VXL

A collection of C++ libraries for computer vision.

The TargetJr Consortium

Short Contents

Table of Contents

1 Introduction

Chapter summary: VXL is efficient, simple, and fun to use.

This document describes vxl, a collection of C++ libraries designed for computer vision research. Because it's for computer vision it needs to be *efficient*, handling large quantities of numerical data with little overhead. Because it's for research it must be *portable*, so that one can write and run one's program on any machine available, for example in order to access a special camera or fast image processing hardware.

The vxl collection's portability and efficiency are traits which are due in large part to its parentage. The package was built by extracting the core functionalities of two large systems: the "Image Understanding Environment" (IUE) and "Target junior" (TargetJr). Although these environments contain a wealth of useful code, and have facilitated much research, they are widely known for their bulk, and the difficulty of learning to use them. Common complaints have been

- "Too heavy": To use even one small routine, a large portion of the environment must be included in the user's program, increasing program size, compile time, and startup time. Sometimes the increase is such that debuggers and other tools fail to work.
- "Too slow": In some cases, the design of the earlier systems restricted the efficiency of code that could be written when using them. This was in some part due to deficiencies of the C++ compilers in use, but also due to the use of since-superseded programming paradigms.
- "Too weird": Because the IUE is a large system, intended to be used as a programming environment, it imposes stylistic quirks on programmers who use it. The most apparent of these was the use of LaTeX to generate code, which offered many benefits, but was ultimately rejected by users.

Despite these complaints, the quality of the software available in the environments, and the promise of a sustained, concerted software development effort has meant that the IUE/TargetJr has many users worldwide. The job of vxl is to make that many *happy* users, by creating a set of libraries which have the same useful code, but which are light, fast, and not weird.

1.1 Core Libraries

The core of vxl was defined in February 2000, and comprises five libraries.

1.1.1 vcl: The C++ compatibility layer.

The job of vcl is to turn your compiler into a standard C++ compiler. In particular, it ensures that your standard library behaves at least something like that defined in the ANSI/ISO standard. In an ideal world, vcl would not be necessary, as all compiler vendors would ship perfectly compliant systems. But then, in a perfect world, C++ wouldn't be the only choice for high efficiency high-level programming.

1.1.2 vbl: Basic Templates.

The basic templates library contains a small set of generally useful templated containers and algorithms, e.g. smart pointers, triples, and sparse arrays. Many of these things should have been in the STL.

1.1.3 vul: Utilities.

This contains a number of generally useful C++ classes and functions, notably file and directory handling, command-line argument parser, string utilities, a timer, etc.

1.1.4 vnl: Numerical algorithms.

The main classes in vnl are the matrix and vector classes, which are very simple c-like objects. There is no reference counting, objects are returned by value, or passed into routines. The use of objects to hold the output of matrix decompositions also reduces the amount of copying that needs to be done. The numerical algorithms are wrappers around public-domain, high quality Fortran code from the TargetJr `netlib` library.

1.1.5 vil: Imaging.

The imaging library is designed to work well with very large images, but its primary design objective is to be fast and convenient for the typical in-core images that are encountered in video and digital image processing. A panoply of file formats is handled, and it is easy to add support for a new format.

1.1.6 vgl: Geometry.

It is in geometry that the spartan aesthetic of the core of vxl becomes most apparent. The library deals with the geometry of points, curves, and other objects in 2 or more dimensions, but may not call on the numerics library. Therefore, it is restricted to operations which it is reasonable to express in standard C++, which places a useful limit on what may be placed there. For example, surface fitting cannot be in geometry as it would require the singular value decomposition. However, there is much here that is useful, and the library is quite large enough already. The `vgl_algo` library contains the higher-level code, which is allowed to depend on `vnl` (and if necessary even on `vul`).

1.1.7 vsl: Binary I/O

This section describes how to save and restore objects using a binary stream. It details how to add the appropriate functions to each class to make use of this facility.

1.2 Additional Libraries

1.2.1 vcsl: Co-ordinate Systems

This library allows you to represent and manipulate transformations, transformation graphs, and units of measurement.

1.2.2 vidl: Video

Load and save images from video files and sources.

1.3 Documentation

The vxl documentation comes in two main forms: this book, and structured comments in the source code, which are automatically extracted. The book is intended to be a single reference, where all the high-level documentation resides. In TargetJr and the IUE there was always a problem finding documentation, as it was in too many places. In vxl, we are following the practice of some other successful systems such as vtk, and providing one monolithic document with, we hope, a good index.

1.3.1 The VXL Book

The book (in subdirectory `vxl/doc/book`) is a human-written collection of VXL documentation, suitable for printing. It is intended to be readable as an introduction to the various facilities provided by the vxl libraries, and to be browsable as a first source of examples and tips.

Why texinfo?

It is written in texinfo, a limited form of LaTeX which allows us to make various forms of output: printed, HTML, windows help files, emacs info, and plain text files. Each of these forms has its adherents, and each is useful. There are other documentation systems which allow multiple output formats, but none match texinfo in the three main formats: paper, hypertext, and ASCII. This is because it's difficult to target such disparate formats, and texinfo has evolved over many years to do it well.

On the other hand, texinfo is maybe hard to learn for those who don't know TEX or LATEX, so we are keeping an eye on formats such as *DocBook*.

1.3.2 Automatically generated documentation

Although the human-written vxl book is the first source, the definitive authority on what a class or function does must be the source code comments. The book will give a high-level description of the most important and commonly-used features of a class, but the source code contains the details. In order to make these details easier to scan, the vxl source files contain specially structured comments, which can be scanned by the *Doxygen* program and converted into an attractive hyperlinked reference.

An example of what the syntax for documentation looks like can be found in `vxl/doc/vxl_doc_rules.[h,cxx]`, but briefly a comment line which begins with `//:` is documentation for the type, function, or variable which follows it. For example

```
//: Brief description of my_class
class my_class {
  //: \brief Brief description, the long one is in the .cxx file
  my_class();
};
```

1.4 The Design and Evolution of vxl

Design goals:

- Cross platform
- Loosely coupled
- light weight — simple interfaces
- high-powered — allow infinitely large images.
- uniform naming – of course.

1.4.1 Motivation

In OpenGL, you guess the name of something, and you're generally right. This is what we would like to achieve with vxl.

Simple and free of programming dogma. For example, many vxl routines exist as C-like functions. For some tasks, C-like functions are more appropriate than a forced object-orientation, for others objects are clearly the more intuitive and compact representation. This is clearly seen in the imaging library, where the `vil_image` object is passed to and returned from C-like functions, for example

```
vil_image i = vil_load(filename);
i = vil_smooth_gaussian(i, 0.6);
vil_save(i, out_filename, "pnm");
```

In this case, object orientation ensures that memory is managed efficiently, without unnecessary copying of data, while using functions makes the code more readable.

