by a movable machine. Attached to this camera there is a coordinate frame and the planar shape held beneath it is described in that frame. In this case, a transformation describes a motion that the machine makes to change the appearance of the planar shape. It allows you to express the new shape in terms of the old.
2. A transformation can also be used to transform coordinate frames; that is, to change the axes in which you are measuring a fixed geometrical entity. For example, imagine that the planar shape in the example above is held fixed and that the camera (and the coordinate frame attached to it) moves instead of the shape. In this case the transformation can be thought of as describing the exact motion of the measurement axes. It allows you to express the shape in the new coordinate frame in terms of how it looked in the old coordinate system.

Figure 104 illustrates an example of these two ways to conceptualize a 2D transformation. In (a) the device is rotated by +50° with respect to the origin of the reference frame attached to the camera, which is kept fixed. In (b) the reference frame attached to the camera is rotated by -50° and the device is fixed. The two situations are equivalent. Consider, for example, the point P in the figure: $x_P' = x'_P$ and $y_P' = y'_P$.

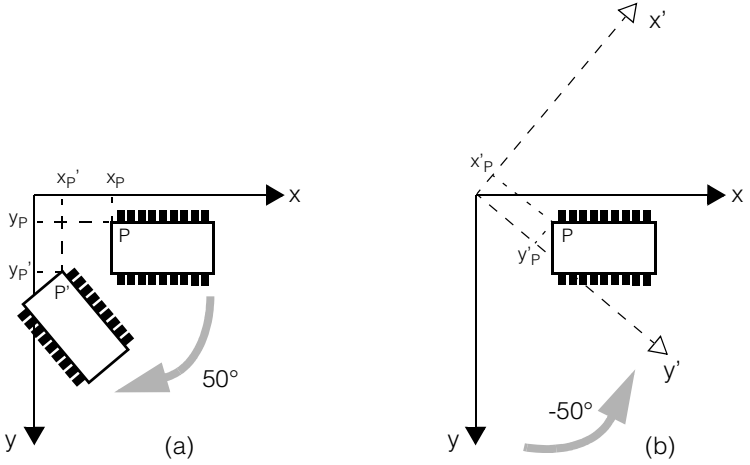


Figure 104. Transforming data as opposed to transforming coordinate frames

3. A transformation can be used to map data measured in two different coordinate frames. For example, imagine that the machine mentioned above has no moving parts at all, but has two cameras instead. In this case, the transformation can be thought of as describing the fixed relationship between the two cameras. It allows you to express the shape as seen from one camera in terms of its appearance in the other camera. In Figure 105 the two cameras are translated and rotated with respect to each other and you can use a 2D transformation to map image features in camera A to the corresponding features in camera B.

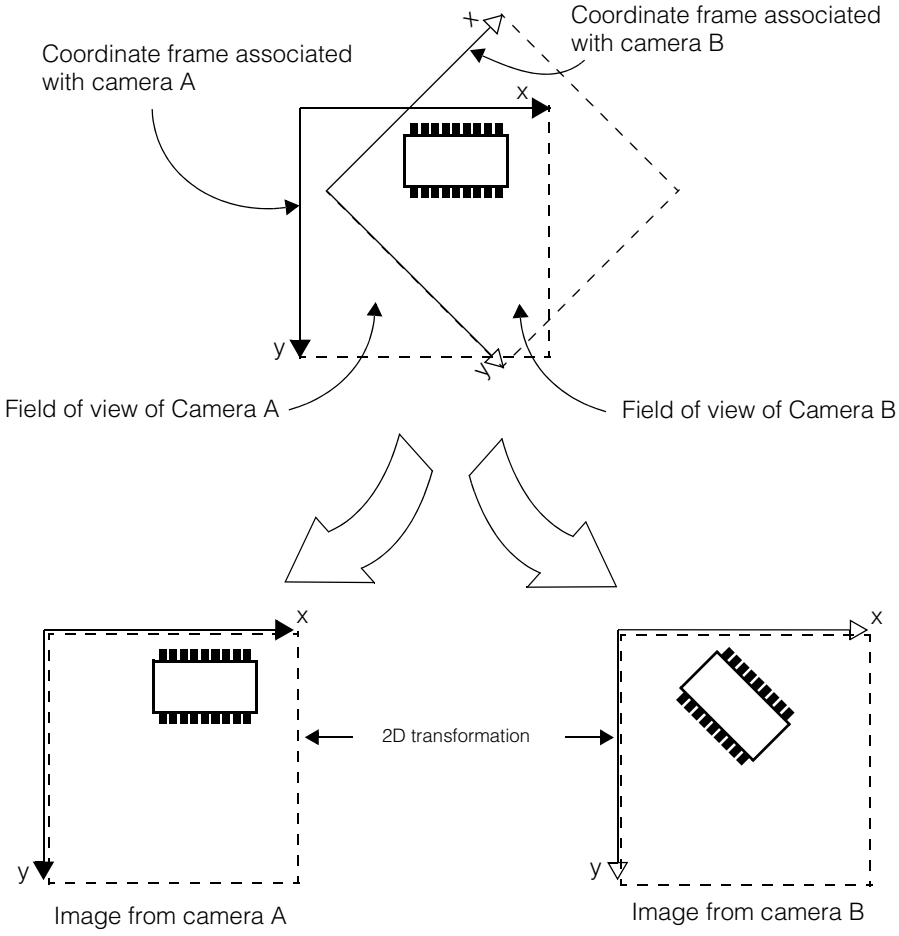


Figure 105. Mapping data between two coordinate frames

The mapping between image coordinate system and robot manipulator coordinate system discussed in the introduction is another example of 2D transformation as a mapping between coordinate frames.

Whether you think of moving the device (as in interpretation 1) or the reference frame (as in interpretation 2) or neither (as in interpretation 3), the intent is exactly the same: a transformation allows you to transform one view of the data to become a second view. The mathematics is also the same: the same transformation matrix representation is used regardless of which way you think. The only difference is in the way you specify and interpret the transformation. For example, if you move the reference frame to the right, that is the same as moving the object to the left. Either way you get the same initial picture, the same final picture, and the same mathematical transformation. If you want to know whether this transformation specifies a move to the left, or a move to the right, you need to know whether you are thinking of moving the object or the reference frame. This problem of interpretation affects both the direction of the motion and the order in which multiple motions are made.

Basic 2D Linear Transformations

This section introduces the properties of some basic 2D linear transformations and their mathematical representations. The 2D linear transformations presented will be discussed in the context of mapping a point between two coordinate frames (interpretation (3) in the preceding section). In what follows the source coordinate frame is known as the *from* frame while the destination coordinate frame is the *to* frame. In all the illustrations the *from* frame is depicted as Frame A while the *to* frame is Frame B. A 2D transformation is designed to map points *from* Frame A *to* Frame B. This interpretation is useful in CVL because many machine vision tasks require a mapping between two coordinate systems: the image coordinate system and the client coordinate system (see *Coordinate Systems* on page 231).

Translation

Translations are the simplest form of mapping. They consist of a simple offset of the coordinate frame as shown in Figure 106.

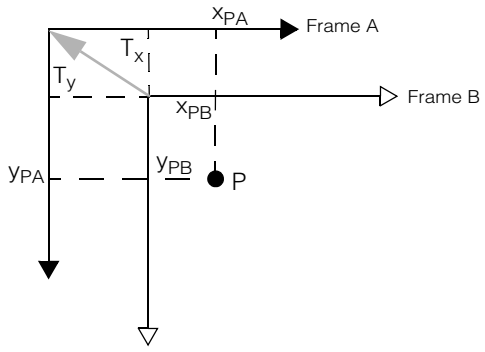


Figure 106. Translation of coordinate frames

If (T_x, T_y) are the components of the translation vector that goes from Frame B to Frame A, expressed in Frame B, then the point P in Frame A is mapped to Frame B by:

$$\begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix} = \begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}$$

The mapping from Frame B to Frame A is given by:

$$\begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix} = \begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix} - \begin{bmatrix} T_x \\ T_y \end{bmatrix}$$

Rotation

Rotations can be of two types, depending on whether the coordinate axes are rotated by the same angle or not. In the first case the rotation is referred to as *rigid*, in the second as *skew*. For simplicity, the two situations are discussed separately, although a rigid rotation is just a special case of a skew rotation.

Rigid Rotation

Coordinate frames can be rotated with respect to each other as shown in Figure 107.

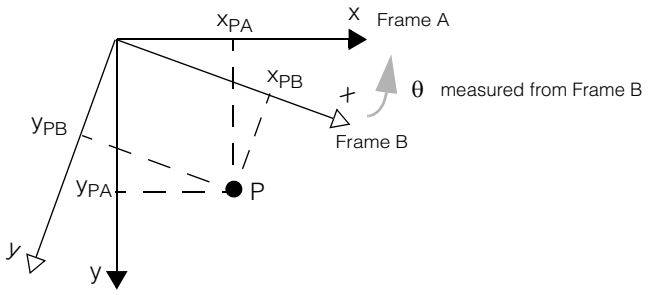


Figure 107. Rigid rotation of coordinate frames

If θ is the rotation angle measured from Frame B (the angle is negative in Figure 107), then the coordinates of P in Frame A are mapped to the corresponding coordinates in Frame B by the following transformation:

$$\begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix}$$

The coordinates of P in Frame B are mapped back to the coordinates in Frame A by the following inverse transformation:

$$\begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix}$$

Notice that the rotation angle θ is the rotation angle *from* Frame B *to* Frame A as measured in Frame B.

Example

Figure 108 shows two frames that are rotated with respect to each other by -45°.

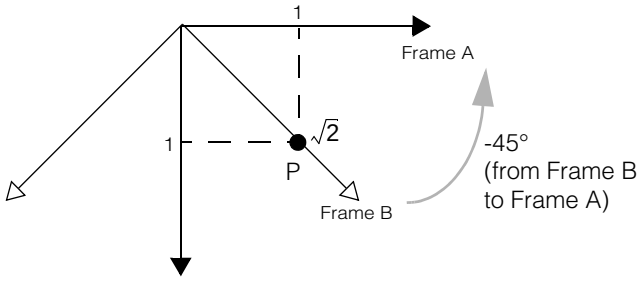


Figure 108. Two frames rotated with respect to each other by -45°

If the coordinates of P in Frame A are (1,1), the coordinates of P in Frame B are:

$$\begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix} = \begin{bmatrix} \cos(-45) & -\sin(-45) \\ \sin(-45) & \cos(-45) \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \sqrt{2} \\ 0 \end{bmatrix}$$

Transformation matrix from Frame A to Frame B — coordinates of P in Frame A — coordinates of P in Frame B

You can use the inverse transformation to go back to the coordinates of P in Frame A:

$$\begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix} = \begin{bmatrix} \cos(-45) & \sin(-45) \\ -\sin(-45) & \cos(-45) \end{bmatrix} \begin{bmatrix} \sqrt{2} \\ 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} \sqrt{2} \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Transformation matrix from Frame B to Frame A — coordinates of P in Frame B — coordinates of P in Frame A

Skew Rotation

In a rigid rotation both the x- and y- axes are rotated by the same angle. In a skew rotation, the coordinate axes may be rotated by a different amount as illustrated in Figure 109.

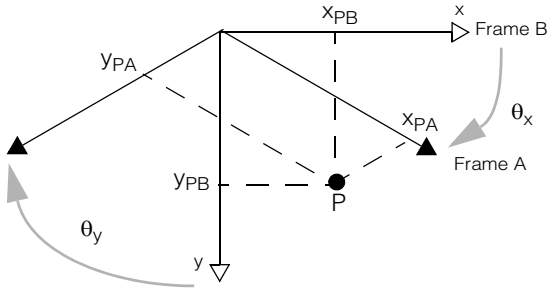


Figure 109. Skew rotation of coordinate frames

If θ_x and θ_y are the rotation angles of the x- and y- axes measured from Frame B to Frame A, the coordinates of the point P in Frame A are mapped to the ones in Frame B by:

$$\begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix} = \begin{bmatrix} \cos\theta_x & -\sin\theta_y \\ \sin\theta_x & \cos\theta_y \end{bmatrix} \begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix}$$

The coordinates of P in Frame B are mapped to the ones in Frame A by:

$$\begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix} = \frac{1}{(\cos\theta_x)(\cos\theta_y) + (\sin\theta_x)(\sin\theta_y)} \begin{bmatrix} \cos\theta_y & \sin\theta_y \\ -\sin\theta_x & \cos\theta_x \end{bmatrix} \begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix}$$

Notice that the angles θ_x and θ_y are the rotation angles *from* Frame B *to* Frame A as measured in Frame B. Frame B must be orthogonal.

Example

Consider the two frames in Figure 110, related to each other by the following skew rotation: $\theta_x = 45°$ and $\theta_y = 90°$.

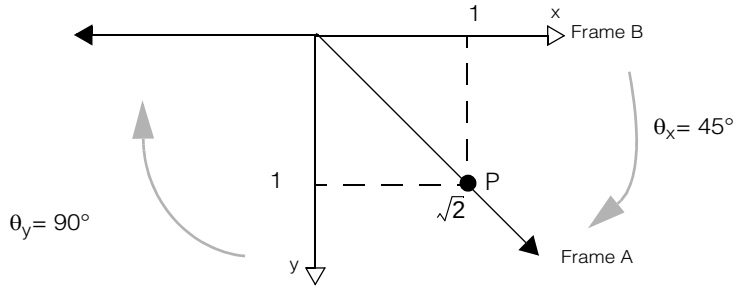


Figure 110. Skew rotation with $\theta_x = 45°$ and $\theta_y = 90°$

If the coordinates of P in Frame A are $(\sqrt{2}, 0)$, the coordinates of P in Frame B are:

$$\begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix} = \begin{bmatrix} \cos(45) & -\sin(90) \\ \sin(45) & \cos(90) \end{bmatrix} \begin{bmatrix} \sqrt{2} \\ 0 \end{bmatrix} = \begin{bmatrix} (1/\sqrt{2}) & -1 \\ (1/\sqrt{2}) & 0 \end{bmatrix} \begin{bmatrix} \sqrt{2} \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

To go back to the coordinates in Frame A:

$$\begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix} = \sqrt{2} \begin{bmatrix} \cos(90) & \sin(90) \\ -\sin(45) & \cos(45) \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \sqrt{2} \begin{bmatrix} 0 & 1 \\ (-1/\sqrt{2}) & (1/\sqrt{2}) \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \sqrt{2} \\ 0 \end{bmatrix}$$

Scaling

This transformation maps two coordinate frames that are a scaled version of each other. The transformation is implemented by the following matrices:

$\begin{bmatrix} s_x & 0 \\ 0 & 1 \end{bmatrix}$ scales the x-coordinate by s_x

$\begin{bmatrix} 1 & 0 \\ 0 & s_y \end{bmatrix}$ scales the y-coordinate by s_y

$\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}$ scales the x- and y- coordinates by s

$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$ scales the x-coordinate by s_x and the y-coordinate by s_y

For example, if Frame B is a scaled version of Frame A ($s_x = s_y = s$), the coordinates of a point P in the two frames are related to each other by the following:

$$\begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix} = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix} = s \begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix}$$

$$\begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix} = \begin{bmatrix} 1/s & 0 \\ 0 & 1/s \end{bmatrix} \begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix} = \frac{1}{s} \begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix}$$

Example

In the following figure, Frame A is an expanded version of Frame B by a factor of 10.

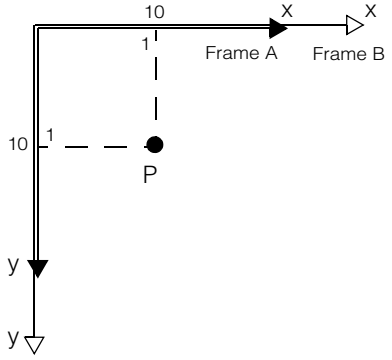


Figure 111. Frame A is an expanded version of Frame B (scaling factor = 10)

If the coordinates of P in Frame A are (1,1), the coordinates of P in Frame B are:

$$\begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix} = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 10 \\ 10 \end{bmatrix}$$

To go back to the coordinates in Frame A:

$$\begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix} = \begin{bmatrix} 1/10 & 0 \\ 0 & 1/10 \end{bmatrix} \begin{bmatrix} 10 \\ 10 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Shear

A shear is a distortion of the coordinate frame that produces a rotation of the y-axis (by a shear angle) and a scaling of the y-axis as illustrated in Figure 112.

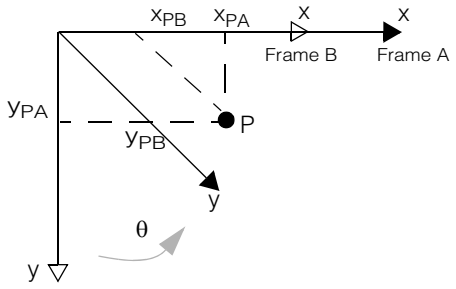


Figure 112. Shear of coordinate frames

If the shear angle is θ (as measured from Frame B), the scaling factor of the x-axis is $\frac{1}{\cos\theta}$. The transformation matrix of the shear can be obtained by composing the following two transformations:

- First scale the y-coordinate by $\frac{1}{\cos\theta}$
- Then rotate the y-axis by θ (a skew rotation with $\theta_x = 0$ and $\theta_y = \theta$)

The coordinates of P in Frame A are mapped to their coordinates in Frame B by:

$$\begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix} = \begin{bmatrix} 1 & -\sin\theta \\ 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & (1/\cos\theta) \end{bmatrix} \begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix} = \begin{bmatrix} 1 & -\tan\theta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix}$$

rotation of the y-axis — scaling of the y-coordinate — x-shear transformation

The mapping from Frame B to Frame A is:

$$\begin{bmatrix} x_{PA} \\ y_{PA} \end{bmatrix} = \begin{bmatrix} 1 & \tan\theta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{PB} \\ y_{PB} \end{bmatrix}$$

General 2D Linear Transformations

All the 2D transformations discussed above are examples of linear transformations. A general linear (or more precisely *affine*) 2D transformation that maps points from frame A to frame B can be represented by using a (2x2) matrix and a (2x1) translation vector.

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_A \\ y_A \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}$$

The translation vector $\begin{bmatrix} T_x \\ T_y \end{bmatrix}$ is the vector from Frame B to Frame A whose components are expressed in Frame B.

Inverse of a Linear 2D Transformation

A very important property of linear transformations concerns the existence of their inverse. The inverse of a transformation that maps points from Frame A to Frame B is the transformation that maps the points back from Frame B to Frame A.

A linear transformation is invertible if and only if the determinant of the matrix component of the transformation is different from 0. The inverse of the matrix M= $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ is: $M^{-1} = \frac{1}{det(M)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \frac{1}{(ad-bc)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$. If the determinant of M is 0, the matrix is called singular and the transformation cannot be inverted.

The matrix obtained by composing M with M^{-1} and M^{-1} with M is the identity matrix:

$$MM^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \left(\frac{1}{(ad-bc)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \right) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$M^{-1}M = \left(\frac{1}{(ad-bc)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \right) \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

If the matrix component of the linear 2D transformation is invertible, then the inverse transformation that maps points from Frame A to Frame B can be simply obtained by using the inverse matrix of M as shown below:

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = M \begin{bmatrix} x_A \\ y_A \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} \Rightarrow M \begin{bmatrix} x_A \\ y_A \end{bmatrix} = \begin{bmatrix} x_B - T_x \\ y_B - T_y \end{bmatrix}$$

$$\Rightarrow M^{-1}M \begin{bmatrix} x_A \\ y_A \end{bmatrix} = M^{-1} \begin{bmatrix} x_B - T_x \\ y_B - T_y \end{bmatrix} \Rightarrow \begin{bmatrix} x_A \\ y_A \end{bmatrix} = M^{-1} \begin{bmatrix} x_B - T_x \\ y_B - T_y \end{bmatrix}$$

The basic 2D transformations discussed above are all invertible if the appropriate transformation parameters are chosen (for example, you cannot invert a scaling transformation if the scaling parameter s = 0). The transformation that maps Frame B to Frame A is the inverse of the transformation that maps Frame A to Frame B and vice versa.

Rigid Transformations

Rigid transformations are 2D linear transformations defined by a rotation matrix and a translation vector. They are implemented in CVL by the **cc2Rigid** class. Figure 113 shows an example of two frames that are related to each other by a rigid transformation.

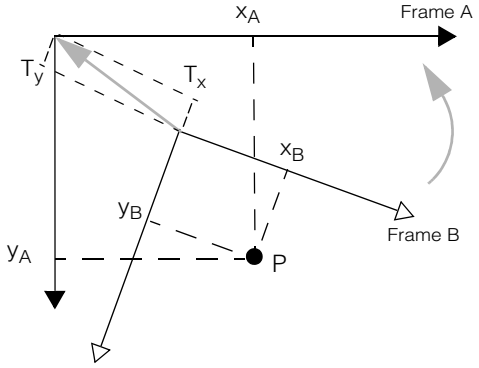


Figure 113. Rigid transformation of coordinate frames

If θ is the rotation angle from Frame B and (T_x, T_y) are the components of the translation vector from Frame B to Frame A, as measured in Frame B, the mapping of a point from Frame A to Frame B is:

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_A \\ y_A \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}$$

The mapping from Frame B back to Frame A is:

$$\begin{bmatrix} x_A \\ y_A \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_B - T_x \\ y_B - T_y \end{bmatrix}$$

Constructing the Matrix of a 2D Linear Transformation

To obtain the matrix form of a 2D linear transformation M, it is only necessary to know how the points (0, 1) and (1, 0) are mapped by the transformation. The first column of the transformation matrix M is, in fact, equal to M[(1, 0)] while the second column of the transformation matrix is equal to M[(0, 1)]. For example, to obtain the matrix of the transformation that maps points between two frames rotated with respect to each other by θ (see Figure 114), you only need to know how the points (0, 1) and (1, 0) are mapped. In this case (1,0) is mapped to (cosθ, –sinθ) while (0, 1) is mapped to (cosθ, sinθ).

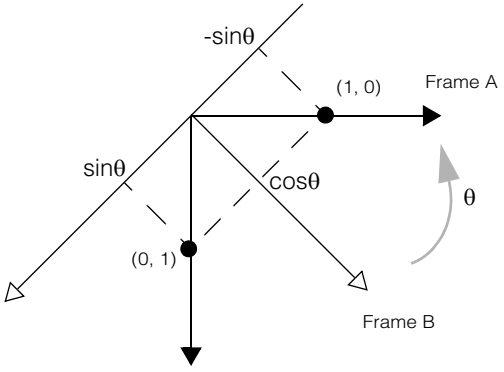


Figure 114. Two coordinate frames rotated by θ. The point (1, 0) is mapped to (cosθ, –sinθ) and the point (0, 1) is mapped to (cosθ, sinθ)

The transformation matrix is then:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

M[(1, 0)] M[(0,1)]

2D Linear Transformations in CVL

There are four classes in CVL that allow you to implement a 2D linear transformation between coordinate frames. They are listed in the following table.

Class	Transformation
cc2Matrix	Implements a general (2x2) transformation matrix
cc2XformLinear **cc2Xform**	Implement a generic linear transformation
cc2Rigid	Implements a rigid transformation

Table 61. CVL transformation classes.

cc2Matrix

You can use **cc2Matrix** to define the four elements of your transformation matrix. This class also lets you specify the transformation matrix according to two modalities called the scale-rotation and the shear-aspect mode (see also *Client Coordinate Transforms* on page 237).

Scale-Rotation Mode

When you are using **cc2Matrix** in the scale-rotation mode, the transformation implemented by the class is:

$$\begin{bmatrix} \cos\theta_x & -\sin\theta_y \\ \sin\theta_x & \cos\theta_y \end{bmatrix} \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} = \begin{bmatrix} s_x\cos\theta_x & -s_y\sin\theta_y \\ s_x\sin\theta_x & s_y\cos\theta_y \end{bmatrix}$$

Skew rotation of the coordinate frame

Scaling of the coordinate frame

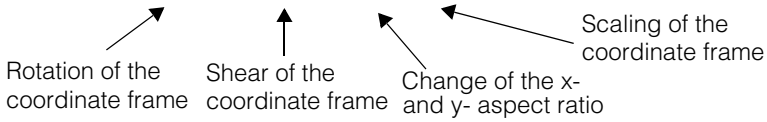
You can specify the parameters s_x, s_y, θ_x, θ_y and the class constructor will automatically set the transformation matrix to the transformation above.

Note In the header files you will see the angles θ_x and θ_y represented as r_x and r_y. This is because r_x and r_y are more easily rendered in ASCII source code than θ_x and θ_y.

Shear-Aspect Mode

When you are using **cc2Matrix** in the shear-aspect mode, the transformation implemented is the following:

$$\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}1 & -\tan K\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & A\end{bmatrix}\begin{bmatrix}S & 0\\ 0 & S\end{bmatrix} = \begin{bmatrix}S\cos\theta & AS(-\sin\theta - \cos\theta\tan K)\\ S\sin\theta & AS(\cos\theta - \sin\theta\tan K)\end{bmatrix}$$



You can specify the parameters S, θ, K, A and the class constructor will automatically set the transformation matrix to the transformation above.

Note In the header files you will see the angle θ represented as R. This is because R is more easily rendered in ASCII source code than θ.

cc2XformLinear

This class is derived from the **cc2XformBase** class and implements a general linear transformation. You can directly specify the elements of the transformation matrix component by component or according to the scale-rotation/shear-aspect specification mode. The mapping performed by **cc2XformLinear::mapPoint()** is:

$$\begin{bmatrix}x_B\\ y_B\end{bmatrix} = \begin{bmatrix}a & b\\ c & d\end{bmatrix}\begin{bmatrix}x_A\\ y_A\end{bmatrix} + \begin{bmatrix}T_x\\ T_y\end{bmatrix}$$

This function will return the right values only if the translation vector components that you specify are the ones *from* Frame B *to* Frame A expressed in Frame B (see also the example in the next section on **cc2Rigid**).

cc2Xform

This class implements a general linear transformation in exact the same way as **cc2XformLinear**.

cc2Rigid

This class implements a rigid transformation. The mapping performed by the member function **cc2Rigid::mapPoint()** is:

$$\begin{bmatrix}x_B\\ y_B\end{bmatrix} = \begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}x_A\\ y_A\end{bmatrix} + \begin{bmatrix}T_x\\ T_y\end{bmatrix}$$

The function will return the right values only if the rotation angle and the translation vector are specified in the following way:

- The rotation angle is the angle by which the coordinate axes are rotated *from* frame B as measured in Frame B.
- The components of the translation vector are the components of the vector *from* Frame B *to* Frame A as measured in Frame B.

Example.

Suppose you want to use **cc2Rigid** to specify a mapping from an orthogonal Frame A to an orthogonal Frame B that is rotated by + 45° from Frame A and translated by (1, 1) from Frame A (see Figure 115).

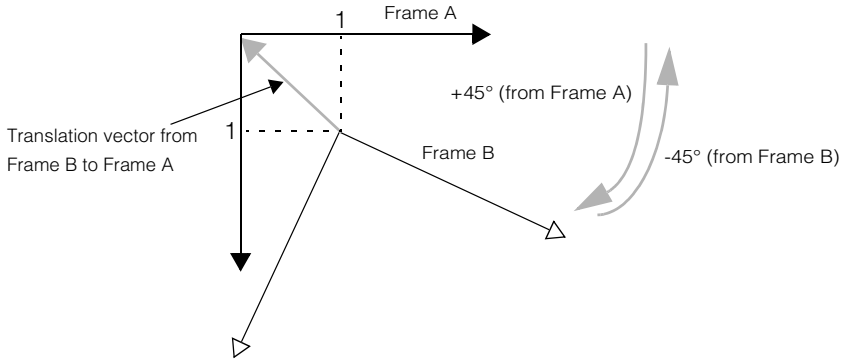


Figure 115. Two frames rigidly transformed with respect to each other

If you want **mapPoint()** to return the correct values, you need to specify the parameters in the following way:

- θ = -45° because the rotation angle *from* Frame B is -45°
- If (1, 1) are the components of the translation vector from Frame A to Frame B in Frame A, the components of the translation vector from Frame B to Frame A in frame A are (-1, -1). To enter the appropriate translation components, you still need to obtain the components of (-1, -1) in Frame B. These are given by:

$$\begin{bmatrix} \cos(-45) & -\sin(-45) \\ \sin(-45) & \cos(-45) \end{bmatrix} \begin{bmatrix} -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} -1 \\ -1 \end{bmatrix} = \begin{bmatrix} -\sqrt{2} \\ 0 \end{bmatrix}$$

. The components of the translation vector to be entered are then: ($-\sqrt{2}$, 0)

If the parameters are entered in this way, the call to **mapPoint()** will return the right mapping. For example the coordinates of (1,1) in Frame A are correctly mapped to (0, 0) in Frame B.

$$\begin{bmatrix}\cos(-45) & -\sin(-45)\\ \sin(-45) & \cos(-45)\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}+\begin{bmatrix}-\sqrt{2}\\0\end{bmatrix}=\begin{bmatrix}1/\sqrt{2} & 1/\sqrt{2}\\ -1/\sqrt{2} & 1/\sqrt{2}\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}+\begin{bmatrix}-\sqrt{2}\\0\end{bmatrix}=\begin{bmatrix}0\\0\end{bmatrix}$$

An alternative way to compute the same **cc2Rigid** transformation would be to first define a transformation that maps points from Frame B to Frame A and then invert it. The transformation that maps points from Frame B to Frame A can be constructed easily because the translation vector is simply $\begin{bmatrix}1\\1\end{bmatrix}$ and $\theta = +45°$.

Nonlinear Transformations

In several machine vision applications, such as SMD placement, a very accurate mapping between image and client coordinates is necessary. Although a simple linear mapping is often good enough for many machine vision applications, there are instances when the nonlinear distortions introduced by the optics of the camera cannot be neglected and a simple linear transformation cannot achieve the desired level of mapping accuracy. Two CVL classes implement nonlinear transforms for this purpose. The **cc2XformPoly** class implements a nonlinear polynomial mapping that allows you to correct for camera lens distortion as well as nonlinear distortion in the acquisition system electronics. The **cc2XformCalib2** class adds intrinsic and extrinsic parameters that allow you to correct for radial and perspective distortion

The next section discusses the polynomial transformation implemented by **cc2XformPoly** and the **cc2XformCalib2** transform is discussed in *Perspective-Polynomial Transforms* on page 432.

Polynomial Transforms

This section discusses polynomial transforms.

Definition of a Polynomial Transformation

If (I_x, I_y) and (C_x, C_y) are the image and client coordinates of generic image features, the mapping performed by a polynomial transformation is defined by the following two equations.
Where a_{ij} and b_{ij} are the polynomial coefficients and n is the degree of the polynomial transformation. For example, the form of a third degree polynomial is:

$$C_x = a_{00} + a_{01}I_y + a_{02}I_y^2 + a_{03}I_y^3 + a_{10}I_x + a_{11}I_xI_y + a_{12}I_xI_y^2 + a_{20}I_x^2 + a_{21}I_x^2I_y + a_{30}I_x^3$$

$$C_y = b_{00} + b_{01}I_y + b_{02}I_y^2 + b_{03}I_y^3 + b_{10}I_x + b_{11}I_xI_y + b_{12}I_xI_y^2 + b_{20}I_x^2 + b_{21}I_x^2I_y + b_{30}I_x^3$$

$$C_x(I_x, I_y) = \sum_{\substack{i+j \le n \\ i,j \ge 0}} a_{ij} I_x^{\,i} I_y^{\,j} \qquad C_y(I_x, I_y) = \sum_{\substack{i+j \le n \\ i,j \ge 0}} b_{ij} I_x^{\,i} I_y^{\,j}$$

In the case of a third degree polynomial, the number of coefficients that define the transformation is 20. In general, given a polynomial of order n, the number of coefficients of the polynomial transformation is $(n+1)(n+2)$.

Fitting a Polynomial Transformation

One of the constructors of **cc2XformPoly** determines the coefficients of the polynomial transformation by implementing a least-square fit on a set of image points specified by the user. For the fitting procedure you need to supply **cc2XformPoly** with the coordinates of two sets of points: the image coordinates of some image features and their corresponding client coordinates (see Figure 116).

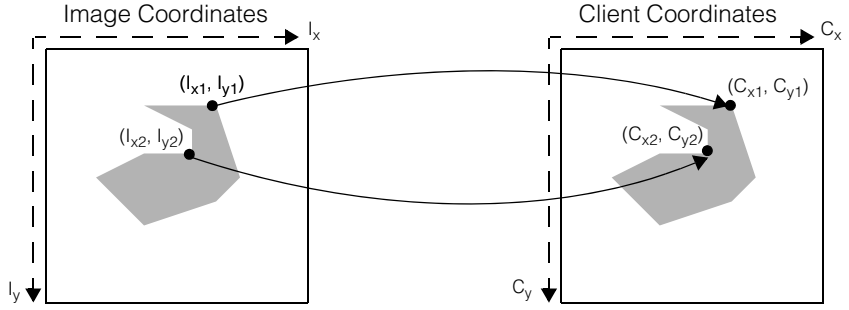


Figure 116. Image and client coordinates of points to be used for the least-square fit

The number of points needed for the fitting depends on the order of the polynomial you want to use. If the order of the polynomial is n, you need a minimum of $(n+1)(n+2)/2$ points:

Image Coordinates	Client Coordinates
(I_{x1}, I_{y1})	(C_{x1}, C_{y1})
(I_{x2}, I_{y2})	(C_{x2}, C_{y2})
.....	.....

Image Coordinates	Client Coordinates
.....	.....
(I_{xk}, I_{yk})	(C_{xk}, C_{yk})

where k must be at least $(n + 1)(n + 2)/2$. Given the two sets of points, **cc2XformPoly** will determine the coefficients that provide the best least-square fit of the polynomial transformation to the points. The accuracy of the fit depends on the number of points you supply. The value $(n + 1)(n + 2)/2$is just a lower-bound limit, the more points you supply the better the fit. Another important factor is how the points to be fit are distributed in the image. If the points you supply to **cc2XformPoly** are concentrated in just a small region, the polynomial fit will provide an accurate mapping only for the points within that region. It is very important that the points you supply span the entire image or region of interest.