When building a large system, it is crucial to maintain consistent conventions for file-system structure, and identifier naming. However, a crucial philosophy in the design of vxl

is that system conventions must be pragmatic. If programmers are ultimately constrained rather than helped by their environment, they will break the rules, and when they do so, they will break them in inconsistent ways. Therefore it is crucial that consistency within the environment is achieved by conventions that are easy to live with. One example, discussed below, is in the rule governing the relationship between header file names and the objects defined within that header. The general principle, then, is "as close to consistent as comfortable, but no more".

1.4.2 Names

Names are always the biggest issue in programming environments. Uniform naming is one of the most important aspects of a library, and part of the design effort in vxl lays in choosing a good, workable scheme for the naming of classes, functions and files. On the other hand, few issues cause as much debate as the choice of naming conventions. However, we are lucky for two reasons: first, because we wish to run cross-platform, many options are restricted; second, as we can never please everybody, we can just choose a convention and pretend we all hate it.

First things first. The name vxl itself refers to a collection of libraries where the x is replaced by a one or two-letter code describing the contents.

Second, do we use `MixedCaseIdentifiers` or `lowercase_with_underscores`? Well, a certain operating system #defines 1300 of the `MixedCaseIdentifiers` in a header file. Therefore, programs which use names like `LoadImage` may or may not link. For this reason, we chose the other ugly solution.

For related reasons, source files in vxl all end in `.cxx`. Template definitions, which look like source files but are essentially header files (or actually #include files, as they are not separately compiled), are suffixed `.txx`.

Each class or function in a vxl library begins with the name of that library, followed by an underscore, for example `vnl_matrix`. This makes it easy to locate the source code for any class, and easy to see the list of libraries on which one's program depends.

This scheme was chosen over namespaces because few compilers support namespaces well, and their implications in large-scale development are as yet poorly understood.

An obvious extension of this convention would be to insist that every class or function (in fact, every globally visible symbol exported by a library), should be defined in its own include file, yielding an easy relationship between symbols and include statements. However, such a scheme imposes great demands on programmers, who must generate long lists of includes, and on library writers who must create many files with very little in each. Pragmatism must rule if we are to avoid programmer discomfort, thus chaos. A rule that is still useful, and easier to adhere to is used:

> The file `vxl/vxl_stuff.h` exports only identifiers which begin with `vxl_stuff_`, and possibly `vxl_stuff` itself.

An example of a header file which obeys this rule is shown in Figure 1.1.

Of course, this rule is less helpful to clients of the library, who might remember that there is a class called `vtl_thing_doer`, but not know which header file to include. However, the disadvantage is not great, as the likely options are just `vtl/vtl_thing.h` and `vtl/vtl_thing_doer.h`. What is ultimately helpful to clients is a uniform naming scheme, kept so by pragmatic constraints on the library developers.

FAQ: What about `operator+(vcl_stuff, vcl_stuff)`? Frequently given half-answer: Koenig lookup means it's OK.

1.4.3 Layering

Which brings us on to layering: core vxl libraries may not depend on each other. The numerics library cannot use the handy string manipulation in vul, the imaging library cannot use matrices, or smart pointers. This restriction is central to the design of vxl, because

```
#ifndef vxl_stuff_h_
#define vxl_stuff_h_

enum vxl_stuff_things {
  vxl_stuff_bare,
  vxl_stuff_spartan
};

class vxl_stuff {
};

bool vxl_stuff_grok(vxl_stuff);
extern int vxl_stuff_val;

#endif
```

Figure 1.1: Example of a header file `vxl_stuff.h` which obeys the naming convention. Only identifiers that begin with `vxl_stuff_` may be defined.

it intrinsically limits the size of the core libraries. It appears to contradict traditional software development practice because it implies that code will be duplicated rather than reused. However, the amount of code duplicated is small, and the benefit of lego libraries is enormous. The current status is encapsulated in this figure

Level:	0	1	2	3
	vcl	vbl	vnl_algo	v**l
		vnl	vil_algo	
		vil	vgl_algo	
		vgl	v*l_io	
		vul		
		vsl		

Library layers: libraries in one layer may not call their siblings, even if this means code copying.

However, other alternatives have been considered, and may yet be considered. For example, what about putting vul at level 0.5, so that common utilities can be available to level 1? The problem of course is that everything will end up in the level 0.5 library. We could impose the rule that nothing goes in vul unless it is used in at least two level 1 libraries. I don't know whether that would help or not...

There have been two significant propositions to accept a level 0.5 library - for smart pointers, and binary IO. In both cases it turned out to be relatively straight forward to provide the required functionality without the 0.5 library, and indeed the proposers have found the alternative implementations superior in many ways.

2 Example programs

Before anything else, let's look at some VXL example programs.

2.0.1 Hello world

Here's one to print "Hello world":

```
// Include standard C++ input/output library
#include <vcl_iostream.h>

// main is the first function to be called
int main()
{
  // send string "hello world" to the standard
  // output stream cout.
  vcl_cout << "Hello world\n";

  // Must return something from main
  return 0;
}
```

This is the same example that one might see in any C++ book, but for the include line at the start. C++ books assume that your C++ compiler works, which is not true of any C++ compiler in the year 2000. Chapter 3 goes into more detail, but for the moment, include standard headers such as `<cstdio>` using `<vcl_cstdio.h>`.

2.0.2 Loading an image

Because most vxl programs use images in some form, here's an example of loading an image, and printing the pixel value at (100,100). It assumes the image is greyscale, and does no error checking.

```
#include <vcl_iostream.h>
#include <vxl_config.h>
#include <vil/vil_rgb.h>
#include <vil/vil_load.h>
#include <vil/vil_image_view.h>

int main()

  // Load image into memory.
  vil_image_view<vil_rgb<vxl_byte> > img = vil_load("foo.ppm");

  // Access pixel (100,100) and print its value as an int.
  vcl_cerr << "Pixel 100,100 = " << img(100,100) << vcl_endl;

  return 0;
```

3 vcl: C++ compatibility

Chapter summary: When you want a `std::string`, use `vcl_string`.

The job of vcl is to fix your compiler. C++ is not just a language; the standard also includes an extensive library of classes and functions, which make ISO C++ a powerful and useful tool for building computer programs. Unfortunately, few C++ compilers available in 2001 have a bug-free implementation of the standard, so we have to supply our own bug fixes.

To give an example of the type of problems that vcl fixes, here are a few interpretations from the standard which have been observed in some well known vendors' libraries. Many are entirely within the letter of the law, but remain prone to introduce confusion.

On one compiler, `<iostream>` and `<iostream.h>` refer to entirely different stream libraries, which may not be linked together. Therefore every file in the program must use the same header. For us, the `<iostream>` version is appropriate, but of course, not all of the unix compilers support its use. The solution is for every vxl program to include `<vcl_iostream.h>`. In this way, we can maintain consistency between the many compilers, and if we ever do need to use another stream library, we can make the switch in one place. Thus rule one is

> Wherever you wish to include standard header `<foo>`, you should include `<vcl_foo.h>` instead.