Linearizing a Polynomial Transformation

The **linearXform** member function of **cc2XformPoly** allows you to derive a linear transformation from a polynomial transformation. This section illustrates how this is done. Consider the following polynomial transformation of order n:

$$C_x(I_x, I_y) = \sum_{i,j \geq 0}^{i+j \leq n} a_{ij} {I_x}^i {I_y}^j \qquad C_y(I_x, I_y) = \sum_{i,j \geq 0}^{i+j \leq n} b_{ij} {I_x}^i {I_y}^j$$

C_x and C_y are functions of two variables. It is possible to express each of them in terms of their Taylor's series expansion around a point. For example, if the point around which the functions are expanded is (I_{xO}, I_{yO}) the Taylor expansion of C_x will be:

$$C_x(I_x, I_y) = C_x(I_{xo}, I_{yo}) + (I_x - I_{xo})\left(\frac{\partial C_x}{\partial I_x}\bigg|_{(I_{xo}, I_{yo})}\right) + (I_y - I_{yo})\left(\frac{\partial C_x}{\partial I_y}\bigg|_{(I_{xo}, I_{yo})}\right) + ...$$

where the ellipses represent all the higher-order components of the expansion. The basic step in deriving the linear transformation is to neglect all these higher order components and retain only the ones that are linear. The equation above can then be rearranged as follows:

$$C_x(I_x, I_y) = C_x(I_{xo}, I_{yo}) + I_x\left(\frac{\partial C_x}{\partial I_x}\bigg|_{(I_{xo}, I_{yo})}\right) + I_y\left(\frac{\partial C_x}{\partial I_y}\bigg|_{(I_{xo}, I_{yo})}\right) - I_{xo}\left(\frac{\partial C_x}{\partial I_x}\bigg|_{(I_{xo}, I_{yo})}\right)$$
$$- I_{yo}\left(\frac{\partial C_x}{\partial I_y}\bigg|_{(I_{xo}, I_{yo})}\right)$$

This expression contains several constant terms. If $\left(\frac{\partial C_x}{\partial I_x}\Big|_{(I_{xo}, I_{yo})}\right) = A$,

$\left(\frac{\partial C_x}{\partial I_y}\Big|_{(I_{xo}, I_{yo})}\right) = B$ and $C_x(I_{xo}, I_{yo}) - I_{xo}\left(\frac{\partial C_x}{\partial I_x}\Big|_{(I_{xo}, I_{yo})}\right) - I_{yo}\left(\frac{\partial C_x}{\partial I_y}\Big|_{(I_{xo}, I_{yo})}\right) = K$

then $C_x(I_x, I_y) = AI_x + BI_y + K$. The same derivation can be done for $C_y(I_x, I_y)$: $C_{xy}(I_x, I_y) = A'I_x + B'I_y + K'$, where A', B' and K' are the constant terms equivalent to A, B and K (just replace C_x with C_y in the expressions for A, B and K).

The polynomial transformation around the point (I_{xo}, I_{yo}) can, then, be approximated by the following linear mapping:

$$C_x(I_x, I_y) = AI_x + BI_y + K$$
$$C_y(I_x, I_y) = A'I_x + B'I_y + K'$$

In matrix form:

$$\begin{bmatrix} C_x \\ C_y \end{bmatrix} = \begin{bmatrix} A & B \\ A' & B' \end{bmatrix} \begin{bmatrix} I_x \\ I_y \end{bmatrix} + \begin{bmatrix} K \\ K' \end{bmatrix}$$

The matrix $\begin{bmatrix} A & B \\ A' & B' \end{bmatrix}$ and the vector $\begin{bmatrix} K \\ K' \end{bmatrix}$ form the linear transformation returned by **cc2XformPoly::linearXform()**. The matrix $\begin{bmatrix} A & B \\ A' & B' \end{bmatrix}$ is the Jacobian matrix of the polynomial transformation.

If you want to use this linear transformation, you must be aware that the linear approximation gives accurate mappings only for points that are close to (I_{xo}, I_{yo}).

Perspective-Polynomial Transforms

A perspective-polynomial transform is the composition of a perspective transform, a radial transform, and an affine transform (see Figure 117). The composition of the radial transform and the affine transform yields a polynomial transform.

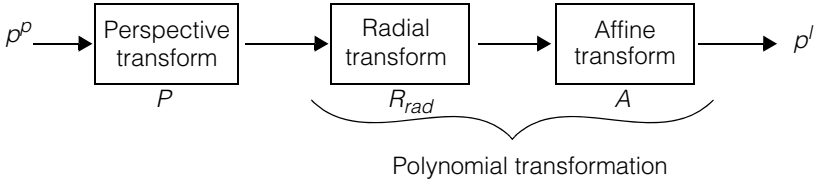


Figure 117. Perspective-polynomial transform

A perspective-polynomial transform transforms points in physical space (p^p) to points in image space (p^i). It is also sometimes called a transformation of a *world point* to an image point. A perspective-polynomial transform is given by:

$$p^i = A\, R_{rad}\, P\, p^p$$

The inverse of the calibration transform describe above can be taken by applying inverses of the three transformations A, R_{rad}, and P in reverse order.

The following sections cover the mathematical basis for these three transforms. We assume throughout this analysis that the camera is a pinhole camera. The notation in Figure 118 is used to describe each transform. The notation ≡ simply means that the right hand side is just another name for the left hand side.

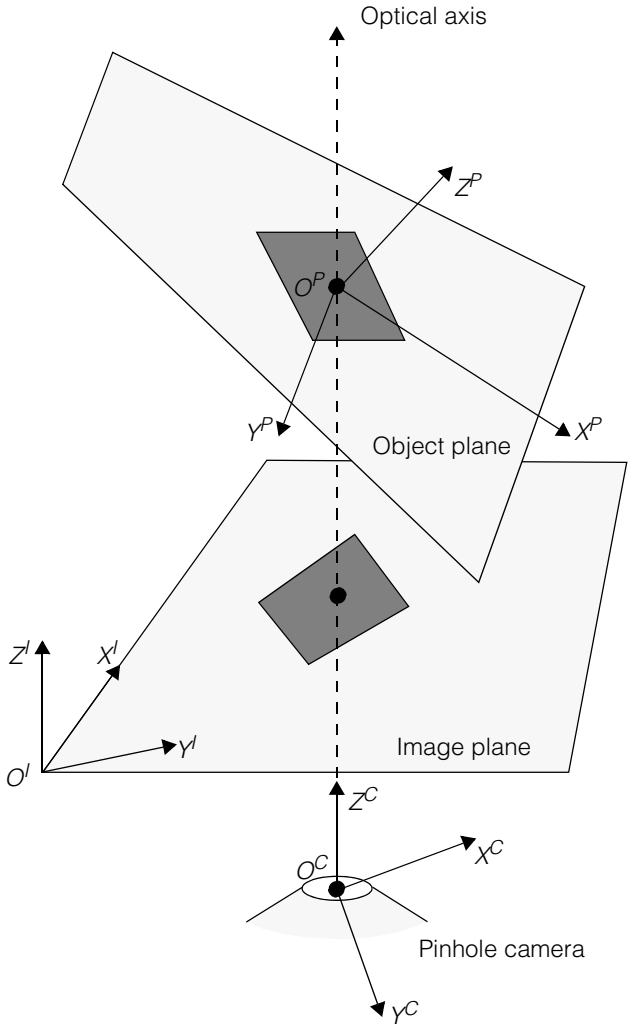


Figure 118. Perspective-polynomial model

Perspective Transform Model

In Figure 118 above, (O^P, X^P, Y^P, Z^P) is the physical (or world) coordinate system. The plane containing axes X^P, Y^P is the object plane containing a planar object. The focal point of the camera is at O^C. The camera coordinate system is (O^C, X^C, Y^C, Z^C). The image is formed in the image plane which is parallel to the plane containing the axes X^C, Y^C. The coordinate system (O^I, X^I, Y^I, Z^I) is the image coordinate system.

The optical axis (or line-of-sight) is the line through the pinhole and perpendicular the image plane. Both the axes Z^C and Z^I, are in the direction of the optical axis.

Let the 2D coordinates of point P^P in the physical coordinate system on the object plane be $\begin{bmatrix} x \\ y \end{bmatrix}$. We will find the 3d coordinates of P^P in the camera coordinate system.

The physical coordinate system (O^P, X^P, Y^P, Z^P) can be transformed to the camera coordinates system (O^C, X^C, Y^C, Z^C) by a 3d rotation followed by a translation.

Let the angles of these rotations be:

r_y = Angle of rotation of axes keeping y-axis fixed
r_x = Angle of rotation of axes keeping x-axis fixed
r_z = Angle of rotation of axes keeping z-axis fixed

The matrix R_{rot} for these three rotations is the following 3x3 orthogonal matrix:

$$R_{rot} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(r_x) & -\sin(r_x) \\ 0 & \sin(r_x) & \cos(r_x) \end{bmatrix} \begin{bmatrix} \cos(r_y) & 0 & \sin(r_y) \\ 0 & -1 & 0 \\ -\sin(r_y) & 0 & \cos(r_y) \end{bmatrix} \begin{bmatrix} \cos(r_z) & -\sin(r_z) & 0 \\ \sin(r_z) & \cos(r_z) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Let the 3d translation of origin O^P to origin O^C be t. Then the 3d coordinates of P^P in the coordinates system are:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = R_{rot} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} + t$$

Now, the perspective transform P of $\begin{bmatrix} x \\ y \end{bmatrix}$ is defined as:

$$P \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} X/Z \\ Y/Z \end{bmatrix} \equiv \begin{bmatrix} \tilde{x} \\ \tilde{y} \end{bmatrix}$$

Radial Distortion Model

The radial transform R_{rad} is specified by a distortion coefficient κ_1 and defined by:

$$R_{rad}\begin{bmatrix}\tilde{x}\\ \tilde{y}\end{bmatrix} = \begin{bmatrix}\tilde{x} + \kappa_1\,\tilde{x}(\tilde{x}^2 + \tilde{y}^2)\\ \tilde{y} + \kappa_1\,\tilde{y}(\tilde{x}^2 + \tilde{y}^2)\end{bmatrix} \equiv \begin{bmatrix}\hat{x}\\ \hat{y}\end{bmatrix}$$

Affine Transform

The affine transform *A* is specified by *x,y*, scales s_x and s_y, skew k, and translations t_x and t_y. It does not contain any rotation of the *x,y* axes. The transform is defined by:

$$A\begin{bmatrix}\hat{x}\\ \hat{y}\end{bmatrix} = \begin{bmatrix}s_x & k\\ 0 & s_y\end{bmatrix}\begin{bmatrix}\hat{x}\\ \hat{y}\end{bmatrix} + \begin{bmatrix}t_x\\ t_y\end{bmatrix} \equiv \begin{bmatrix}\bar{x}\\ \bar{y}\end{bmatrix} \equiv P^I$$

Degrees of freedom

- The perspective transform contains a third rotation (which can be specified by three angles) and a third translation. Therefore this transform contains six degrees of freedom.
- The radial distortion model is based on the coefficient of distortion, thus it contains one degree of freedom.
- The affine transform contains two scale parameters, one skew parameter and two translation parameters. Thus, this transform contains five degrees of freedom.

Adding these independent degrees of freedom, we conclude that the model of calibration transform that we are using contains 6 + 1 + 5 = 12 degrees of freedom.

Extrinsic and Intrinsic Parameters

The perspective transform depends on the extrinsic parameters which are orientation and translation of the world coordinate system with respect to camera coordinate system. The radial and affine transforms depend on the intrinsic parameters; radial distortion coefficient, scales, skew and translations.

References

- *Multiple View Geometry In Computer Vision*, Richard Hartley and Andrew Zisserman, Cambridge University Press, 2000.
- *The Geometry of Multiple Images: The Laws That Govern The Formation of Images of A Scene and Some of Their Applications*, Olivier Faugeras, Quang-Tuan Luong, T. Papadopoulou, MIT Press 2001.

Writing Multithreaded CVL Applications

11

CVL enables you to write a multithreaded application. This chapter contains information you need to know to write multithreaded applications using the Cognex Vision Library (CVL).

Using Threads with CVL discusses reasons to use multithreading in a CVL application and recommends using the CVL threads API.

Calling MFC Functions from Non-MFC Threads discusses changes in the behavior of non-MFC threads in Microsoft Visual C++ .NET. Be sure to read this section if you are porting your application to Visual C++ .NET.

CVL's Threading Interface describes the functions and classes in CVL's threading interface.

Thread Creation discusses creating threads and avoiding common problems.

Thread Cleanup discusses ensuring that memory that CVL allocates internally per thread is cleaned up when the thread exits.

CVL Objects Require Apartment Threading discusses the fact that CVL classes are not inherently thread safe, and that access to resources by multiple threads must be synchronized.

Thread Synchronization Objects discusses the mutex, semaphore, and event objects available in CVL for synchronizing threads.

Requirements and Recommendations provides some guidance in the proper use of threads in a CVL application.

Multi-Core and Multi-Processor Systems provides information on how to write your CVL application to take advantage of the capabilities of multi-core and multi-processor systems.

Multiprocess Applications discusses a subject related to multithreading, the use of multiple processes in your vision processing application.

Using Threads with CVL

This section discusses why you might use threads in your CVL vision processing application, and recommends using the CVL threads API over other thread control APIs.

Why Use Multithreading?

Common reasons for making an application multithreaded include:

- Handling situations that involve waiting for external events, such as image acquisition
- Keeping a GUI responsive while your application is engaged in other work
- Using thread priorities to make sure the most important code executes first
- Distributing tasks among processors in machines with multiple CPUs

Use multiple threads only when they offer clear advantages. A multithreaded application is more complicated to debug, to maintain, and to test than an equivalent single-threaded application.

Use the CVL Threads API

When you do use threads, Cognex recommends using the CVL threads API discussed in this chapter instead of the Win32 thread control classes or a third party class library's classes. This recommendation is for the following reasons:

- When you start a thread with the CVL threads API, data structures are allocated for use by CVL functions. When the thread exits, those resources are deallocated automatically.
- If you start a thread with any other API, and then call CVL functions in that thread, you must still call a CVL threads API to clean up CVL resources, and this call must come at the right moment before the thread exits.
- The CVL threads API runs on both host and embedded systems.

Calling MFC Functions from Non-MFC Threads

When porting CVL applications to Visual C++ .NET, be aware that MFC will now assert if you make an MFC function call from a thread that is not an MFC thread. You can create MFC threads with the MFC function **AfxBeginThread()** or the CVL global function **cfCreateThreadMFC()**, explained below.

The implication for CVL is that when using Visual C++ .NET you cannot make any **ccDisplayConsole** function calls from threads that were created with **cfCreateThread()**. If you do, you may see asserts from deep in MFC with no clear indication of what the problem is or how to fix it. Moreover, it is possible to write code that is not thread safe and forget to call the **cfThreadCleanup()** function to clean up any memory after a thread is destroyed.

To solve these problems, the following changes have been made to the CVL thread creation API:

- The new **cfCreateThreadMFC()** function allows creation of MFC worker threads using the CVL thread creation API. This makes it easier and less error-prone to update your applications when you move to Visual C++ .NET.
- The **cfCreateThread()** function has been renamed to **cfCreateThreadCVL()**. The **cfCreateThread()** function is still available, but is deprecated. When compiling with Visual C++ .NET, any use of **cfCreateThread()** will cause a compiler warning (the new Visual C++ .NET *deprecated function* warning).

Always use **cfCreateThreadMFC()** to create MFC threads if your application is an MFC application and you are making calls to MFC functions (or **ccDisplayConsole**) from within the created thread.

See also *Upgrading Visual C++ 6.0 Projects to .NET* on page 33.

CVL's Threading Interface

Table 62 briefly describes the CVL classes and functions used for multithreading.

Class or function	Notes
cfCreateThreadCVL()	Starts a new non-MFC thread at a specified priority. Any resources allocated internally by CVL for this thread are cleaned up automatically when the thread exits. Use this function for non-MFC applications.
cfCreateThreadMFC()	Starts a new MFC thread at a specified priority. Any resources allocated internally by CVL for this thread are cleaned up automatically when the thread exits. Use this function for MFC applications, or if you will be using any CVL component (such as **ccDisplayConsole**) that uses MFC.
cfThreadCleanup()	Must be called at the termination of all threads that call CVL functions and were not created using **cfCreateThreadCVL()** or **cfCreateThreadMFC()**.
cc_Resource	Base class for **ccMutex**, **ccSemaphore**, and **ccEvent**. The **cc_Resource::breakLocks()** method can be used to release a thread you are about to terminate.
ccMutex	A resource that can be locked one or more times by a single thread.
ccSemaphore	A resource that can be locked by one or more threads, up to a specified count.
ccEvent	A resource that describes an event. An event can have two states, set and reset. A thread that locks the event can change its state. One or more threads can block while waiting for an event's state to change from reset to set.
ccLock	Sets a lock on a mutex, semaphore, or event. Destructor automatically unlocks, so unlocking is assured.

Table 62. CVL Interface for multithreading

Class or function	Notes
ccCriticalSection	An object that may be owned by only one thread of execution at a time. Critical sections are a lightweight, faster equivalent of mutexes.
ccCriticalSectionLock	Sets a lock on a critical section. Destructor automatically unlocks, so unlocking is assured.
ccThreadID	Platform-independent thread identifier.
ccThreadLocal	A parameterized class that allocates thread-safe global storage at static initialization time.
cfSetThreadPriority(), cfGetThreadPriority(), cfGetCurrentThreadID()	Functions for manipulating thread priority and getting a thread ID.
cfWaitForThreadTermination()	Blocks execution until a specific thread terminates.

Table 62. CVL Interface for multithreading

Thread Creation

You can call CVL functions in threads that you create using the CVL function **cfCreateThreadCVL()**, **cfCreateThreadMFC()**, or the Windows runtime library functions **_beginthread()** or **AfxBeginThread()**. All of these functions take as their arguments a pointer to a thread-entry function. Thread-entry functions take a single void pointer argument, which you can use to pass starting information to the thread.

Note Do not call CVL functions in threads created by the Win32 function **CreateThread()**. This function does not initialize the Windows runtime library, and should not be used in CVL applications.

Threads you create using **cfCreateThreadCVL()** or **cfCreateThreadMFC()** will, on exit, automatically clean up any per-thread resources that CVL allocates internally. Threads you create using other functions require an explicit, and precisely positioned, call to **cfThreadCleanup()**.

Common Problems with Thread Creation

Keep in mind that a thread-creation function can return before the thread begins executing. The following code demonstrates this problem.

```
// Code example showing INCORRECT way to create threads

void myThreadEntry(void* args)
{
  int myThreadNumber = *((int*)args);
  // etc...
}

void startThreads()
{
  // start three threads, passing each its thread number
  int n;
  for (n=0; n<3; ++n)
    cfCreateThreadCVL(myThreadEntry, &n);
}
```

The first problem is that **cfCreateThreadCVL()** (or any other thread-creation function) returns immediately after starting a new thread. There is no guarantee that the entry point you supply for the new thread will have been called by the time the thread creation function returns. The second problem is that the value of *n* may have changed by the time **myThreadEntry()** retrieves it. Or, if **startThreads()** returns after the last thread is created, *n* will have gone out of scope.

The safest and most extensible way to start a thread is to pass a heap-allocated structure of starting information to the thread-entry function and have the newly started thread delete the structure after using it.

```
// Code example showing CORRECT way to create threads

struct ThreadStartupInfo
{
  ThreadStartupInfo(int nval, const std::string& str);
  int n;
  std::string cmd;
};

void myThreadEntry(void* args)
{
  ThreadStartupInfo* myInfo = (ThreadStartupInfo*)args;
  // perform thread startup actions here
  delete myInfo;  // delete when done with it
}

void startThreads()
{
  // start three threads, passing each of them startup info
  int n;
  for (n=0; n<3; ++n)
  {
    ThreadStartupInfo* info = new ThreadStartupInfo(n, "");
    cfCreateThreadCVL(myThreadEntry, info);
  }
}
```

Thread Cleanup

In many cases, CVL allocates internal memory for each thread in which CVL functions are called. This memory must be freed when the thread exits. Threads created by **cfCreateThreadCVL()** or **cfCreateThreadMFC()** perform this cleanup automatically. If you do not use one of these functions to create a thread, you must call **cfThreadCleanup()** as the last CVL call before the thread exits.

Problem When cfCreateThreadCVL() Is Not Used

The following code example shows the call to **cfThreadCleanup()** that is required if you do not use **cfCreateThreadCVL()** or **cfCreateThreadMFC()**.

```
// Code example showing INCORRECT way to call cfThreadCleanup()

myThreadProc(void *arg)
{
  ccPelBuffer<c_UInt8> myImage;
    ...
  cfThreadCleanup();
}
```

cfThreadCleanup() is the last call made before the thread exits; however, there is a problem with *myImage*. This CVL object was allocated on the stack, so its destructor will be called after **cfThreadCleanup()**. This violates the rule that **cfThreadCleanup()** must be the last CVL call in the thread. Using an extra scope fixes this problem:

```
// Code example showing CORRECT way to call cfThreadCleanup()

myThreadProc(void *arg)
{
  {
  ccPelBuffer<c_UInt8> myImage;
    ...
  } // End of extra scope -- myImage destroyed here
  // Now safe to call cfThreadCleanup()
  cfThreadCleanup()
}
```

CVL Objects Require Apartment Threading

With the exception of certain threading support classes, none of the classes in CVL should be considered thread safe. Essentially all CVL classes require the use of the apartment threading model. For a given instance of a CVL class, only a single thread can use the instance at one time.

This means that:

- Multiple threads can use different instances of a vision tool object at the same time.
- Multiple threads can use the same instance of a vision tool object at different times.
- Multiple threads *cannot* use the same instance of a vision tool object at the same time.

You can use thread synchronization to prevent the unsupported case from occurring.

Thread Synchronization Objects

CVL offers several mechanisms for synchronizing threads: mutexes, critical sections, semaphores, and events. All of these mechanisms involve locking and unlocking thread synchronization objects.

Mutex

A mutex object (**ccMutex**) is generally used to protect either an object or a section of code from simultaneous access by multiple threads. Only one thread can lock the mutex at a time. Another thread attempting to lock the resource blocks until the number of locks held by the first thread is zero, or a specified timeout is reached.

Critical Section

Like a mutex, a critical section object (**ccCriticalSection**) also represents an object or section of code that must be protected against simultaneous access.

Critical sections differ from mutexes in the following ways:

- Locking and unlocking is faster with critical sections than with mutexes. For example, the combined time of locking and unlocking can take 0.05 microseconds for a critical section as opposed to 3.2 microseconds for a mutex.

Note Unlike other synchronization objects, the Win32 implementation of **ccCriticalSection** does not require the operating system to switch from user mode to kernel mode on lock or unlock. The lack of a context switch provides a performance improvement, at the expense of waitability and interprocess visibility.

- You claim a **ccCriticalSection** with a **ccCriticalSectionLock**, whereas you claim a **ccMutex** with a **ccLock**. However, the locking/unlocking behavior is identical.
- Critical sections do not support timeouts. Their timeout is the equivalent of an infinite timeout on a mutex. Therefore, critical sections can block indefinitely.
- A critical section is not a **cc_Resource** and, therefore, will not respond to a **cc_Resource::breakLock()** call (by throwing *BrokenLock*). This means that you cannot use a **ccCriticalSection** for the usual thread shutdown techniques used with a **ccMutex**.
- A critical section has no name and cannot be shared among processes as it has no visibility outside the creating process.

If your application does not need to set timeouts when locking resources, to break locks when shutting down threads, or to share a locked resource among processes (the last three bullets), Cognex recommends using critical sections rather than mutexes whenever possible as critical sections are faster.

Semaphore

A semaphore object (**ccSemaphore**) is generally used to restrict access to some fixed number of objects, each of which is intended for use by one thread at a time. A semaphore has a counter equal to the number of objects available. Each time the semaphore is locked, the locking thread gets access to one of the objects and the semaphore's counter decrements. When the counter equals zero, no further objects are available. Each time the semaphore is unlocked (typically because a locking thread has finished using the object), the counter increments.

Event

An event object (**ccEvent**) is used to coordinate the timing of activities in multiple threads. An event has two states, *reset* and *set*. Locking an event causes the locking thread to block until the event's state changes to *set*. Multiple threads can be blocked waiting for an event to be *set*. Depending on the *manualReset* property of the event, changing the state of the event to *set* unblocks either the thread that first locked the event or all of the threads that have locked the event.

Lock

A lock object (**ccLock**) is used with all of the thread-synchronization objects. **ccLock** is a stack object whose constructor claims a lock and whose destructor releases the lock. This makes it impossible to accidentally forget to unlock a synchronization object due to multiple return statements or an unexpected C++ throw.

The following code locks the mutex without using a **ccLock** object.

```
// This sample FAILS to lock the mutex

ccMutex dataLock;
ccSomeObject data;

void myThreadEntry(void* args)
{
  while (running)
  {
    dataLock.lock();   // Lock the mutex.
```

```
    data.run();        // Use the shared data.
    dataLock.unlock(); // Unlocks mutex; however, if data.run()
                       // throws, mutex is not unlocked.
  // etc.
  }
}
```

If the `data.run()` call above were to throw, the mutex would be locked forever and all other threads that attempted to lock it would hang. Using the **ccLock** object avoids this problem. Note that **ccLock** unlocks its synchronization object in its destructor, so use an extra scope to control the unlocking (or call **ccLock::unlock()** explicitly).

The following code allocates a **ccLock** object on the stack that locks the mutex.

```
// This sample CORRECTLY locks the mutex

ccMutex dataLock;
ccSomeObject data;

void myThreadEntry(void* args)
{
  while (running)
  {
   {
      ccLock temp(dataLock)
      data.run();        // Use the shared data. If data.run()
                         // throws, mutex is unlocked when
                         // "temp" goes out of scope.
   }
   // etc.
  }
}
```

Requirements and Recommendations

The following sections present some requirements and recommendations for using multiple threads in CVL applications.

Exit Threads Correctly

When exiting an application, always stop all the threads that you started. When stopping a thread, always wait until the thread has actually stopped before allowing your application to exit.

Threads must be designed to exit correctly. Threads created to perform a discrete piece of work can exit as soon as the work is done. Other threads execute until some external event takes place, and periodically check a flag to see if they should continue running or exit. To stop these threads, the application sets the flag to indicate that the thread should exit, then waits for the thread to exit.

Even if you design your threads correctly, it still may be necessary to use the static function **cc_Resource::breakLocks()** to break the locks on a thread before exiting the application. For example, your thread could call a CVL function that blocks on an event that never gets set. Before terminating the thread, use **cc_Resource::breakLocks()** to break that lock (and any others that may exist). When writing a thread in which you call **cc_Resource::breakLocks()**, wrap the body of your thread in a try-block, catch the *cc_Resource::BrokenLock* exception that is thrown whenever **cc_Resource::breakLocks()** is used, and clean up any resources you allocated in the try-block in the exception handler before terminating the thread.

For any thread created with **cfCreateThreadCVL()** or **cfCreateThreadMFC()**, use the **cfWaitForThreadTermination()** function to wait for the thread to exit. Otherwise, wait on the thread handle using a Win32 function such as **WaitForSingleObject()**.

Avoid Deadlocks

A common problem in multithreaded applications is deadlocking. This occurs when two threads lock one object each, and then each tries to lock the object that the other has already locked. Both threads will be stalled.

To avoid this, adopt a convention defining the order in which objects are locked. As long as all threads attempt to lock the objects in the same sequence, then a deadlock cannot occur because the second thread will block before it can lock even one object.

Use cfCreateThreadCVL() to Create Threads

Using **cfCreateThreadCVL()** or **cfCreateThreadMFC()** to create threads that call CVL will guarantee that the per-thread resources allocated internally by CVL are freed when the thread exits.

Use Global Variables in a Thread-Safe Way

Mutexes address the situation where multiple threads must share global data, and access the data one thread at a time. When each thread needs its own copy of global data, you can use thread-local storage,

An object that is created using thread local storage can be accessed in the same way as any static or global data. The difference is that each thread sees a different copy of the object. Since each copy will be accessed from only one thread, no other protection mechanism is needed. The way to construct an object in thread local storage is to use **ccThreadLocal**. In all cases, static variables that can be accessed from more than one thread should either be declared with **ccThreadLocal** or protected with a **ccMutex**.

Waiting for Multiple Synchronization Objects

A thread can block on multiple mutexes, semaphores, and events (all of which derive from **cc_Resource**) To do this, get native handles to the thread-synchronization objects (using **cc_Resource::rawResourceHandle()**), put the handles in an array, and use the Win32 function **WaitForMultipleObjects()** to block on any one or on all of the synchronization objects.

Note Blocking on multiple events is only supported in threads executing on the host PC.

Multi-Core and Multi-Processor Systems

Many contemporary PCs are equipped with multi-core processors and/or multiple processors. These systems are intended to offer higher performance than a single-core/single-processor system by dividing the system's work across the multiple processor cores.

Note Throughout this document the term *multi-core* is used to refer to a computer system with multiple processor cores or processors or both.

Part of the performance improvement is provided automatically by the system's operating system. Multi-core-aware operating systems such as Windows 7, Windows 8.1, and Windows 10 automatically use multiple cores to execute threads concurrently. If your application creates and uses multiple threads intelligently, the Windows thread manager will automatically distribute your applications work as efficiently as possible across all the cores in the current system.

Multi-core Optimized Tools

Starting with release 6.7 of CVL, selected CVL vision and image processing tools are optimized to take advantage of multi-core systems. These tools have been modified to automatically divide their processing work across multiple worker threads. The following tools have been optimized:

- PatMax PMAlign tool
- PatInspect tool
- Affine transform tool
- Image processing tools (3x3 median filter, all variable-size kernel tools apart from NxM Morphology with configurations other than 3x3 kernel configuration)
- Image stitching tool

CVL provides the *Work Manager* interface that lets you manage how these multi-core-optimized tools make use of multiple cores. Using this interface you can configure the maximum number of worker threads that CVL tools will create during execution.

You can specify that tools create

- one worker thread for each core in the current system
- a fixed number of worker threads
- no worker threads (the default)

When deciding how many worker threads to specify, you should consider how your application (and potentially any other applications on your system) are already using threads.

In general, if you have already written a multi-threaded application to take advantage of multiple cores, by allocating a thread for each camera processing thread for example, and the number of threads is equal to the number of cores on your system, then there is little advantage to allowing vision tools to create multiple worker threads. In fact, it may even slow your application.

If, however, your system has more cores than your application has threads, you may be able to improve performance by configuring the Work Manager to create worker threads to run on the potentially unused cores.

Note The Work Manager interface does not let you specify a flexible number of threads, other than "one thread per system core". If you want to write code that will reserve the same number of cores regardless of the number present in the system, you can use the **ccSystemInformation** class described in *ch_cog/thrdperf.h* to determine the number of cores on the current system.

Multiprocess Applications

Independent of multithreading, you might also decide to split your vision processing application into more than one process. A process consists of an executable plus any number of DLLs, OCXs, or other binaries.

Each Cognex frame grabber board can be used for image acquisition, I/O, or vision processing by only one process at a time. Once a process uses a board for any of these purposes, it has claimed that board, and the board will not be usable by another process until the owning process has exited. Attempts to access a board while another process has ownership of it will result in a *HardwareInUse* exception being thrown.

CVL classes that implement Cognex vision tools may access Cognex hardware only to check for the presence of security bits for license verification. Hardware accesses for this purpose are exempt from the above restriction.

All CVL classes that do not use Cognex hardware, or use the hardware only for license verification, can be used from multiple processes without claiming the hardware.

Exception Handling 12

This chapter describes the CVL exception handling mechanism. CVL exception handling is based on the standard C++ exception handling. CVL defines a hierarchy of exception classes. You can handle CVL-generated exceptions globally, you can catch exceptions on a per-tool basis, or you can catch individual exceptions.

CVL Exception Handling Overview provides an overview of the CVL exception handling mechanism and exception class hierarchy.

Handling CVL Exceptions describes how to handle exceptions generated by CVL.

Deriving Your Own Exceptions tells you how to add your own exceptions to the CVL exception hierarchy.

CVL Exception Handling Overview

CVL objects handle errors using the standard C++ mechanism: they throw *exceptions*. All CVL exceptions are derived from the CVL base class **ccException**. Most individual CVL tools and subsystems derive a single tool-wide abstract base exception class from **ccException**, as well as individual exceptions for specific problems.

The CVL Exception Class Hierarchy

All CVL exceptions derive from the class **ccException**. CVL defines a number of global exceptions derived directly from **ccException**. Individual tools typically define an abstract base class called **Errors** from **ccException**. The **Errors** class is scoped to either a namespace class associated with the tool, or to one of the tool's classes.

Tools then define one or more specific exceptions from the tool-wide **Errors** abstract base class. These individual exceptions are also scoped to the tool's namespace or tool class.

Figure 119 illustrates the CVL exception hierarchy.

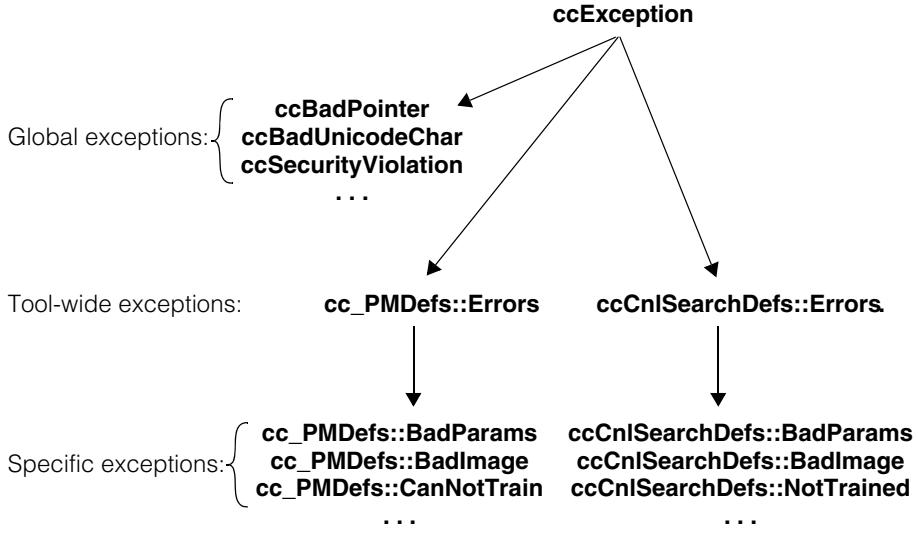


Figure 119. CVL exception hierarchy

Handling CVL Exceptions

The first step in handling CVL exceptions is to decide which exceptions to handle. The CVL exception hierarchy makes it possible to handle all CVL exceptions using a single **try**-**catch** block; to handle all exceptions generated by a single tool or subsystem using a single **try**-**catch** block; and to handle individual exceptions using individual **try**-**catch** blocks. This section describes each of these techniques.