Some compilers place STL classes such as `vector<>` and `string` in namespace `std::`, some don't. Yet others place them there, but do not implement namespaces properly. Therefore, it is very difficult to write portable code because sometimes one must say `std::vector`, sometimes one must use `vector`. Again, we need a way which works on all systems. We could try to insert `using namespace std;` or `using std::vector` commands throughout the program, but (a) this is not considered good C++, and (b) it doesn't work anyway.

The low-tech solution is simply to prefix each standard identifier with `vcl_`, so that `vcl_vector` works everywhere. And this is what vxl does, when you include `<vcl_vector.h>`. Thus, safe programmers prefix *everything* in the standard library with `vcl_`.

> Wherever you wish to use standard class or function `foo`, you should write `vcl_foo` instead.

This may seem excessive, but one gets used to it very quickly, and it quickly indicates to novice C++ programmers which functions are from the standard library. You might think that the designers of vxl would have been clever enough to avoid the vcl˙ prefix by using fancy compiler flags, and many #defines. However, that way lies madness—trying to confuse a C++ compiler always rebounds on one.

Also, when time comes when all compilers will implement ANSI STL classes in a consistent way, it's very easy to 'perl away' the `vcl_` prefixes, or replace them with `std::`; it's much more difficult, if not impossible, to insert `std::` prefixes when there are no `vcl_` prefixes.

3.1 Example

This program is exemplary. It shows how every identifier in the ISO library has been prefixed by `vcl_`. It may look like extreme overkill, but it works, and can be made to work on all compilers we've seen.

```
#include <vcl_string.h>    // string
#include <vcl_iostream.h>  // cout
#include <vcl_vector.h>    // vector
#include <vcl_algorithm.h> // copy
#include <vcl_iterator.h>  // ostream_iterator

int main()
```

```
{
  vcl_vector<vcl_string> strings;
  strings.push_back("Hello, ");
  strings.push_back("World.");
  vcl_copy(strings.begin(), strings.end(),
           vcl_ostream_iterator<vcl_string>(vcl_cout));
  return 0;
}
```

The alternative is somewhat scary. It begins

```
#if defined(_WIN32) || (defined(__SUNPRO_CC) && (__SUNPRO_CC + 0) >= 0x500)
#include <string>
#else
#include ...
```

This document has little more to say about the contents of VCL—a book on C++ should describe it better than we can. However, it is important to note that nothing more can go in there. If it's not in the standard, it's not in VCL. Remember, VCL is full, nothing else can go in there. It cannot for example be "helpfully" modified, Microsoft-style, to send standard error to a window (but see also `vul_redirector`).

3.2 Macros and standard C++

The C++ ISO standard library headers include the functionality of the C ISO standard library headers. For example, the declarations found in `<stdlib.h>` can be found in `<cstdlib>` but in namespace `std::`. This means that functions like `printf()` should be called using `std::printf()` instead; omitting the `std::` is wrong and won't work if the compiler is truly conforming. The exception to this (see `[C.2.3]` in the standard) is those names from ISO C which are actually macros. The following is an incomplete list:

- assert
- setjmp
- NULL
- errno
- stdin, stdout, stderr
- va˙arg, va˙start, va˙end

For example, the following code is the correct way to use C streams in VXL:

```
#include <vcl_cstdio.h>
#include <vcl_cassert.h>

void f(char const *file_name)
{
  assert(file_name);
  vcl_FILE *fp = vcl_fopen(file_name, "r");
  if (! fp) {
    vcl_fprintf(stderr, "failed to open %s for reading.\n", file_name);
    vcl_abort();
  }
  ... other stuff ...
}
```

Note that it uses `assert`, `stderr` and not `vcl_assert`, `vcl_stderr` even though it uses `vcl_fprintf`, `vcl_abort`. This may seem complicated and hard to remember, but it isn't the fault of VCL. If your compiler were strictly conformant you would still have to use `std::fprintf` and you couldn't use `std::stderr`.

3.3 Which parts of standard C++ may be used in VXL?

Eventually the answer to this will be "all parts" but until compilers catch up with the language standard, the answer is "all but the following":

- run-time type information (RTTI) in core libraries.
- exceptions in core libraries.
- nested classes (not supported very well by MSVC)
- wide characters (not supported by FreeBSD).
- member templates (not supported by SGI CC 7.2.x).
- partial template specializations (not supported by SGI CC 7.2.x).
- non-type parameters in function templates (not supported by SunPro 5.0).
- default template parameters which depend on previous template parameters.

Of course, if you are just using VXL for your own purposes you may use whatever C++ constructs you like, you just can't put them in the core VXL libraries.

The justification for banning certain things in core libraries is to encourage the adoption of the core by reducing the possibility of porting problems. The justification for allowing it for Level 2 and greater libraries is that they are really pretty useful and hard to do without in more complex libraries than those in the core (e.g. RTTI for doing things like strategy patterns, or managing polymorphic class trees).

3.4 Template instantiation

In C++, template instantiation is done by the compiler. In real life, it doesn't work as the standard says. In brief here is how template instantiation is supposed to work:

- The compiler must instantiate every needed template before its first use.
- The definition of a template must be in scope at the point of instantiation unless the template is declared as exported.

To understand the implications of this (and the meaning of "exported") let's consider the following program, composed of two "translation units" (i.e. files):

```
// matrix.h
template <typename T>
struct matrix
{
  matrix(int, int);
  ...
};
```

The class template `matrix<>` just declared is defined in

```
// matrix.txx
#include "matrix.h"
template <typename T>
matrix<T>::matrix(int m, int n) { ... }
```

Finally we refer to the matrix class in a little program:

```
// program.cxx
#include "matrix.h"
int main()
{
  matrix<double> P(3, 4);
  ...
  return 0;
}
```

The program is ill-formed because the `matrix<double>` must be instantiated before its use in `program.cxx`, but the definition isn't in scope at that point. One way to fix this is to explicitly instantiate the required template in some source file and make sure to compile that source file first. Another is to include the definition of the template in the header file. A third solution is to put the keyword `export` in front of the declaration of `matrix<T>`, which makes it possible to implicitly instantiate `matrix<double>` even when the definition is not in scope.

Unfortunately, there are at the time of writing (April 2001) no compilers which understand and implement `export` so we are currently limited to using two kinds of templates:

- "Inline" templates whose definitions are in scope at the point of use. This includes (in most implementations) the containers and algorithms from the STL. An example is `vcl_vector<vcl_pair<int, vcl_string> >`.
- "Export" templates which are explicitly instantiated in the program (despite being exported). In VXL, these instantiations all live in the `Templates/` directories in the source tree and include things like `vnl_svd<T>` which only need to be instantiated for a handful of types anyway.

Now, it gets worse. For various reasons it is sometimes advantageous to turn off automatic instantiation of the first kind of template. This is only really the case for some architectures but if you are unlucky enough to be using one of them, you also have to explicitly instantiate your STL container classes and algorithms in the `Templates/` directories. [You should consider skipping the rest of this section until you actually have a template problem. Don't read it just for pleasure.] To make it easier to do this, and to make sure it works on all platforms, explicit instantiation is done using preprocessor macros. The macro used to instantiate a class or function is obtained by capitalizing the name (of that class or function) and appending `_INSTANTIATE`. For example, here is how to instantiate a `vcl_map<int, X>` where `X` is the name of some class:

```
// Templates/vcl_map+int.X-.cxx
#include "X.h"         // declaration of class X.
#include <vcl_map.txx> // the instantiation macro lives here.
VCL_MAP_INSTANTIATE(int, X, vcl_less<int>);
```

and here is how to instantiate `vcl_vector<X *>`:

```
// Templates/vcl_vector+X~-.cxx
struct X; // forward declare the class.
#include <vcl_vector.txx>
VCL_VECTOR_INSTANTIATE(X *);
```

The naming convention for such files is described **** where ? ****.

If you are using the build system that comes with VXL and you aren't using implicit instantiation you should put such instantiations in the `Templates/` directory or you *will* be stuffed.

3.5 Use of Assertions (Developer Topic)

First of all a definition: Assertions include anything that acts like an assert(). They check for some error condition that should not occur if the code is working correctly[1]. They are there to detect broken code. The fact that they abort rather than do something more graceful is irrelevant because the program is already broken. Typical things to check for include array bounds violations, container size mismatches, invalid function parameters. The following things should not be considered as assertions; invalid user input, file input failure; users are too good at messing these things up, and should be treated sympathetically.

[1] Strictly, computing science defines assertions in association with a code section's pre- and post-conditions.

When putting an assertion in one of the vxl libraries, you should make sure that it can be turned off using `NDEBUG`. This is the intention of the NDEBUG macro, and is very useful for time-critical code. The easiest way to do this is using the `assert()` macro. If you want to print out a more useful error message you could try

```
#include <vcl_iostream.h>
#include <vcl_cstdlib.h>
int f()
{ ...
#ifndef NDEBUG
  if (vcl_sqrt(4.0) < 1.0)
  {
    vcl_cerr << "There is something very wrong with your"
             << "vcl_sqrt() function" << vcl_endl;
    vcl_abort();
  }
  ...
  return 0;
}
```

However you should bear in mind the extra compilation overhead compared to just `#include <vcl_cassert.h>`.

If you want finer control you can add extra control macros. Indeed in the case of time-critical code, you are encouraged to provide this extra control. You can have the default (i.e. when the control macro is undefined) either include, or not include, the assertion. In any case, you *should* ensure that defining `NDEBUG` will override your specialist macros, and turn off *all* assertions. For example,

```
#include <vcl_iostream.h>
#include <vcl_cstdlib.h>
int f()
{ ...
#if (!defined NDEBUG) && (defined I_AM_PARANOID_ABOUT_VCL_SQRT)
  if (vcl_sqrt(4.0) < 1.0)
  {
    vcl_cerr << "There is something very wrong with your"
             << "vcl_sqrt() function" << vcl_endl;
    vcl_abort();
  }
  ...
  return 0;
}
```

Of course, you should also document the effect of your macro in the function Doxygen markup (or class level if appropriate.)

3.6 Notes

3.6.1 Forward Declaration of vcl Classes

Do not forward declare classes in vcl. For example,

```
class vcl_string;   // This is not allowed. std::string is a typedef.
class my_class {
..
```

In this case you should just include <vcl˙string.h>. In the case of stream stuff, there is an include file of forward declarations that will work.

```
#include <vcl_iosfwd.h>
```

General rule: never forward declare vcl˙something with "class vcl˙something;" but either '`#include <vcl_something_fwd.h>`' or '`#include <vcl_something.h>`'

4 vbl: Basic Templates

Chapter summary: The STL isn't perfect. vbl provides some of the missing classes.

All C++ programs need a few basic utilities, and all C++ programmers write their own at some point. The things included in vbl are not considered any better or worse than the several other libraries available, but they are consistently named and lightweight. The key elements of this library are:

`vbl_smart_ptr` class template
: Reference counted smart pointers for any class that defines `ref` and `unref`.

`vbl_triple` class
: Three element version of `vcl_pair`.

`vbl_array_2d` class
: A simple two dimensional array

`vbl_sparse_array_2d` class
: A simple two dimensional sparse array

5 vul: Utilities

Chapter summary: General utilities are often handy. vul has a few.

Like vbl, vul also provides some basic utilities. The difference is philosophical: vbl includes things that could be thought of as extending the STL, while vul provides just plain utilities which don't claim to be useful or general enough to incorporate into the STL, and may not follow the spirit of the STL. The key elements of this library are:

`vul_file` class
`vul_directory` class
: File handling utilities, directory reading

`vul_url` class
: Downloading files over HTTP.

`vul_arg` class template
: Parse command-line arguments conveniently.

`vul_redirector` class
: Simplify redirection of standard output/error

`vul_awk` class
: Read text files, breaking each line into fields.

`vul_reg_exp` class
: Regular expression matching.

5.1 Redirecting standard output: `vul_redirector`

The class `vul_redirector` is provided to simplify the task of filtering the output of `vcl_cerr` and `vcl_cout`, a common requirement in graphical applications. This encapsulates some of the subtleties of deriving from `vcl_streambuf`, providing a simpler interface.

The basic usage is to subclass from `vul_redirector`, implementing the `putchunk` method, which is then called whenever characters are ready for output. The `vul_redirector` constructor takes care of attaching the new buffer to the stream, and of restoring the original behaviour on destruction. Here is a simple example, which switches output on or off depending on the value of a global flag;

```
#include <vul/vul_redirector.h>

bool on = true;

struct my_redirector : vul_redirector {
  my_redirector(vcl_ostream& s): vul_redirector(s) {}
  int putchunk(char const* buf, int n) {
    if (on)
      return vul_redirector::put_passthru(buf, n);
    else
      return n;
  }
};
```

and here is a calling program which exercises the example.

```
int main(int argc, char* argv[])
{
  vcl_cerr << "hi\n";
  {
    my_redirector redir(vcl_cerr);
    on = false;
    vcl_cerr << "magic\n";
  }
  vcl_cerr << "what did I miss?\n";
  return 0;
}
```

When this program is run, the word `magic` is not displayed, because `my_redirector::putchunk` finds that `on == false`. Question, what to you think `put_passthru` does? What happens if you set `on = true` on line 6?