Catching Exceptions Globally

Because all CVL exceptions are derived from **ccException**, you can create a single **try**-**catch** block to catch **ccException** that will catch all CVL thrown exceptions.

Note Standard exceptions such as **bad_alloc** which are thrown during the execution of a CVL function are thrown directly out of CVL.

The following example shows how to catch all CVL exceptions that occurred in your program.

```
int main(int argc, char *argv[]
{
  try
    {
    // Main program body
    ...
    }
  catch(const ccException& ex)
    {
    // ex.errorNumber() returns the 32-bit error number
    // ex.message() returns a string describing the error
    ...
    }
}
```

Catching Exceptions By Tool

Because all of the exceptions thrown by a single CVL tool or subsystem are derived from a single tool-wide abstract base exceptions class (named **Errors**), you can create a single **try**-**catch** block that will catch all errors thrown by a specific tool.

The following example shows how to capture all exceptions thrown within a block of code by PatMax or CNLSearch.

```
void locateModel(void)
{
  // Catch all CNLSearch and PatMax errors thrown during the
  // model-location phase
  //
  try
    {
    // CNLSearch and PatMax calls
    ...
    }
  catch(const ccCnlSearchDefs::Errors& ex)
    {
     cogOut << "CNLSearch exception thrown!" << cmStd endl;
    }
  catch(const cc_PMDefs::Errors& ex)
    {
     cogOut << "PatMax exception thrown!" << cmStd endl;
    }
}
```

The file *ch_cvl\cnlsrch.h* defines an abstract exceptions class called **Errors** within the **ccCnlSearchDefs** namespace class and the file *ch_cvl\pmpbase.h* defines an abstract exceptions class called **Errors** within the **cc_PMDefs** namespace class. Since all of the individual CNLSearch exceptions are derived from **ccCnlSearchDefs::Errors** and all of the individual PatMax exceptions are derived from **cc_PMDefs::Errors**, you can catch all CNLSearch exceptions by catching **ccCnlSearchDefs::Errors** and all PatMax exceptions by catching **cc_PMDefs::Errors**.

Catching Individual Exceptions

To catch one or more specific exceptions thrown by a CVL tool or subsystem, simply create **catch** blocks for each exception you want to catch. The following example shows how to catch training image-related exceptions thrown by CNLSearch's training function:

```
ccCnlSearchModel* trainModel(
  const ccCnlSearchTrainParams& tParams,
  const ccPelBuffer_const<c_UInt8>& tImage)
{
  ccCnlSearchModel* model = new ccCnlSearchModel;

  // Catch image-related training errors, but pass
  // any other exceptions along to the caller.
  try
  {
```

```
    model->train(tImage, tParams);
  }
  catch (const ccCnlSearchDefs::BadImageSize& ex)
  {
    delete model, model = 0;
    cogOut << "Bad image size!" << cmStd endl;
    return (ccCnlSearchModel*)0;
  }
  catch (const ccCnlSearchDefs::BadImageContent& ex)
  {
    delete model, model = 0;
    cogOut << "Bad image content!" << cmStd endl;
    return (ccCnlSearchModel*)0;
    }
  catch (ccException& ex)
  {
    delete model, model = 0;
    cogOut << ex.message() << cmStd endl;
    return (ccCnlSearchModel*)0;
  }
  catch (...)
  {
    // Handle any non-ccException exceptions.
    // Unless you call non-CVL code inside the
    // try block, this is unlikely to happen.
    //
    delete model, model = 0;
    return (ccCnlSearchModel*)0;
  }
  return model;
}
```

Handling Exceptions

Once you have written the code to catch an exception, you then need to handle the caught exception. CVL exceptions provide information about themselves in three ways:

- Each specific exception is associated with a particular error or condition, as documented in the header files in *ch_cvl* and in the documentation for each public member function in the *CVL Class Reference*.
- Each exception has a unique 32-bit numerical value associated with it.
- Each exception has a human-readable message string associated with it.

If you are catching groups of exceptions (by catching **ccException** or one of the scoped **Errors** classes), then you will need to use the second and third methods to obtain information about the exception

Exception Numbers

Each Cognex-defined exception has a unique 32-bit numeric value associated with it. A range of exception numbers are defined for each Cognex tool or subsystem in the file *ch_err/errbase.h*. Individual files in *ch_err* define the error codes for each individual tool's exceptions.

If you create your own exceptions, use numbers in the range 0x00000001 through 0x0FFFFFFF only. Exception numbers from 0x10000000 through 0xFFFFFFFF are reserved by Cognex. You can use 0 to indicate "no error".

Exception Strings

The **ccException** base class from which all CVL exceptions are derived has a member function (**message()**) which returns a human-readable string providing additional information about an exception. The message usually contains the name of the exception. Some CVL exception strings contain addition information about what caused the exception.

Deriving Your Own Exceptions

There is no requirement that you use the CVL exception handling mechanism for exceptions that you write as part of your application development. However, using the CVL exception handling mechanism (which means, in part, deriving your exceptions from **ccException**) offers the following advantages:

- You can use the same exception handling techniques for your exceptions and CVL exceptions.
- Any exceptions that you throw within RPC functions are automatically re-thrown on the host system. If you do not use the CVL exception handling mechanism, then exceptions you throw within RPC functions are thrown on the host as **ccRPC::RuntimeError**.

This section describes how to create and throw exceptions using the CVL exception handling mechanism.

Creating Exceptions

The general steps to creating a CVL exception are as follows:

1. Declare the exception, deriving it from **ccException**.
2. Define an override for the **errorNumber()** function that returns a unique exception number for your exception. Keep in mind that your exception numbers should be unique and must be within the range 0x00000000 through 0x0FFFFFFF.
3. Optionally define an override for the **message_()** function that returns a human-readable string describing the exception.

In addition to deriving individual exceptions directly from **ccException**, you can also derive an abstract class from **ccException**, then derive individual exceptions that all relate to a single component or subsystem or your application from the abstract class. (This is the mechanism used by the CVL exception hierarchy.) The steps to perform this are as follows:

1. Declare the abstract exception class, deriving it from **ccException** and scoping it to a namespace class. Usually, the name of this class should be **Errors**. Make the class's constructor protected (to prevent direct instantiation of the class).
2. Declare the individual exceptions, deriving each of them from the **Errors** class defined in the first step.
3. Define an override for each exception's **errorNumber()** function that returns a unique error number.
4. Optionally define an override for each exception's **message_()** function that returns a human-readable string describing the exception.

Note All exceptions that you derive from **ccException** must provide a default constructor. You can make this constructor private to prevent instantiation of an abstract exception base class.

Using the Exception Macros

The header file *ch_cvl\except.h* defines a number of macros that make it easy to create exceptions derived directly from **ccException** as well as abstract exception base classes. The macros are divided into *declaration* macros and *definition* macros. You use the declaration macros in your header file when you declare the exceptions. You use the definition macros in your source file when you create the exception definitions.

Each group of macros is described in this section.

Declaration Macros

You use the declaration macros to produce the class body when you declare an exception. For example, to declare an exception called **fileExcept** derived from **ccException**, use the **cmExceptionDcl** macro as follows:

```
class fileExcept: public ccException
{ cmExceptionDcl; };
```

Table 63 summarizes the declaration macros defined in *ch_cvl\except.h*.

Declaration	Use
`cmExceptionDcl`	Declares an exception.
`cmExceptionDclAbstract`	Declares an abstract exception base class. You can use this type of base class to derive a group of exceptions that are all related to a particular subsystem or component of your application.
`cmExceptionDclMsg`	Declares an exception that overrides its base class's **message_()** function.
`cmExceptionDclAbstractMsg`	Declares an abstract exception base class that overrides its base class's **message_()** function. You can use this type of base class to provide a default message for a group of exceptions while enabling individual exceptions to return different strings.

Table 63. Declaration macros

Definition Macros

You use the definition macros to provide implementations for the functions in the exception classes that you declare with the declaration macros.

Table 64 summarizes the declaration macros defined in *ch_cvl\except.h*.

Macro	Usage
`cmExceptionDef(D,B,N)`	Defines a exception *D* derived from an abstract base exception class *B* and having a unique exception number *N*.
`cmExceptionDefMsg(D,B,N,M)`	Defines a exception *D* derived from an abstract base exception base class *B* and having a unique exception number *N* with an implementation of **message_()** that returns the supplied message string *M*.
`cmExceptionDefAbstract(D,B)`	Defines an abstract exception base class *D* derived from an exception base class *B*.
`cmExceptionDefAbstractMsg(D,B,M)`	Defines an abstract base exception class *D* derived from an exception base class *B* with an implementation of **message_()** that returns the supplied message string *M*.

Table 64. Definition macros

Exceptions with Data

If the exceptions that you derive from a single abstract base exception class have data members, then you must use the macros that support serialization and you must implement the serialization functions for the exceptions. The macros that support serialization are **cmExceptionDefAbstractSpecialSerialize** (use instead of **cmExceptionDefAbstract**), **cmExceptionDefSpecialSerialize** (use instead of **cmExceptionDef**), **cmExceptionDefAbstractMsgSpecialSerialize**(use instead of **cmExceptionDefAbstractMsg**), and **cmExceptionDefDefMsgSpecialSerialize**(use instead of **cmExceptionDefDefMsg**).

Example Declarations and Definitions

The following examples show how to:

- Declare an abstract exceptions class for file I/O exceptions.
- Declare several concrete file I/O exceptions derived from it the abstract exceptions class

The following code shows how the exception declarations would appear in the header file:

```
// Declare an abstract class (with a default
// message) for file errors.
//
class fileExceptions: public ccException
{ cmExceptionDclAbstractMsg;
protected:// Prevent instantiation
  fileExceptions(){}
};

// Declare individual file I/O exceptions
// (with messages)
//
class ReadOnly: public fileExceptions
{ cmExceptionDclMsg; };

class NotFound: public fileExceptions
{ cmExceptionDclMsg; };

class BadName: public fileExceptions
{ cmExceptionDclMsg; };
```

The following code shows the definitions that you might create to implement the exceptions classes declared above. This code would appear in a source file.

```
// Define the abstract exception class for file errors.
// If an exception derived from fileExceptions does not
// implement message_(), then the default message is provided.
//
cmExceptionDefAbstractMsg(fileExceptions, ccException,
  "Unspecified File Error")

// Define the individual exceptions derived
// from fileExceptions
//
```

```
cmExceptionDefMsg(ReadOnly, fileExceptions, 0x00001001,
  "File is read-only")
cmExceptionDefMsg(NotFound, fileExceptions, 0x00001002,
  "File not found")
cmExceptionDefMsg(BadName, fileExceptions, 0x00001003,
  "Invalid file name")
```

Exceptions With Multiple Messages

The exceptions described in the preceding section limit you to a single message for each exception. The CVL exception handling system offers a special abstract class, **ccExceptionWithString**, from which you can derive exceptions that support multiple messages.

To create this type of exception, you derive your exception class from **ccExceptionWithString** instead of from **ccException**. When you throw an exception derived from **ccExceptionWithString**, you supply the message as an argument to the exception's constructor. If you do not supply a string to the constructor, then the base class's implementation of **message_()** is called.

Note that your derived class's constructor must be declared to take a single string argument, and must in turn call **ccExceptionWithString**'s constructor. In addition, your derived class must declare a default constructor; you can meet this requirement by providing a defaulted string argument value.

The following sample code shows how to declare a single exception derived from **ccExceptionWithString** to handle file errors. The following code would appear in a header file:

```
// Declare a single concrete exception for file errors.
//
class fileExceptions: public ccExceptionWithString
{ cmExceptionDcl;
public:
  fileExceptions(const ccCvlString& msg = "File Error") :
    ccExceptionWithString(msg);
};
```

The following sample code shows the definitions that you might create to implement the exceptions classes declared above. This code would appear in a source file.

```
// Define the single exception class for file errors.
//
cmExceptionDef(fileExceptions, ccException, 0x00001000)
```

Then supply the specific message when you throw the exception, as shown below:

```
switch(dosErr)
  {
  case (ERROR_WRITE_PROTECT):
       throw fileExceptions("Write protect error");
  case (ERROR_WRITE_FAULT):
       throw fileExceptions("Write fault error");
  case (ERROR_READ_FAULT):
       throw fileExceptions("Read fault error");
  default :
       // This will return the string "File Error"
       throw fileExceptions();
  }
```

Object and Image Persistence

13

This chapter tells you how to save and restore the state of your CVL application and how to store and load images.

The CVL *persistence* mechanism lets you easily save the state of most CVL objects you have instantiated to an archive. You can then restore the state of the object by loading it from the archive. CVL includes an additional mechanism for saving and loading images to and from a file. This chapter describes both of these mechanisms.

Object Persistence describes the object persistence mechanism. You use this mechanism to store and load objects instantiated from CVL classes. This section also tells you how to save and restore objects instantiated from classes that you write using the standard CVL persistence mechanism.

Image Persistence describes the different methods you can use to save and load images. CVL includes support for reading and writing images stored using other Cognex software frameworks as well as standard Windows bitmap images.

Object Persistence

Whenever you create an instance of a CVL class, you are creating an object. As you call the various methods implemented by the class, the state of the object changes. For many applications you will want to be able to save (and later restore) the state of an object.

The CVL persistence mechanism is based on the use of a **ccArchive** object together with overloads of the C++ stream input and output operators (**>>**, **<<**) for input and output and an overload of the **II** operator to perform bidirectional I/O.

Creating an Archive Object

The first step in using the CVL object persistence mechanism is to create a **ccArchive**-derived object. When you create a **ccArchive**-derived object, you must specify the following information:

- Whether the **ccArchive**-derived object is a *file* archive (objects are persisted to or from a file) or a *memory* archive (objects are persisted to or from a block of memory).

  The *ch_cvl\archive.h* file declares two concrete classes derived from **ccArchive**. **ccFileArchive** (for file archives) and **ccMemoryArchive** (for memory archives).

- Whether the **ccArchive**-derived object is for *loading* (you are instantiating objects from an archive) or for *storing* (you are archiving objects to the archive).

  Although it is possible to both read and write to a given archive, Cognex recommends that once you create a given **ccArchive**-derived object for loading or storing, you use it only for that purpose.

Table 65 shows the appropriate declaration for each of the four possible types of **ccArchive** objects.

Type	Action	Usage
File	Store	`cmStd string fName = "c:\\myarchive.arc";` `ccFileArchive arch(fName, ccArchive::eStoring);`
	Load	`cmStd string fName = "c:\\myarchive.arc";` `ccFileArchive arch(fName, ccArchive::eLoading);`
Memory	Store	`ccMemoryArchive arch;`
	Load	`char *bufPtr;` `ccMemoryArchive arch(bufPtr);`

Table 65. Declaring ccArchive-derived objects.

Persisting Objects

To store an instance of a CVL class to a **ccArchive**-derived object, simply use the **<<** operator, as shown in the following code:

```
ccFileArchive farch("c:\\myarchive.arc", ccArchive::eStoring);
ccPelRect rect(1,2,3,4);

farch << rect;
```

To load an archived object from a **ccArchive**-derived object, simply use the **>>** operator, as shown in the following code:

```
ccFileArchive farch("c:\\myarchive.arc", ccArchive::eLoading);
ccPelRect rect;

farch >> rect;
```

Archives With Multiple Objects

You can store or load any number of objects to or from a single archive. The only requirement is that you load the objects in the same order that you save them. The following code shows how to store and load multiple objects:

```
ccCnlSearchResult      myResult;
ccCnlSearchRunParams   myRunParams;
ccCnlSearchTrainParams myTrainParams;

{ // Store the objects
  ccFileArchive farch("c:\\cnlsearch.arc", ccArchive::eStoring);
  farch << myResult << myRunParams << myTrainParams;
}

{ // Load the objects in the order they were stored
  ccFileArchive farch("c:\\cnlsearch.arc", ccArchive::eLoading);
  farch >> myResult >> myRunParams >> myTrainParams:
}
```

Using the || Operator Instead of << or >>

You can use the **||** operator instead of the **<<** or **>>** operator. The **||** operator will either load from the archive or save to the archive depending on whether the archive was created for loading or storing. This behavior of the **||** operator makes it easy for you to use the same code for saving and loading a collection of objects. Using the same code helps prevent mismatches between the storing and loading operations.