5.2 Complex output formatting: `vul_printf`

While it is possible to achieve all of the functionality of the C `printf` function in C++, it is very very difficult. There are many times when programs can be made clearer by the use of printf formatting, rather than the standard iostream operators. On the other hand, one needs iostreams for type-safe (and convenient) output of user-defined objects. Thus vul provides a stream-aware version, `vul_printf`:

```
vcl_ostream& vul_printf(vcl_ostream&, char const* format, ...);
```

so that one can say, for example,

```
vul_printf(vcl_cerr, "Line %05d, Code %-30s\n", __LINE__, code);
```

5.3 Reading command-line arguments: `vul_arg`

My favourite bit of vul is the `vul_arg` header which provides the easiest way to parse command-line arguments that I've seen. The basic idea is that a minimal specification for a command-line argument includes: the argument's type, a variable to hold it, its flag, and possibly some descriptive text and a default value.

In the default, easy to use (and a bit naughty) form, each argument is declared anywhere in the program, like so:

```
vul_arg<double> my_threshold("-fudge", "Twiddle fudge", 1.7);
//     Type     Variable     Flag      Help text        Default
```

and when `vul_arg_parse` is called, all the arguments are gathered, and extracted from the command line. To use an argument anywhere in the program, use its `()` operator:

```
vcl_cerr << "The threshold = " << my_threshold() << vcl_endl;
```

To check if an argument was changed from its default value, one can check `bool my_threshold.set()`.

Here is a complete example which uses `vul_arg`. I tend to give these argument variables names beginning with `a_`, but don't let that put you off.

```
#include <vcl_iostream.h>
#include <vul/vul_arg.h>

vul_arg<double> a_naughty_global_arg("-hack", "Fudge", 1.2);

void main(int argc, char* argv[])
{
  vul_arg<char const*>     a_filename(0, "Input filename");
  vul_arg<bool>            a_fast("-fast", "Go fast", false);
```

```
  vul_arg_parse(argc, argv);

  vcl_cerr << "Filename [" << a_filename() << "]\n";
}
```

Passing a `0` as the flag string means that the argument is obligatory, and will be taken as the first unparsed word on the command line.

5.3.1 Help text

The help text supplied with each argument is used to provide a summary of options when the special argument `-?` is seen. Running this example with the `-?` flag produces the output.

```
Usage: ./example_vul_arg.exe [-hack float] string [-fast bool]

REQUIRED:
         string       Input filename  ['-']

Optional:
  Switch Type         Help [value]

   -hack float        Quick hack factor  [1.2]
   -fast bool         Go fast  [not set]
```

5.3.2 Lists of numbers

A very useful specialization also exists to read ranges of numbers. Imagine a program called makemovie which operates on a list of frames, specified on the command line:

```
makemovie -frames 1:10,9:-1:1,0,0,0,0
```

These can be easily read into a `vcl_list<int>`:

```
#include <vcl_list.h>
vul_arg<vcl_list<int> >  a_frame_list("-frames", "List of indices");
```

The list will preserve the order specified on the command line, so in the above example, the result of printing the list would be

```
1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1 0 0 0 0
```

As a gratuitous STL example, here is the code that printed that list

```
vcl_copy(a_frame_list().begin(), a_frame_list().end(),
         vcl_ostream_iterator<int>(vcl_cout, " "));
```

5.3.3 More structured argument handling

Of course, it's disgraceful programming practice to throw args around ones program higgledy piggledy, so one can collect arguments in objects of type `vul_arg_list`.

5.4 Timing operations: vul˙timer

If you want to measure elapsed time, for example to find out how long a section of your program takes to run, use the `vul_timer` class. When a `vul_timer` is initialized, like so, it records the current time.

```
vul_timer mytimer;
```

Later, you can find out the elapsed time using the `real()` method:

```
vcl_cerr << "That took " << mytimer.real() << " milliseconds\n";
```

If you want to reset the timer to make a new measurement, use the `mark()` method.

```
mytimer.mark();  // Reset and start counting from zero again.
```

If you are running other jobs on your computer, you might like to know how much time was used by your program alone. For that, one would use the `user()` method.

```
vcl_cerr << "Of which " << mytimer.user() << "ms was actually me\n";
```

In general, it is better to use the `real`, "wall-clock" elapsed time, as the CPU time returned by user can fail to include work carried out on your program's behalf by the operating system. For example, if you're using a lot of memory, the system will swap pages in and out of virtual memory, and your program will run slowly, but `user()` will not report it.

Finally, there's a super-convenient `print` method, which is used to just print the real and user times to a stream without any formatting, for quick testing purposes. Here's an example

```
vul_timer tic; // Start timing
// do stuff
tic.print(vcl_cerr); // Print times to vcl_cerr.
```

5.5 Reading ASCII files: `vul_awk`

It is often convenient to read text files a line at a time, and split each line into space-separated fields. The `vul_awk` class is an easy way to do this. It also adds a few handy extras like stripping comments, and remembering the line number for error messages. It is used like this:

```
vcl_ifstream thefile("myfile.txt");
vul_awk awk(thefile); // initialize and read 1st line
for (; awk; ++awk) {
  vcl_cerr << "Field 0 = " << awk[0] << vcl_endl;
  vcl_cerr << "Field 2 = " << awk[2] << vcl_endl;
}
```

If `myfile.txt` contained the text

```
dapple dawn-drawn falcon,
solihull 1  grimsby 3
```

the above program would print

```
Field 0 = dapple
Field 2 = falcon,
Field 0 = solihull
Field 2 = grimsby
```

5.6 Regular expression parsing: `vul_reg_exp`

A regular expression allows a programmer to specify complex patterns that can be searched for and matched against the character string of a string object. In its simplest form, a regular expression is a sequence of characters used to search for exact character matches. However, many times the exact sequence to be found is not known, or only a match at the beginning or end of a string is desired. The vul regular expression class implements regular expression pattern matching as is found and implemented in many UNIX commands and utilities, notably perl. The perl code

```
$filename =~ m"([a-z]+)\.h";
print $1;
```

is written as follows in vxl

```
vul_reg_exp re("([a-z]+)\\.h");
re.find(filename);
vcl_cerr << re.match(1);
```

The vul syntax is similar to perl's, although not quite as clean. Here are the metacharacters:

```
 ^        Matches at beginning of a line
 $        Matches at end of a line
 .        Matches any single character
[ ]       Matches any character(s) inside the brackets
[^ ]      Matches any character(s) not inside the brackets
 -        Matches any character in range on either side of a dash
 *        Matches preceding pattern zero or more times
 +        Matches preceding pattern one or more times
 ?        Matches preceding pattern zero or once only
()        Saves a matched expression and uses it in a later match.
```

Note that more than one of these metacharacters can be used in a single regular expression in order to create complex search patterns. For example, the pattern `[^ab1-9]` says to match any character sequence that does not begin with the characters "ab" followed by numbers in the series 1-9.

5.7 Loading files using Internet protocols: `vul_url`

`vul_url` provides a useful means of downloading files over the internet. For example, if you used to have code like this

```
bool read_ascii(vcl_istream &);
...
int main() {
...
  vcl_ifstream  input(filename);
  if (!(!input))
    read_ascii(input);
  input.close;
...
```

You can get this code to work, even if `filename` begins with `"http://"` by modifying your code to look like.

```
bool read_ascii(vcl_istream &);
...
int main() {
...
  vcl_istream  *p_input = vul_url::open(filename);
  if (p_input != 0 && !(!(*input)))
    read_ascii(*input);
  delete input;
...
```

This will now work both when filename is an HTTP URL and when it is an ordinary filename.

6 vnl: Numerics

Chapter summary: C++ *can* be like Matlab, but faster and more powerful.

The numerics library, `vnl` is intended to provide an environment for numerical programming which combines the ease of use of packages like Mathematica and Matlab with the speed of C and the elegance of C++. It provides a C++ interface to the high-quality Fortran routines made available in the public domain by numerical analysis researchers.

This release includes classes for

- Matrices and vectors. The library is based on the old TargetJr/IUE classes, which provide the standard operations without excessive overhead.
- Specialized classes for matrices and vectors with particular properties. Class `vnl_diagonal_matrix` provides a fast and convenient diagonal matrix, while fixed-size matrices and vectors allow "fast-as-C" computations (see `vnl_matrix_fixed<T,n,m>` and example subclasses `vnl_double_3x3` and `vnl_double_3`).
- Matrix decompositions. Classes `vnl_svd<T>`, `vnl_symmetric_eigensystem<T>` , `vnl_generalized_eigensystem`.
- Real polynomials. Class `vnl_real_polynomial` stores the coefficients of a real polynomial, and provides methods of evaluation of the polynomial at any x, while class `vnl_rpoly_roots` provides a root finder.
- Optimization. Classes `vnl_levenberg_marquardt`, `vnl_amoeba`, `vnl_lbfgs`, `vnl_conjugate_gradient` allow optimization of user-supplied functions either with *or without* user-supplied derivatives.
- Standardized homes for commonly used functions and constants. Class `vnl_math` defines constants (`pi`, `e`, `eps`...) and simple functions (`sqr`, `abs`, `rnd`...). To quote the header "That's right, `M_PI` is nonstandard!" Class `numeric_limits` is from the ISO standard document, and provides a way to access basic limits of a type. E.g. `numeric_limits<short>::max()` returns the maximum value of a short.

Most routines are implemented as wrappers around the high-quality Fortran routines which have been developed by the numerical analysis community over the last forty years and placed in the public domain. The central repository for these programs is the "netlib" server `http://www.netlib.org/`. The National Institute of Standards and Technology (NIST) provides an excellent search interface to this repository in its Guide to Available Mathematical Software (GAMS) at `http://gams.nist.gov`, both as a decision tree and a text search.

Compliance with the ANSI standard C++ library

The ANSI standard includes classes for 1-dimensional vectors (`valarray<T>`) and complex numbers (`complex<T>`). There is no standard for matrices. The current vnl classes are not implemented in terms of `valarray`, as there is a potential performance hit, but in the future they might be.

6.1 Example: Basic matrix and vector operations

This section provides a brief tutorial in using the main components of vnl. The main components which vnl supplies are the vector and matrix classes. The basic linear algebra operations on matrices and vectors are fully supported. Some very brief examples follow, but for the most part the usage of the `vnl_vector` and `vnl_matrix` classes is (we hope) obvious and intuitive.

Using these is easy, and is often modelled on Matlab. For example, this declares a 3x4 matrix of `double`:

```
#include <vnl/vnl_matrix.h>
int main()
{
  vnl_matrix<double> P(3,4);
  return 0;
}
```

Operators are overloaded as expected, so if we have another 3x4 matrix Q, we can add the two like this

```
vnl_matrix<double> R = P + Q;
```

The `vnl_vector` is equally straightforward. Here we make a 4-element vector of doubles, premultiply it by P, and print the result:

```
vnl_vector<double> X(4);
vcl_cerr << P*X;
```

Several more examples are shown in the figure below.

The vnl matrices are indexed from zero, as in C. This is always a difficult decision for C++ matrix libraries, as mathematical matrices use indices starting from 1—the top left element of A is generally written *a˙11*. However, efficiently achieving this in C or C++ is a little bit tricky, and can confuse some tools like Purify. In the end, it was decided that zero-based indexing was closer to being "not weird".

6.1.1 Efficiency: Fixed-size matrices and vectors.

A C programmer looking at the above examples will immediately grumble about the inefficient memory allocation that is being performed. Let's look into the construction of P in more detail. One can guess that the line

```
vnl_matrix<double> P(3,4);
```

might result in a sequence of actions something like the following:

```
struct vnl_matrix<double> P;
P.rows = 3;
P.columns = 4;
P.data = new double[P.rows * P.columns];
```

The expensive part of this operation is the call to *new*, which might involve many instructions, and even a bit of operating system activity. (Typically a call to *new* or *malloc* will cost about as much as a 2x2 matrix multiply).

If the matrices are small, as in these examples, this cost is significant—if they're bigger than about 20x20 it is not so important. Always remember, when thinking about efficiency, to consider what else is going on in the program. For example, if a matrix is being read from disk, the time taken to read the matrix will be many times greater than a few copies. If you are about to do a matrix multiply (an O(n^3) operation after all), an O(n^2) copy or an O(1) *new* are not going to be hugely significant.

However, for small matrices we should try to avoid calls to *new*, and vnl provides some fixed-size matrices and vectors which do so. The templates which define these are called `vnl_vector_fixed` and `vnl_matrix_fixed`, and the template instances include the size of the vector or matrix in their parameters. A vector of double with fixed length 4 is defined using

```
vnl_vector_fixed<double, 4>
```