The following code shows how you use the **||** operator to implement simple **loadSystem()** and **saveSystem()** functions:

```
saveSystem()
{
  ccFileArchive farch("c:\\myarchive.arc", ccArchive::eStoring);
  _autoArchive(farch);
}

loadSystem()
{
  ccFileArchive farch("c:\\myarchive.arc", ccArchive::eLoading);
  _autoArchive(farch);
}

void _autoArchive(ccFileArchive &farch)
{
  // List all of the objects that are saved/restored
  // in the following line. The objects will be saved
  // or restored depending on the mode of the supplied
  // ccFileArchive.
  //
  farch || myResult || myRunParams || myTrainParams;
}
```

Persisting Objects to a Non-ccFileArchive-Based File

You can use the **ccMemoryArchive** class to persist CVL objects as part of a larger file-based archive that your application creates. You use a **ccMemoryArchive** the same way you use a **ccFileArchive** object, except that instead of archiving to a file, **ccMemoryArchive** archives to a block of memory. You then store the memory in your own file-based archive. Note that you must archive both the size of the memory and the contents of the memory.

The following code shows how to store CVL objects to a non-**ccFileArchive** file:

```
// Create a memory archive. Memory is allocated
// as objects are stored, freed when the archive is
// destroyed.

{
ccMemoryArchive march;

// Persist some CVL objects to the memory archive

march << myResult << myRunParams;

// Obtain the archive memory's location and size

const char* marchData = march.storage();
long marchSize = march.tell();

// ...
// User-written code to store to a file the archive size
// (marchSize) followed by the archive data (marchData).
// ...

// ccMemoryArchive goes out of scope here and is freed
}
```

The following code shows how to load CVL objects from a non-**ccFileArchive** file:

```
{
long size;
// ...
// User-written code to load the archive size from the file.
// ...

char* buf = new char[size];
// ...
// User-written code to load the archive memory from the file.
```

```
// ...

// Construct a ccMemoryArchive using the memory. (Set the
// fourth parameter to true to have the ccMemoryArchive
// free the supplied buffer when it is destroyed.

ccMemoryArchive march(buf, size, true, true);

// Load the objects from the archive

march >> myResult >> myRunParams;

// ccMemoryArchive goes out scope here; memory is freed
}
```

Simple and Complex Persistence

CVL implements object persistence using the **<<**, **>>**, and **II** operators for all CVL classes. Each class in the *CVL Class Reference* is identified as using *simple* or *complex* persistence.

Objects that support simple persistence can be archived using the method described in the preceding section. Objects that support complex persistence offer the following additional capabilities:

1. Complex-persistent objects can be persisted through a pointer to the object; simple-persistent objects cannot.
2. The persistence mechanism will only persist a single instance of a complex-persistent object, even if multiple pointers to the object are persisted (as long as the object has not changed).
3. Derived complex-persistent objects can be persisted through a base class pointer.

The Cognex-supplied CVL classes are complex-persistent wherever it is required. For example, **ccPelBuffer** is simple-persistent while **ccPelRoot** is complex-persistent. This means that when you persist a **ccPelBuffer**, its associated **ccPelRoot** is also persisted correctly.

Persisting STL Vectors, Lists, and Pairs

CVL includes implementations of the persistence operators for the Standard Template Library (STL) **vector**, **list**, and **pair** containers. Declarations for these operators are in the file *ch_cvl\stlhelp.h*.

The only restrictions on persisting STL **vector**, **list**, and **pair** containers is that the contained type or types must themselves be persistent. Simply include the file *ch_cvl\stlhelp.h* and persist the STL objects as you would any other object.

Persisting Strings

The CVL persistence mechanism allows you to archive objects of type **std::string** and **std::wstring**.

Note **ccCvlString** is a typedef of **std::string** for ANSI mode builds and **std::wstring** for Unicode mode builds.

Whenever you store a **std::string** (including a **ccCvlString** string in ANSI build mode) to a CVL archive, the 8-bit string is converted to a Unicode representation before it is written to the archive. This conversion is performed using the current C-runtime locale of your application. If the conversion cannot be performed using the current C-runtime locale, a *ccBadUnicodeChar* error is thrown.

CVL archives may contain two types of archived **std::string** objects.

- In CVL archives created using versions of CVL prior to 7.1, **std::string** objects are archived as 8-bit strings. When **std::string** objects are loaded from these archives, the archived bytes are loaded without any processing.
- In CVL archives created using CVL versions 7.1 and newer, **std::string** objects are archived as Unicode strings (as noted above). When **std::string** objects are loaded from these archives, the archived Unicode string is converted to an 8-bit representation using the current C-runtime locale of your application. If the conversion cannot be performed using the current C-runtime locale, a *ccBadUnicodeChar* error is thrown.

The persistence mechanism allows you to load an archived **std::string** into a **std::wstring** object as well as loading an archived **std::wstring** into a **std::string** object.

As shown in Table 66, depending on the direction of the conversion and the type of archive, this may require an ANSI-to-Unicode conversion. This conversion is performed using your application's current C-runtime locale setting. If the conversion cannot be performed, the system will throw *ccBadUnicodeChar*.

	Object Stored in Archive		
	std::string	**std::wstring**	**std::string (pre-CVL 7.1 Archive)**
Load into a std::string	• Archive contains Unicode • String is converted to 8-bit using C-runtime locale. • ***Throws if conversion can't be performed***.	• Archive contains Unicode • String is converted to 8-bit using C-runtime locale. • ***Throws if conversion can't be performed.***	• Archive contains 8-bit data • String is loaded into object as-is.
Load into a std::wstring	• Archive contains Unicode • String is loaded into object as-is.	• Archive contains Unicode • String is loaded into object as-is.	• Archive contains 8-bit data • String is converted to Unicode using C-runtime locale. • ***Throws if conversion can't be performed.***

Table 66. Archive string conversion

Note The conversion rules shown in Table 66 apply to both ANSI-mode and Unicode-mode applications.

Writing Your Own Persistent Objects

This section describes how to make the classes that you write as part of your CVL applications persistent.

Writing Simple-Persistent Classes

To add simple persistence to a class, simply implement global **operator||**, **operator<<**, and **operator>>** functions for the class. Your **operator||** function should have the following declaration:

```
ccArchive& operator||(ccArchive &ar, yourClass& val)
```

where *yourClass* is the class to which you are adding persistence. Within your class declaration, declare the **operator||** function as a **friend** of your class so that it has access to your class's private data. Note that you must implement a separate set of global functions for each class you are making persistent.

The implementation of your **operator||** function needs to persist the state of the object being persisted to the supplied archive by persisting the built-in types that make up the object.

You code should implement the **operator>>** and **operator<<** functions using your **operator||** function.

The following code shows how to write a simple-persistent class:

```
class ccExample
{
public:
  ccExample(int x=0, int y=0) : x_(x), y_(y) { }

  int x() const { return x_; }
  int y() const { return y_; }

  // Declare persistence operators as friends of mine
  //
  friend ccArchive& operator|| (ccArchive&, ccExample&);
  friend ccArchive& operator<< (ccArchive& ar,
                                const ccExample& r);
  friend ccArchive& operator>> (ccArchive& ar, ccExample& r);

private:
  int x_, y_;
};

// Implement the global persistence operator functions
```

```
//
ccArchive& operator||(ccArchive& ar, ccExample& ex)
{
  return ar || ex.x_ || ex.y_;
}
ccArchive& operator<< (ccArchive& ar, const ccExample& r)
{
  return ar || (ccExample&) r; // Cast away const-ness
}
ccArchive& operator>> (ccArchive& ar, ccExample& r)
{
  return ar || r;
}
```

For more information on writing simple-persistent objects, refer to the files *ch_cvl\archive.h* and *sample\archive.cpp*.

Writing Complex-Persistent Classes

To add complex persistence to a class, you must follow these steps:

1. Your class must be derived using the **virtual** keyword from **ccPersistent**. (In a complex derivation hierarchy there must be only a single instance of the **ccPersistent** base class.)
2. Your class declaration must begin with the **cmPersistentDcl** macro. (Use the **cmPersistentDclTemplate** macro for template classes and the **cmPersistentDclAbstract** macro for abstract base classes.)
3. Your class implementation must begin with the **cmPersistentDef** macro. (Use the **cmPersistentDefTemplate** macro for template classes and the **cmPersistentDefAbstract** macro for abstract base classes.)
4. Your class must override the **ccPersistent::serialize_()** function. This function takes a **ccArchive** as its first argument. Your implementation of this function must persist the state of the object to the supplied **ccArchive**.
5. Whenever your class's state changes, the class must call the **ccPersistent::mutating()** function.

For more information on writing complex-persistent objects, refer to the *CVL Class Reference* for information on **ccPersistent** and to the files *ch_cvl\persist.h* and *sample\persist.cpp*.

Object Persistence Usage Notes

Starting with CVL 6.2, reading a **ccTriggerProp**, **ccAcqProps**, or **ccAnalogAcqProps** object from an archive causes the **triggerModel()** and **syncModel()** properties to be reset to their default values. If non-default values are used for **triggerModel()** or **syncModel()**, your application should set them to appropriate values after reading from an archive.

When reading an archive, you must restore the exact objects that were written to the archive. For example, if a **ccAnalogAcqProps** was written to the archive, then you must load a **ccAnalogAcqProps**, even though this class is now deprecated.

Image Persistence

Because CVL images are implemented as a persistent CVL class, **ccPelBuffer**, you can simply archive them using the mechanism described in this chapter. When you archive a **ccPelBuffer**, the archived version contains the following information pertaining to the image:

- The pixels that made up the **ccPelRoot** (all of the pixels, not just those visible through the **ccPelBuffer** that you persisted)
- The client coordinate system transformation associated with the **ccPelBuffer**
- The image offset and origin associated with the **ccPelBuffer**

When you reload the image from the archive, all of this information is correctly restored.

Cognex Image Database (CDB)

In addition to the standard persistence mechanism, CVL includes an additional method for storing and loading images: the Cognex Image Database (CDB).

When compared with the standard archiving mechanism, the CDB offers the following advantages:

- CDB files are compatible with other Cognex software frameworks including PVE, Checkpoint, and OMI. You can use a CDB file to exchange image data among applications written using any of these Cognex programming frameworks.
- The version of CVL you are using *may* include an image browsing utility that lets you read and edit CDB files.
- The CDB programming interface is somewhat simpler than the archiving mechanism.

The CDB also has the following disadvantages when compared with CVL's object persistence mechanism:

- Images archived to CDB files do not retain any client coordinate information.
- Images archived to CDB files do not retain any image offset or origin information.
- Only 8-bit images can be archived to a CDB file.

Using the CDB

The CVL interface to CDB files is implemented as a pair of CVL classes: **ccCDBFile** (which provides access to an entire CDB file) and **ccCDBRecord** (which provides access to an individual CDB record).

CDB File and Record Structure

While Cognex CDB files can contain many different record types, the CVL interface to the CDB file mechanism only supports two record types: Image records and acquire records. Both types of records contain the same information: image pixel data.

You manipulate a CDB file using a two-step process:

1. You create a **ccCDBFile** object and associate it with a file.
2. You create one or more **ccCDBRecord** objects and read or write the individual **ccCDBRecord** objects to or from the CDB file.

The next two sections show how to do this.

Loading Images from a CDB File

To load an image into a CVL application from a CDB file, you first create a **ccCDBFile** object, then call the **ccCDBFile::open()** function to associate that object with a CDB file.

Next you create a **ccCDBRecord** object and read the records from the CDB file one at a time using the **ccCDBFile::loadRecord()** and **ccCDBFile::nextRecord()** functions. For each record, if the record's type specifies that it contains image data, call the **ccCDBRecord::image()** function to obtain a **ccPelBuffer** representation of the image.

The following code loads the first image from a CDB file.

```
ccCDBFile file;
ccCDBRecord rec;
ccPelBuffer_const<c_UInt8> pb;

file.open("c:\\images.cdb");

if (file.isOpen())// Make sure file opened OK
{
  while (file.currentRecord() != ccCDBFile::END_OF_FILE)
  {
    file.loadRecord(rec);
    if (rec.type() == ccCDBRecord::ACQUIRE_TYPE ||
         rec.type() == ccCDBRecord::IMAGE_TYPE)
    {
      pb = rec.image();
      break;
    }
    file.nextRecord();
  }
  file.close();
}
```

Saving Images to a CDB File

To save an image from a CVL application to a CDB file, you first create a **ccCDBFile** object, then call the **ccCDBFile::open()** function to associate that object with a CDB file.

Next create a **ccCDBRecord** object and set its type to be *ccCDBRecord::IMAGE_TYPE* and its contents to be the image to save. Then call the **ccCDBFile::append()** function to append the record to the end of the CDB file.

The following function appends the supplied **ccPelBuffer** to the end of the CDB file named *c:\images.cdb*:

```
void appendImageToCDB(ccPelBuffer_const<c_UInt8> pb)
{
  ccCDBFile file;

  file.open("c:\\images.cdb", cmStd ios::out);

  if (file.isOpen())// Make sure file opened OK
  {
    // Create a ccCDBRecord specifying its type
    //
    ccCDBRecord rec(ccCDBRecord::IMAGE_TYPE);

    // Set the record data to be our image
    //
    rec.image(pb);

    // Append it to the CDB file
    //
    file.appendRecord(rec);

    file.close();
  }
}
```

Foreign Image Formats

CVL includes a special class that lets you load and save images to a standard Windows Device-Independent Bitmap (DIB file. This section describes how to use the **ccDIB** class to load and save CVL images from and to DIB files.

Converting a DIB to a ccPelBuffer

Instantiate a **ccDIB** object, then call the **ccDIB::init()** function to load the DIB into the object. You can specify that the DIB be loaded from a disk file, an **istream**, or from packed DIB data on the Windows clipboard.

Once the DIB has been loaded into the **ccDIB** object, call the **ccDIB::pelBuffer()** function to obtain a **ccPelBuffer** containing the image.

The following code shows how to load a DIB file into a **ccPelBuffer**:

```
ccDIB myDIB;

myDIB.init("c:\\flower.dib");
ccPelBuffer<c_Uint8> myPB = myDIB.pelBuffer();
```

Converting a ccPelBuffer to a DIB

Instantiate a **ccDIB** object, then call the **ccDIB::init()** function to load the **ccPelBuffer** into the object. You can save the DIB to a file by calling the **ccDIB::write()** function, or you can obtain the DIB as a packed-DIB format block of memory suitable for placing on the clipboard by calling **ccDIB::clipboardDib()**. (Note that you must first call **ccDIB::clipboardDibSize()** to determine how much memory to allocate for the DIB.)

The following code shows how to save a **ccPelBuffer** as a DIB file:

```
ccPelBuffer<c_Uint8> myPB;
ccDIB myDIB;

myDIB.init(myPB);
myDIB.write("c:\\flower.dib");
```

Note The DIB file created by the **ccDIB** object includes some additional private data at the end of the pixel data. This data should have no effect on the DIB itself.

Input and Output 14

This chapter describes CVL's control of the parallel input/output capabilities of Cognex frame grabbers. The phrase *parallel I/O* may be abbreviated throughout this chapter as PIO.

Parallel I/O and CVL provides an overview of parallel I/O devices and how to control them using CVL APIs.

Programming Parallel I/O shows how to use the CVL classes that create objects representing parallel I/O lines.

Hardware-Specific Parallel I/O Capabilities describes the parallel I/O capabilities of each member of the MVS-8000 vision board family.

Parallel I/O and CVL

This section provides an overview of parallel I/O devices and the control capabilities provided by CVL.

Parallel I/O Devices

Parallel I/O devices are generally small electronic controls, switches, or LEDs that perform a single function. These devices are generally binary: they are either on or off. A parallel I/O device can be as simple as a single LED, which is either illuminated or not; or can be as complex as a trigger device that sends out an electrical pulse in response to a transition from light to dark under its field of view.

In manufacturing automation systems, parallel I/O devices can be used for many purposes. Table 67 shows only a few examples of the many possibilities.

Parallel I/O Device Type	Description
LED	Can be illuminated under CVL control to show, for example, the status of a software action, or to signal an operator of a failed inspection.
Counter	CVL can increment one or more counters. For example, one counter could maintain the number of parts passing inspection while a second counter tracks rejected parts.
Reject switch	On a conveyor belt inspection system, CVL can control a switch that pushes off the line any parts that fail inspection.
Trigger	A trigger device can be used to signal a Cognex frame grabber to acquire an image, based on a visible or electronic stimulus of some kind.
Strobe	A Cognex frame grabber can be used to fire a photo strobe at the instant of image acquisition to illuminate the scene under the camera lens.
Control computer	Lines to and from the Cognex board can be wired to a computer that controls the operation of the entire manufacturing assembly, to report status to, and take direction from, the control computer.

Table 67. Examples of parallel I/O devices

Types of Connections

Parallel I/O devices connect to Cognex frame grabbers in one of two ways:

- **TTL** (transistor-transistor logic). On Cognex frame grabbers, TTL lines are single connection points, usually a screw terminal at the end of a parallel I/O cable. Devices connect to TTL lines with one or two wires: the device's signal wire to the line's connection point, and the device's ground wire to any of theframe grabber's connection points marked TTL GND.

  TTL lines are either on (+5 V), off (0 V), or pulsed (a +5 V square wave is sent). On Cognex frame grabbers, some TTL lines are dedicated as input-only or output-only lines. Other lines are bidirectional and can be set to act either as input or output lines.

  On Cognex frame grabbers, dedicated TTL input lines are usually active low, which is designated with an overbar on the signal name. Active low lines are 0 V when on and are pulled up to +5 V when off.

- **Opto-isolated**. On Cognex frame grabbers, opto-isolated lines are found as a positive and negative pair of connection points. Devices connect to opto-isolated lines with the positive or hot wire from the device to the opto pair's positive line, and the ground wire from the device to the opto pair's negative line. See your Cognex board's hardware manual for circuit diagrams that explain both contact closure wiring and voltage source wiring for opto-isolated circuits.

  Optical isolation circuits stop ground current leakage and act as surge protectors. Opto-isolators protect the PIO device and the Cognex board from possible differences in ground-loop or voltage potential that can damage the device or the board. An opto-isolator contains an LED and a photosensitive transistor. The signal coming into the opto-isolator is converted to light by the LED. The light is detected across an electrically neutral isolation gap on the photosensitive transistor, which generates its own signal in response to the LED's light. Thus, the incoming signal is passed to the other side of the opto-isolator circuit without physical connection.

  The opto-isolation circuitry adds a small time delay in both incoming and outgoing directions, on the order of a few hundred microseconds, compared to a TTL line.

Parallel I/O Control Capabilities of CVL

CVL provides the following capabilities for controlling parallel I/O output lines:

- **Enable/disable line**. An output line can be enabled or disabled, but must be enabled before any use. Disabling a line may leave its state floating or undetermined. For a bidirectional TTL line, enabling with the output function sets the line as a TTL output line. On some frame grabbers, setting one bidirectional TTL line as an output line also sets *all* of that frame grabber's bidirectional lines as output lines.
- **Set line state**. You can set the state of an output line high (logic 1) or low (logic 0) by means of the **ccOutputLine::set(*polarity*)** member function. The effect of **set(true)** depends on whether the line is TTL or opto-isolated, as described in *Enabling and Setting a ccOutputLine* on page 492.
- **Get line state**. You can request the status of an output line with the **ccOutputLine::get()** member function. A returned value of true indicates the line's state is high.
- **Toggle line state**. You can toggle the line's state to its opposite state.
- **Pulse line**. You can generate a pulse of specified polarity and width (subject to granularity limitations).

CVL provides the following capabilities for controlling parallel I/O input lines:

- **Enable/disable line**. An input line can be enabled or disabled, but must be enabled before any use. Disabling a line may leave its state floating or undetermined. For a bidirectional TTL line, enabling with the input function sets the line as a TTL input line. On some frame grabbers, setting one bidirectional TTL line as an input line also sets *all* of that frame grabber's bidirectional lines as input lines.
- **Get line state**. You can request the status of an input line with the **ccInputLine::get()** member function. A returned value of true indicates the line's state is high.
- **Enable callback**. You can specify a callback function to be run on detection of a signal on an input line, and you can specify the incoming signal's polarity state that qualifies as a detection.

Hardware Trigger and Strobe Features

This section discusses the hardware trigger and hardware strobe features of some Cognex frame grabbers. When those features are enabled for a line, that parallel I/O line is consumed, and cannot be controlled with the general control APIs discussed in this chapter.

Hardware Triggering

On some Cognex frame grabbers, four input lines have a special reserved function as hardware trigger lines. On these platforms, each trigger line is associated with a camera port. The correspondence between camera ports and trigger input lines is generally one-to-one. That is, camera port 0 is associated with trigger input line 1 (TTL_IN_5), camera port 1 is associated with trigger line 2 (TTL_IN_6), and so on.

On some platforms, depending on the CVM in use, the association between camera ports and trigger input lines may not be what you expect. See the chapter *Cognex Video Modules and Cameras* in your platform's hardware manual for a table of camera port to trigger line correspondences for your board and CVM combination.

You can enable hardware triggering for any acquisition FIFO, which corresponds to a particular camera port. You enable hardware triggering by:

- specifying the auto or semi trigger model, as described in *Trigger Models* on page 101.
- enabling triggers with **ccAcqFifo::triggerEnable()**, as described in *Enabling Triggers* on page 84.

If you enable hardware triggering for an acquisition FIFO, then the hardware trigger input line associated with the FIFO's camera port cannot be controlled as a **ccInputLine** object as described in this chapter.

Hardware triggering is disabled by default when you create your acquisition FIFO objects. Thus, each of the four hardware trigger lines can be used as **ccInputLine** objects by default, until hardware triggering is explicitly enabled.

Each hardware trigger line is considered separately. You can have a camera set up for hardware triggering on camera port 0, thereby consuming input line TTL_IN_5. At the same time, you can have a software triggered camera set up on camera port 1, and you are free to control input line TTL_IN_6 with **ccInputLine** functions.

Hardware Strobing

The Cognex frame grabbers that support hardware trigger input lines also support hardware strobe output lines. On these platforms, each of four opto-isolated output lines is associated with a camera port. For most platforms, camera port 0 is associated with line OPTO_OUT1, camera port 1 is associated with line OPTO_OUT2, and so on.

On some platforms, depending on the CVM in use, the association between camera ports and strobe output lines may not be what you expect. See the chapter *Cognex Video Modules and Cameras* in your platform's hardware manual for a table of camera port to strobe line correspondences for your board and CVM combination.

If hardware strobing is enabled, when a software or hardware trigger is detected for an acquisition FIFO's camera port, then a pulse is sent on the OPTO_OUT strobe line associated with that camera port. You can enable hardware strobe support with the **ccStrobeProp::strobeEnable()** function, as discussed in *Setting FIFO Properties* on page 75.

If you enable hardware strobing for an acquisition FIFO, then the associated hardware strobe output line cannot be controlled as a **ccOutputLine** object as described in this chapter.

Hardware strobing is disabled by default when you create your acquisition FIFO objects. Thus, each of the four opto-isolated strobe output lines can be used as **ccOutputLine** objects by default, until hardware strobing is explicitly enabled.

Each hardware strobe line is considered separately. You can have a camera set up for hardware strobing on camera port 0, thereby consuming output line OPTO_OUT1. At the same time, you can have a manual (software) triggered camera set up on camera port 1, and you are free to control output line OPTO_OUT_2 with **ccOutputLine** functions.

Mapping Software Lines to Hardware Pins

CVL documentation uses standard signal names in both hardware and software documentation. These standard signal names that describe the primary function of each parallel I/O line are shown in Table 68, where *n* is an integer.

Signal Name	Description
TTL_IN_*n*	TTL input line
TTL_OUT_*n*	TTL output line
TTL_BI_*n*	Bidirectional TTL line, can be enabled for use as either input or output line
TTL GND	Common ground line for TTL lines
OPTO_OUT*n*+, OPTO_OUT*n*–	Optically isolated output pair, with positive and negative connections
OPTO_IN*n*+, OPTO_IN*n*–	Optically isolated input pair, with positive and negative connections

Table 68. Standard PIO signal names in Cognex documentation

The standard signal names serve as bridges between Cognex hardware and software documentation, as follows:

- Look in the reference page in the *CVL Class Reference* for the class that describes your frame grabber to associate software commands with signal names.
- Look in the Cognex hardware documentation for your frame grabber to associate signal names with physical pin locations.

Programming Parallel I/O

This section describes how to use the CVL API for controlling parallel I/O lines.

Setting Up to Use Parallel I/O

To control any parallel I/O lines on a Cognex hardware platform, you must set up a **ccParallelIO** object for your frame grabber. Use code like the following example for the MVS-8504 frame grabber:

```
cc8504& fg = cc8504::get(0);
ccParallelIO* pio = dynamic_cast<ccParallelIO*> (&fg);
pio->setIOConfig(ccIO8504());
```

The code for other frame grabbers is the same except for:

- The hardware device-specific class whose **get()** function you invoke in line 1
- The argument to **setIOConfig()** in line 3

The first line of the above example is already part of your image acquisition code (as described in *Getting a Frame Grabber Reference* on page 72) and is repeated here for clarity. The hardware-specific class names to use are shown in Table 69. The **setIOConfig()** arguments available for each frame grabber platform are also shown in this table. For more information on each **setIOConfig()** argument, see *Hardware-Specific Parallel I/O Capabilities* on page 497 and the documentation for class **setIOConfig** in the *CVL Class Reference*.

Cognex Hardware Platform	Class Name	setIOConfig Argument
MVS-8501 MVS-8511	**cc8501**	**ccIO8501()** or **ccIOExternal8501()** **ccIOSplit8501()** as appropriate for your PIO setup
MVS-8504 MVS-8514	**cc8504**	**ccIO8504()** or **ccIOExternal8504()** **ccIOSplit8504()** as appropriate for your PIO setup

Table 69. Classes describing Cognex vision hardware

Using Output Line Features

Once you have cast your frame grabber object (*fg*) as a **ccParallelIO** object, and have a *pio* object to work with, then you can invoke any of that object's functions inherited from **ccParallelIO** and **ccOutputLine**.

Getting a ccOutputLine Object

To set up a **ccOutputLine** object, use the **outputLine()** function, as shown in the following example:

```
ccOutputLine oline = pio->outputLine(line);
```

where *line* is an integer representing:

- The dedicated TTL output line you wish to use (TTL_OUT_*n*)
- The bidirectional TTL line you wish to use as an output line (TTL_BI_*n*)
- The opto-isolated output line you wish to use, if that line is not already in use as a hardware strobe line (OPTO_OUT*n*)

You can find the values of *line* for each platform in the class reference page for the class that describes your hardware (**cc8501** or **cc8504**). In those references, the *line* values are associated with standard signal names. Refer to your board's hardware manual to associate these signal names with a particular pin number for physical connection of your PIO device.

For example, on the MVS-8100D (which is not supported any longer), the fourth line of code in the following example sets up a **ccOutputLine** object to control the third bidirectional TTL line, TTL_BI_3:

```
cc8100d& fg = cc8100d::get(0);
ccParallelIO* pio = dynamic_cast<ccParallelIO*> (&fg);
pio->setIOConfig(ccIOSingleChannel8100d());
ccOutputLine bi3_out = pio->outputLine(2);
bi3_out.enable(true);
```

Enabling and Setting a ccOutputLine

You must enable your output line before using any other function on it:

```
oline.enable(true);
```

To set the output line high, use the set() function:

```
oline.set(true); // sets the line high
oline.set(false); // sets the line low
```

Invoking **set(true)** has different effects on TTL and opto-isolated lines:

- For TTL output and bidirectional lines, **set(true)** sets the TTL high state (+5 V)
- For opto-isolated lines, **set(true)** enables the no-current-flowing state (the LED inside the opto-isolator is OFF, and thus the circuit is open and not transmitting a signal through the opto-isolator)

The meanings of **set(false)** are as follows:

- For TTL output and bidirectional lines, **set(false)** sets the TTL low state (0 V)
- For opto-isolated lines, **set(false)** enables the current-flowing state (the LED inside the opto-isolator is ON, thus the circuit is closed and is transmitting a signal through the opto-isolator)

Using ccOutputLine Functions

The following examples show more **ccOutputLine** functions in action. See the **ccOutputLine** entry in the *CVL Class Reference* for more information on each function.

There are several status-checking functions.

```
c_Int32 linecheck = oline.lineNumber();
```

This function returns the logical line number of this object. Use this function, for example, to check the last valid value of *line* if a loop in which you set **pio.outputLine(*line*)** is interrupted.

```
bool checking = oline.get();
```

get() returns the last true or false setting specified by **set()**. **get()** does not actually read the hardware to determine the line's state.

```
bool state = oline.enabled();
```

enabled() returns true if the line is enabled, false otherwise.

```
bool usable = oline.canEnable();
```

canEnable() returns true if the line can be enabled. Use this function on bidirectional lines on the frame grabbers where setting the direction of one bidirectional line sets all bidirectional lines. Use **canEnable()** before running **ccOutputLine::enable()** to test whether another part of your application might have left all bidirectional lines configured as input lines.

Finally, there are two functions that change the TTL state or opto-isolated state of the line.

```
oline.toggle();
```

The **toggle()** function runs **set(true)** if the line's state is currently false, and runs **set(false)** if the line's state is currently true.

```
oline.set(false);
oline.pulse(true, 0.300);
```

These lines set this output line to its low state, then raise the line to high for 300 milliseconds. The first **pulse()** argument specifies the direction of the transition; true specifies a zero-to-one transition, thus low to high.

Using Input Line Features

To set up and use a line as an input line, you must start with a *pio* object, as described in *Setting Up to Use Parallel I/O* on page 491.

Getting a ccInputLine Object

To set up a **ccInputLine** object, use the **inputLine()** function, as shown in the following example:

```
ccInputLine iline = pio->inputLine(line);
```

where *line* is an integer representing:

- The dedicated TTL input line you wish to use (TTL_IN_*n*), if that line is not already in use as a hardware trigger line
- The bidirectional TTL line you wish to use as an input line (TTL_BI_*n*)
- The opto-isolated input line you wish to use (OPTO_IN*n*)

You can find the values of *line* for each platform in the class reference page for the class that describes your hardware (**cc8501** or **cc8504**). In those references, the *line* values are associated with standard signal names. Refer to your board's hardware manual to associate these signal names with a particular pin number for physical connection of your PIO device.

For example, on the MVS-8120 (which is not supported any longer) with the standard configuration parallel I/O board, the following lines set up and enable a **ccInputLine** object to control the second dedicated TTL input line, TTL_IN_2:

```
cc8120& fg = cc8120::get(0);
ccParallelIO* pio = dynamic_cast<ccParallelIO*> (&fg);
pio->setIOConfig(ccIOStandardOption());
ccInputLine ttlin2 = pio->inputLine(1);
ttlin2.enable(true);
```

Enabling and Querying a ccInputLine

You must enable your input line before using any other function on it:

```
iline.enable(true);
```

You can query the state of an input line:

```
bool linestate = iline.get();
```

get() returns the true or false state of the line, using the same definitions of true and false as for output lines, as discussed in *Enabling and Setting a ccOutputLine* on page 492.

On Cognex frame grabbers, dedicated TTL input lines are usually active low, which means they will read as 0 V when on and +5 V when off. Bidirectional TTL lines used as input lines will read as +5 V when on and 0 V when off.

Using ccInputLine Functions

The following examples show the **ccInputLine** functions in action. See the **ccInputLine** entry in the *CVL Class Reference* for more information on each function.

You can set up a function to be called when a specified input line state is detected. This function must accept either one or two arguments, and must fit within the constraints described in the *CVL Class Reference* entries for **ccCallback** and **ccCallback1** or **ccCallback2**. The *Class Reference* entry for **ccInputLine::enableCallback** includes an extended source code example of using enabling and triggering **ccInputLine** callback functions.

There are several status-checking functions.

```
c_Int32 linecheck = iline.lineNumber();
```

This function returns the logical line number of this object. Use this function, for example, to check the last valid value of *line* if a loop in which you set **pio.inputLine(*line*)** is interrupted.

```
bool state = iline.enabled();
```

enabled() returns true if the line is enabled, false otherwise.

```
bool enablable = iline.canEnable();
```

canEnable() returns true if the line can be enabled. Use this function on bidirectional lines on the frame grabbers where setting the direction of one bidirectional line sets all bidirectional lines. Use **canEnable()** before running **ccInputLine::enable()** to test whether another part of your application might have left all bidirectional lines configured as output lines.

Notes on Bidirectional Lines

On the MVS-8120, all bidirectional TTL lines are enabled together as a block of input lines or a block of output lines. The first bidirectional line you enable as input or output sets all other bidirectional lines in the same direction.

By contrast, on the MVS-8100D and MVS-8500, each bidirectional TTL line is independent. You can enable one line as output and the adjacent line for input, if your application so requires.

Notes on Opto-isolated Lines

Consult your platform's hardware manual for circuit diagrams of two methods for connecting parallel I/O devices to opto-isolated lines. Those methods are contact closure wiring and voltage source wiring.

Hardware-Specific Parallel I/O Capabilities

This section describes the parallel I/O capabilities of the frame grabbers in the Cognex MVS-8000 series.

MVS-8500 Frame Grabbers

The term MVS-8500 applies to all members of the 8500 frame grabber series, including the MVS-8501 and MVS-8504.

The MVS-8500 provides sixteen independent bidirectional parallel I/O lines. Four of these lines can be enabled as dedicated hardware trigger lines, and four lines can be enabled as dedicated hardware strobe lines. Each line's direction can be set independently of the other lines. For example, you can set one line as a **ccOutputLine** object and then set the adjacent line as a **ccInputLine** object.

The MVS-8500's TTL lines can be converted to opto-isolated lines by using the appropriate cable connecting to the Cognex opto-isolated I/O module. In this case, each line's direction is determined by the cable and I/O module.

The MVS-8500 does not support dynamic light control.

The parallel I/O capabilities of the MVS-8500 depend on the cable and connection option you use, as shown in Table 70.

MVS-8500, cable, connection option	Programmable parallel I/O capability	Dedicated alternate use (if any)
All TTL option: cable 300-0390 and TTL connection box, 800-5818-1	8 bidirectional TTL lines	
	4 bidirectional TTL lines	4 of the 16 lines can be enabled in software as hardware trigger lines
	4 bidirectional TTL lines	4 of the 16 lines can be enabled in softare as hardware strobe lines

Table 70. Parallel I/O capabilities for MVS-8500

MVS-8500, cable, connection option	Programmable parallel I/O capability	Dedicated alternate use (if any)
All opto option: cable 300-0389 and external opto-isolation module, 800-5712-2	4 opto-isolated input lines	
	4 opto-isolated input lines	4 of the 8 input lines can be enabled in software as hardware trigger lines
	4 opto-isolated output lines	
	4 opto-isolated output lines	4 of the 8 output lines can be enabled in software as hardware strobe lines
Half TTL, half opto option: cable 300-0399 to (1) terminal strip connector and (2) opto-isolation module, 800-5712-2	4 bidirectional TTL lines	
	4 bidirectional TTL lines	4 of the 8 input lines can be enabled in software as hardware trigger lines
	4 opto-isolated output lines	
	4 opto-isolated output lines	4 of the 8 output lines can be enabled in software as hardware strobe lines

Table 70. Parallel I/O capabilities for MVS-8500

Version and Security Information

15

This chapter contains information you need to know to retrieve information on which version of CVL software you are running and which vision tools are licensed on the system.