with analogous syntax for matrices. Thus a more efficient version of the above sequence would be

```
#include <vnl/vnl_matrix_fixed.h>
#include <vnl/vnl_vector_fixed.h>
int main()
{
```

```
vnl_matrix<double> A(3,3); // 3x3 matrix, elements not initialized
vnl_matrix<double> B(3,3, 1.0); // 3x3 matrix, filled with ones.
vnl_matrix<double> R(3,4); // Rectangular matrix
vcl_cerr << "A is " << A.rows() << 'x' << A.columns() << vcl_endl
         << "A has a total of " << A.size() << " elements" << vcl_endl;
A(0,0) = 2.0; // Set top-left component of A.
A(3,3) = 0.0; // *** Error, (3,3) is outside the range of A.
A.set_size(3,4); // Change size of A, invalidating elements.
R.update(A, 0, 1); // Copy A into R, starting at (0,1): last 3 cols
R.set_column(0, B.get_column(0)); // Copy 1st col of B into R
vcl_cerr << R.extract(3,3, 0,1) // Print last 3 cols
         << R.get_n_columns(1, 3) const; // Ditto

A.fill(0.0); // Set all elements of A to 0.0
A.fill_diagonal(1.0); // Set diagonal elements to 1.0
A.set_identity(); // Set A to identity matrix
R = R.transpose(); // Make transposed copy, assign to R
R.inplace_transpose(); // Transpose R without copying.
A.flipud(); // Reverse order of rows of A
A.fliplr(); // Reverse columns
A.normalize_rows();  // Divide each row by its 2-norm
A.scale_row(0, 2.0); // Multiply row 0 by 2
vcl_memset(A.data_block(), 0); // Access A's raw storage
fill(A.begin(), A.end(), 0.0); // Fill using STL iterators

vnl_matrix<double> C = B + 0.1 * A; // Arithmetic
C += 2.3;
vnl_matrix<double> Csqrt = C.apply(sqrt); // Square root all elements
element_product(Csqrt, Csqrt); // Should be equal to C, modulo roundoff

vcl_cerr << A.fro_norm() // Print sum of squares of elements
         << A.min_value(); // Print minimum element

if (A.is_zero(1e-8))
  vcl_cerr << "Each element of A is within 1e-8 of zero\n";
if (A.is_identity(1e-8)) vcl_cerr << "(A - I) is_zero to 1e-8\n";

A.read_ascii(vcl_cin); // Read A from standard input
```

Figure 4.1: Matrix basics. A sample of the defined matrix operations.

```
  vnl_matrix_fixed<double,3,4> P;
  vnl_vector_fixed<double,4> X;
  vcl_cerr << P*X;
  return 0;
}
```

It's a bit clumsy typing these long names, so it is common to use `typedef` to make shorter ones. Indeed, a few are supplied with vnl, for example `vnl_double_3x4` (defined, of course, in a header called `vnl_double_3x4.h`). So a more compact rendition of our example is

```
#include <vnl/vnl_double_3x4.h>
#include <vnl/vnl_double_4.h>
int main()
```

```
{
  vnl_double_3x4 P;
  vnl_double_4 X;
  vcl_cerr << P*X;
  return 0;
}
```

Note again that in this example there will be no noticeable speedup, because 99% of the runtime will be spent on the last line, printing the vector.

Because some operations such as multiplication have been specially coded for the fixed-size classes, they are also made more efficient by knowing the sizes in advance. For example, this snippet

```
vnl_double_3x3 R;              // Declare a 3x3 matrix
vnl_double_3 x(1.0,2.0,3.0);   // Declare a 3-vector using
                               // local storage
vnl_double_3 rx = R * x;       // Multiply R by x and place
                               // the result in rx
```

is expanded by many compilers into an open-coded sequence of 9 multiplies and 6 adds.

6.1.2 Caveats when using the fixed-size classes

The fixed-size classes are optimally space efficient; `sizeof(vnl_vector_fixed<double,4>)` and `sizeof(double[4])` are the same. To achieve this, it is necessary to decouple `vnl_vector` from `vnl_vector_fixed`, in the sense that neither inherits from the other. This means that you cannot pass a `vnl_vector_fixed` to a function that expects a `vnl_vector` without some conversion. Luckily, there is a cheap conversion operator from `vnl_vector_fixed` to `vnl_vector_ref`, which is a derived class of `vnl_vector`. This conversion operator will be applied behind the scenes in most cases, so you often don't have to worry about it.

```
double norm( vnl_vector<double> const& v );
...
vnl_vector_fixed<double,6> fixed_v;
double n = norm(fixed_v); // this will create a temporary
                          // vnl_vector_ref<double> const
                          // to pass to norm
```

The cost of the conversion is on the order of 1 pointer copy (data pointer) and 1 integer copy (length) for a vector and 1 pointer and 2 integers for a matrix.

Unfortunately, this is not the end of the story. According to the 1998 ISO C++ standard, user defined conversion operators will not be applied when determining candidate template functions. Therefore, the following snippet fails to compile.

```
template<typename T>
T norm( vnl_vector<T> const& v );
...
vnl_vector_fixed<double,6> fixed_v;
// no match for
//    norm(vnl_vector_fixed<double,6>)
// User defined conversion operators are not
// tried since norm is a template.
double n = norm(fixed_v);
```

For these cases, and other cases where the implicit conversion operator cannot be applied, you have to do the conversion explicitly using `as_ref()`.

```
template<typename T>
T norm( vnl_vector<T> const& v );
...
```

```
vnl_vector_fixed<double,6> fixed_v;
double n = norm(fixed_v.as_ref()); // calls norm with
                                   // a vnl_vector_ref<double> const
```

When writing general purpose templated functions that are equally useful for both the dynamically allocated `vnl_vector` and statically allocated `vnl_vector_fixed`, it is often useful to provide a simple forwarding wrapper so that the user is spared the inconvenience of doing the explicit conversion.

```
template<typename T>
T norm( vnl_vector<T> const& v );  // real function
template<typename T, unsigned n>
inline
T norm( vnl_vector_fixed<T,n> const& v ) { // thin wrapper
  return norm( v.as_ref() );
}
...
vnl_vector_fixed<double,6> fixed_v;
double n = norm(fixed_v); // this calls the second norm
```

The final wrinkle with mixing `vnl_vector` and `vnl_vector_fixed` is that the conversion operators, both the implicit and explicit, create temporary `vnl_vector_ref` objects, which, according to the standard, cannot bind to non-const references. Therefore, you cannot pass these to a mutator function that modifies the values in your vector.

```
void mutator( vnl_vector<double>& v );
...
vnl_vector_fixed<double,6> fixed_v;
mutator(fixed_v); // the temporary object created by the
                  // conversion is const => cannot be
                  // passed to mutator.
```

The only solution to this is to explicitly force the temporary object to "give away" its const-ness, using the `non_const()` method in `vnl_vector_ref`.

```
void mutator( vnl_vector<double>& v );
...
vnl_vector_fixed<double,6> fixed_v;
mutator(fixed_v.as_ref().non_const());
```

The discussion above applies equally well to `vnl_matrix` and `vnl_matrix_fixed`.

6.2 Example: Matrix decomposition

The most frequently asked question about `vnl_matrix` is "where is the `inverse` method", and the answer is that the inverse is not defined as a method, because there are too many ways of forming it, each with different tradeoffs. If you really don't care to hear about these things, you can use the `vnl_matrix_inverse` class to compute an inverse object:

```
#include <vnl/algo/vnl_matrix_inverse.h>
int main()
{
  vcl_cerr << vnl_matrix_inverse<double>(A) * B;
  return 0;
}
```

If you want more control over how the inverse is taken, then you might want to look at `vnl_inverse` or at one of the decomposition classes.