Retrieving CVL Version Information describes how to use the version API to query the CVL version.

Retrieving Security Information describes how to use the security API to determine the licenses enabled on a system. This section also describes how to use the security API to obtain information on a time-limited dongle, if one is installed, such as whether or not it is active, and if so, how much time is left until it is set to expire.

Retrieving CVL Version Information

CVL version information includes the name of the product, the version, any service release, customer release, and/or patch release numbers, and possibly a build number, in the following format:

```
CVL x.y.z [SR n] [CR n] [PR n] (BUILD n)
```

For example,

```
CVL 6.0.0
CVL 5.5.2 SR1
CVL 5.4 CR16 PR4
```

The CVL version information API allows you to retrieve CVL version information programmatically in your vision application, both at compile time and at run time. This information can be useful in several scenarios:

- To display the CVL version in an About box.
- To log information in a file (or dialog box) that can be sent to Cognex to report bugs.
- To confirm that an application is run only against a particular CVL version.
- To write code that uses the new CVL interfaces but continues to compile against older CVL versions.

CVL Version API

The CVL version API allows you to do the following.

- At compile time: retrieve the CVL version being compiled against as either a text string for display or an object for comparison or logging of information.
- At compile time: use #ifdef's to select different code sections to build based on the CVL version being compiled against.
- At run time: retrieve the CVL version being run against as either a text string or an object.

The CVL version API is implemented in the following files, shipped with each CVL release.

- *<ch_cvl/verdefs.h>*: contains version symbol definitions.
- *<ch_cvl/version.h>*: specifies the **ccVersion** class and its public version query interface.
- *cogversn.dll*, *cogver.dll*, and *cogver.lib* files: implementation of the public version query interface.

Version Macros

The following macros (defined in *<ch_cvl/verdefs.h>)* specify the complete CVL version.

- **cmCvlVersion**: specifies the CVL version. For example, for CVL 6.0.0 this symbol is defined as follows:

  ```
  // CVL 6.0.0 Release
  #define cmCvlVersion     0x06000000
  ```

  The **cmCvlVersion** macro defines a hexadecimal number in the format *0xAABBCCDD* where:
  AA is the major version.
  BB is the minor version.
  CC is the point release version.
  DD is the release type (will always be 00 for released software).

- **cmCvlVersionDetails**: specifies the service release, customer release, patch release, and build. For example, for CVL 5.4 CR7 PR1 (Build 3) this symbol would be defined as follows:

  ```
  // CVL 5.4 CR7 PR1 B03
  #define cmCvlVersionDetails     0x00070103
  ```

  The **cmCvlVersionDetails** macro defines a hexadecimal number in the format *0xAABBCCDD* where:
  AA is the service release number (00 if not a service release).
  BB is the customer release number (00 if not a customer release).
  CC is the patch release number (00 if not a patch release).
  DD is the build number.

Retrieving Compile-Time and Run-Time Versions

CVL also provides two global functions (defined in *<ch_cvl/version.h>*) for retrieving the compile time and run time versions. Both of these functions return **ccVersion** objects.

To retrieve the CVL version used at compile time, use:

cfGetCompileTimeCvlVersion()

To retrieve the CVL version used at run time, use:

cfGetRunTimeCvlVersion()

If the run-time version is not available (for example, because you are linking against static libraries), the compile-time version is returned in both cases.

Using the ccVersion Class

The **ccVersion** class (defined in *<ch_cvl/verdefs.h>*) contains the public version query interface. Use its methods to perform the following tasks.

To retrieve version information as a formatted text string (**ccCvlString**), use:

ccVersion::getAsText()

To retrieve the product name (for example, “CVL”), use:

ccVersion::product()

To retrieve the CVL version as the hexadecimal number defined by **cmCvlVersion**, use:

ccVersion::version()

To retrieve details on the service, customer, patch release, and build as the hexadecimal number defined by **cmCvlVersionDetails**, use:

ccVersion::details()

To retrieve the major version number (for example, the 6 in 6.0.0), use:

ccVersion::major()

To retrieve the minor version number (for example, the 3 in 5.3.1), use:

ccVersion::minor()

To retrieve the point version number (for example, the 0 in 5.4.0), use:

ccVersion::point()

To retrieve the release type (always 0 for released software), use:

ccVersion::type()

To retrieve the service release number, use:

ccVersion::sr()

To retrieve the customer release number, use:

ccVersion::cr()

To retrieve the patch release number, use:

ccVersion::pr()

To retrieve the build number, use:

ccVersion::build()

Some CVL versions can be compared and others cannot. For example, you cannot compare the versions of two different products. To determine whether you can compare two CVL versions, use:

ccVersion::canCompare(const ccVersion& otherVer)

For versions that can be compared (where **ccVersion::canCompare()** returns true), use the standard equality (==), greater-than (>), less-than (<), greater-than-or-equals (>=), and less-than-or-equals (<=) operators to compare the versions.

Version Query Sample Code

One of the samples provided with CVL demonstrates how to query version information. See *Using the Single File Code Samples* on page 43 for information on how to build samples.

The *%VISION_ROOT%\sample\cvl\version.cpp* source file contains the following sample code.

```
/* CVL Sample code for ccVersion */
#include <ch_cvl/version.h>
#include <ch_cvl/constrea.h>

int cfSampleMain(int argc, TCHAR** const argv)
{
  ccVersion compileVersion = cfGetCompileTimeCvlVersion();
  ccVersion runVersion = cfGetRunTimeCvlVersion();
  cogOut << cmT("CVL Version Information") << cmStd endl;
  cogOut << cmT("Compile time: ") << compileVersion.getAsText()
    << cmStd endl;
  cogOut << cmT("Run time:     ") << runVersion.getAsText()
    << cmStd endl;

  return 0;
}
```

This sample code retrieves the compile-time CVL version using the global function **cfGetCompileTimeCvlVersion()** and then retrieves the run-time version using the global function **cfGetRunTimeCvlVersion()**. The code then outputs the two versions as formatted text strings using the **ccVersion::getAsText()** member function.

Retrieving Security Information

The security API allows you to retrieve information on which vision tools are licensed to run on the system. You can use the security API to report licenses enabled on all boards in the system, on a specific board, or on a security key dongle.

CVL Security API

The **ccSecurityInfo** class encapsulates security information. If your CVL application is running in a multi-board configuration, you must decide before you create the **ccSecurityInfo** object whether you want it to report security information for all boards or only a single board.

The default constructor for **ccSecurityInfo** creates a security information object that reports information on all licenses enabled on all boards in the system. If a security key (dongle) is installed rather than a frame grabber, you can also use this simple method of construction to create a security information object to report information on the licenses enabled on the dongle. You would construct such an object, for example, as follows:

```
ccSecurityInfo secinfo;
```

To construct a security information object that reports information on licenses enabled on a specific board, pass a pointer to the board object to the **ccSecurityInfo** constructor. For example, the following code constructs a security information object that can report on licenses enabled specifically on the second MVS-8501 in the system:

```
cc8501& fg = cc8501::get(1);
ccSecurityInfo secinfo(fg);
```

Once the security information object is constructed, you can use it to retrieve a list of the licenses enabled on the specified board (or boards) as follows:

```
cmStd vector<ccCvlString> licenses = secinfo.licenses();
```

The **secinfo.licenses()** call in the above example returns a vector of strings containing the names of all vision tools licensed to run on the board (or boards) used to construct the *secinfo* object. For example:

```
Blob
BoundaryInspector
OCV
PMAlign
PatFlex
```

Retrieving Information About a Time-Limited Dongle

The **ccSecurityInfo** class allows you to determine whether or not security is being provided by a time-limited security key, also called a dongle. If so, you can retrieve further information about the time-limited dongle itself, such as whether it is currently active or expired. If it is currently active, you can query the number of days remaining before it is set to expire.

The following methods of **ccSecurityInfo** retrieve information about a time-limited security key:

- **isTimeLimited()** returns true if security is being provided by a time-limited dongle.
- **is Active()** returns true if a time-limited dongle is installed and active.
- **daysRemaining()** returns the number of days remaining on a time-limited dongle.
- **isExpired()** returns true if the expiration date of a time-limited dongle has been reached.

See the reference page for **ccSecurityInfo** in the *CVL Class Reference* for more information.

Deployment Installation of CVL

16

This chapter addresses Cognex OEMs who want to bundle CVL run-time files with their own application, so that their customers do not need to install CVL separately.

The information in this chapter applies to OEMs developing an application for an MVS-8000 series frame grabber.

The information in this chapter presumes intermediate to advanced knowledge of the following areas:

- Using InstallShield, or another Windows installation program generator. Refer any questions on using these programs to the vendor's technical support staff.
- Editing the Windows registry and registering COM objects. Refer questions on these subjects to a reference book on Windows administration or on COM objects.

Preparing Your Application for Deployment

In order for your application to run on any PC, you must build it using a release configuration. You can make the release configuration active as follows: In Visual C++ .NET, select **Configuration Manager** from the **Build** menu (or select **Release** from the **Solution Configurations** drop down list in the standard toolbar)

The release configuration uses the redistributable versions of the Microsoft DLLs that your application requires. The debugging versions of these DLLs are not redistributable under the terms of Microsoft's license agreement.

If you want to run a debug version of your application on your target PC, you will need to install a licensed copy of Microsoft Visual C++ on it so that the debugging versions of the Win32 DLLs are available to your application.

Build Deployed Applications Only in Release Mode

Cognex strongly recommends building all deployed applications in release mode, not in debug mode. Debug mode applications link against the debug version of the Microsoft run time libraries (*msvcrtd.dll*, and so on). There is a feature in the debug version of the run time libraries that allows you to cause a breakpoint to occur on a specified allocation number (on the 10,034th malloc, for example). The allocation number is stored in a variable of type *long*. The requested breakpoint allocation number defaults to -1 and will always trigger a breakpoint, halting the application, when the allocation number overflows after 4 billion allocations. Depending on the application code, this condition may never occur, or may occur within hours. This is true for both Visual C++ 6.0 and Visual C++ .NET.

Choosing Your Deployment Method

There are two approaches to creating a deployment installation of CVL:

- You can run the CVL installation program silently according to a script that you generate. Your application's installation program would then silently call CVL's installation program. For detailed instructions, see *Silent Installation* in the *CVL Getting Started* manual.
- You can install the deployment-only version of CVL on a test PC, then copy the CVL run-time files to your own application's directory structure. Once integrated with your own application's files, you can prepare a single installation program for your application that includes the run-time components of CVL. This option is discussed in the following section.

Run-Time Only Installation of CVL

Note The technique described in this section can only be used with Cognex frame grabbers. If you are using a dongle (security key), you *must* use the CVL installation program to install CVL on your deployment PCs, as described in the section *Silent Installation* in the *CVL Getting Started* manual.

OEMs may wish to incorporate CVL into their application's installer, without using the silent installation option. For these OEMs, this section lists the minimum run-time files that must be installed in order to invoke CVL as part of their application.

You can install CVL using the **Deployment** installation option to get a feel for the minimum CVL run-time installation.

In the following discussion, the term %VISION_ROOT% is used as shorthand for "the top-level directory in which the CVL files are installed." For example, in a full CVL development installation, VISION_ROOT might be set to *C:\Vision*. However, the use of this term in these instructions does not mean that your installer must define the VISION_ROOT environment variable. This presence of this variable is a requirement to use the CVL sample code, but is *not* required for a CVL run-time installation.

The following lists the minimum installation requirements to run CVL as part of an OEM application. These are the minimum steps that your application's installer must take to duplicate the effect of CVL's **Deployment** installation option.

1. All files in *%VISION_ROOT%\bin\win32\cvl* must be installed in a directory in the target PC's command path.
2. Install and register any Microsoft redistributable DLLs that your CVL-based application requires, as described in *Microsoft Redistributable DLLs* on page 511.

3. For CVL version 6.0 and later, you must copy any CCF files you are using to the directory that contains *cogacq.dll*.
4. Install the device driver for the Cognex frame grabber that your application will use. Follow the steps in one of the subsections of *Manual Driver Installation for Deployment PCs* on page 512, as appropriate for your hardware and target operating system.
5. If you are using the MVS-8600 Camera Link frame grabber, the file *clsercgx.dll* must be placed in the *Windows\System32* directory, or it may be placed in the directory that is pointed to by the registry key *HKEY_LOCAL_MACHINE\software\cameralink\CLSERIALPATH*.

Microsoft Redistributable DLLs

The DLLs you need to install will depend on your application, and thus a precise list cannot be provided by Cognex. The process of determining which DLLs to distribute is described in the Microsoft Developer Network (MSDN) documentation, which is basis of the online help system for Visual C++ .NET. Search for keywords such as “deployment” and “redistribute” in the MSDN Library.

The following points serve only as starting points and reminders. Rely on the latest MSDN documentation for the exact steps to determine the DLLs you need, and to install and register them.

- Use the Dependency Walker utility (*depends.exe*) on your application’s executable to generate a list of DLLs used. If your application dynamically loads some DLLs, use the profiling feature of the Dependency Walker as described in the MSDN documentation. The Dependency Walker utility is installed as part of Visual C++ .NET.
- Use an installation packager such as Installshield that enforces strict version checking of Microsoft redistributable DLLs at installation time. Make sure your installer does not overwrite a Microsoft DLL with an older version.
- Follow the recommendations in the MSDN documentation about where to install the Microsoft redistributable DLLs, whether in your own application’s target directory or in the Windows system directory.
- Follow the recommendations in the MSDN documentation about which DLLs need to be registered with Windows, and about whether to register them using the *regsvr32.exe* utility or with Win32 registration API calls in your installation program.
- If your application will support non-English languages, follow the recommendations in the MSDN documentation about localizing the Microsoft redistributable DLLs.

- For CVL-based applications built with Visual C++ .NET, take advantage of the Merge Module feature of the Microsoft Installer, as supported by Visual C++ .NET. Use an installation packager such as Installshield that supports Merge Modules, and add the Merge Module packages that support ATL, MFC, and the C++ runtime to your application's installation program.

Manual Driver Installation for Deployment PCs

The simplest method to install Cognex drivers on a deployment PC is to use the Cognex Driver Installer, as described in the section *Installing Device Drivers* in the *Getting Started* manual.

In some cases, you may wish to install the specific driver files for a frame grabber or camera manually. That procedure is described in this section. The procedure described here only applies to MVS-8500 and MVS-8600, and not to "GigE Interface and CFG-8700 Driver".

Note The driver files and utilities required for manual driver installation can be found in two places:

- In the directory *\drivers\driverfiles* on the CVL product CD-ROM
- In the drivers directory on a PC where CVL has been installed. The default location for the driver files is *C:\Program Files\Cognex\Common\Drivers*. You can override the default directory during installation of the Cognex Drivers package.

Installing MVS-8500 Drivers

To install the Windows driver for an MVS-8511 or MVS-8514 on a deployment system, follow these steps:

1. Copy the following files from either the *\drivers\driverfiles* directory of the CD-ROM or the drivers directory on your development system to the deployment system:

   mvs8500.cat
   mvs8500.inf
   fg8500.sys
   instw2k.exe

2. You have two ways to install these drivers:

   a. Power the PC down, install the MVS-8510, and boot the PC. When Windows starts up, the Plug and Play Manager detects the new hardware and displays the following message.

   A series of dialogs prompts for the location of the manufacturer's setup files, looking in particular for the device's *.inf* file. Specify the directory containing the files you copied in step 1. The Plug and Play Manager then installs and starts the device driver.

   b. You can use this method either before or after the MVS-8510 is installed in the PC.

   Open a command prompt session on the deployment system, switch to the directory where you copied the files in step 1, and type the following command:

```
instw2k 8500
```

   Note that the files listed in Step 1 must be present in the same directory as *instw2k.exe*.

Installing MVS-8600 Drivers

To install the Windows driver for an MVS-8602e on a deployment system, follow these steps:

1. Copy the following files from either the *\drivers\driverfiles* directory of the CD-ROM or the drivers directory on your development system to the deployment system:

   mvs8600.cat
   mvs8600.inf
   fg8600.sys
   instw2k.exe

2. You have two ways to install these drivers:

   a. Power the PC down, install the MVS-8600, and boot the PC. When Windows starts up, the Plug and Play Manager detects the new hardware and displays the following message.

   A series of dialogs prompts for the location of the manufacturer's setup files, looking in particular for the device's *.inf* file. Specify the directory containing the files you copied in step 1. The Plug and Play Manager then installs and starts the device driver.

   b. You can use this method either before or after the MVS-8600 is installed in the PC.

   Open a command prompt session on the deployment system, switch to the directory where you copied the files in step 1, and type the following command:

```
instw2k 8600
```

   Note that the files listed in Step 1 must be present in the same directory as *instw2k.exe*.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Index

Index