TODO - order in general-specific, give flop counts, show decomps.

The following fragment demonstrates use of the `vnl_svd<double>` class to find the approximation of a 3x3 matrix `F` by the nearest matrix of rank 2

```
#include <vnl/vnl_matrix.h>
#include <vnl/vnl_vector.h>
#include <vnl/algo/vnl_svd.h>
#include <vnl/algo/vnl_symmetric_eigensystem.h>
#include <vcl_iostream.h>

int main()
{
  // Read points from stdin
  vnl_matrix<double> pts;
  vcl_cin >> pts;

  // Build design matrix D
  int npts = pts.rows();
  int dim = pts.columns();
  vnl_matrix<double> D(npts, dim+1);
  for (int i = 0; i < npts; ++i) {
    for (int j = 0; j < dim; ++j)
      D(i,j) = pts(i,j);
    D(i,dim) = 1;
  }

  // 1. Compute using vnl_svd<double>
  {
    vnl_svd<double> svd(D);
    vnl_vector<double> a = svd.nullvector();
    vcl_cout << "vnl_svd<double> residual = " << (D * a).magnitude() << vcl_endl;
  }

  // 2. Compute using eigensystem of D'*D
  {
    vnl_symmetric_eigensystem<double>  eig(D.transpose() * D);
    vnl_vector<double> a = eig.get_eigenvector(0);
    vcl_cout << "Eig residual = " << (D * a).magnitude() << vcl_endl;
  }
  return 0;
}
```

Figure 4.2: Example of linear algebra operations. Points are read from stdin into matrix `pts`, and a hyperplane fitted using two different methods.

```
vnl_double_3x3 rank2_approximate(const vnl_double_3x3& F)
{
  // Compute singular value decomposition of F
  vnl_svd<double> svd (F);
  // Set smallest singular value to 0
  svd.W(2,2) = 0;
  // Recompose vnl_svd<double> into UWV^T
  return vnl_double_3x3(svd.recompose());
}
```

A more extensive example of the use of linear algebra is provided in Figure 2, which contains a program to fit a hyperplane to points read from standard input.

6.3 Polynomials

The `vnl_rpoly_roots` class in `vnl/algo` is used to compute the roots (or "zeros") of a real polynomial. For example, given the cubic equation

```
4 x^3 + 3 x^2 - 7 x + 5 = 0
```

we can compute the values of `x` using `vnl_rpoly_roots`. The first step is to collect the coefficients into a vector, listing from the highest power down. In the above example, we should make the vector

```
[4, 3, -7, 5]
```

In C++, this could be written

```
vnl_double_4 poly;
poly[0] = 4;
poly[1] = 3;
poly[2] = -7;
poly[3] = 5;
```

Having prepared the polynomial, we compute the roots:

```
vnl_rpoly_roots roots(poly);
```

Now, `roots` contains the roots, which can be made use of, or simply admired. To facilitate the latter, we shall print them to the console:

```
for (int k = 0; k < 3; ++k) // Cubic polynomial ==> 3 roots
  vcl_cerr << roots[k] << vcl_endl;
```

To get just the real or imaginary parts of the (generally complex) roots, convenience methods `real(int)` and `imag(int)` are provided. So to print only the real roots, one might use

```
for (int k = 0; k < 3; ++k)
  if (roots.imag(k) < 1e-8)
    vcl_cerr << roots.real(k) << vcl_endl;
```

6.3.1 Implementation

The implementation is a wrapper for the fortran code in algorithm 493 from the ACM Transactions on Mathematical Software. This is the Jenkins-Traub algorithm, described by Numerical Recipes under "Other sure-fire techniques" as "practically a standard in black-box polynomial rootfinders". (See M.A. Jenkins, ACM TOMS 1 (1975) pp. 178-189.).

The algorithm fails if poly[0] is zero, so it's often good to try to write your problem so that the leading coefficient (i.e. poly[0]) is equal to 1.

6.4 Nonlinear Optimization

It is not uncommon in computer vision research to meet problems for which there is no known closed-form solution, and a common class of such problems are of the form "find x, y and z, such that the function $f(x, y, z)$ takes its minimum value". For example, fitting a line to a set of 2D points {(x`i,y`i) | i=1..n}. The problem is then to find a,b,c to minimize the sum of distances of each point to the line ($ax + by + c = 0$)

```
                n   (a * x[i] + b * y[i] + c)^2
f(a,b,c) = sum  ---------------------------
               i=1          (a^2 + b^2)
```

In the case of line fitting, a closed-form solution can be found, but in many other problems, no such solution is known, and an iterative method must be employed.

In those cases, one needs a good, general purpose nonlinear optimization routine. Of course, such a panacea does not exist, so `vnl` provides several from which to choose. The

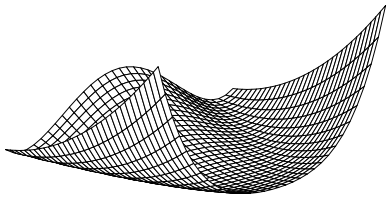

Figure 2: The Rosenbrock "banana" function, used as an optimization test case. Optimization starts on one side of the valley, and must find the minimum around the corner.

factor that decides which is best is most frequently the amount of knowledge that one has about the form of the function. The more you know, the more quickly you can expect the optimization to proceed. For example, if you can compute the function's derivatives, you would expect to achieve better performance.

6.4.1 Choosing a minimizer

The routines provided in vnl may be arranged roughly in decreasing order of generality—and correspondingly, increasing order of speed—as follows:

1. `vnl_amoeba`: Nelder-Meade downhill simplex. The method of choice if you know absolutely nothing about your function, but fear the worst. It you think the function might be noisy (i.e. the error surface has many small pockets), or you don't trust it to have reasonable derivatives, downhill simplex is a good choice. If you want the code to run fast, it's not.
2. `vnl_powell`: Powell's direction-set method. Powell's method, like simplex, doesn't require that you supply the derivatives of f with respect to a, b, and c, but it does assume they are moderately well behaved.
3. `vnl_conjugate_gradient`: Fletcher-Reeves form of the conjugate gradient algorithm.
4. `vnl_lbfgs`: Limited memory Broyden Fletcher Goldfarb Shannon minimisation. Requires 1st derivatives. Considered to be the best general optimisation algorithm for functions which are well behaved (i.e. locally smooth without too many local minima.)
5. `vnl_levenberg_marquardt`: The Levenberg-Marquardt algorithm for least-squares problems. This is usually the best method for any function which can be expressed as `f(x) = (f_1 (x))^2 + (f_2(x))^2 + (f_3(x))^2 + \dots`

As an example of the use of the optimization routines, we'll use a common test case, the "notorious" Rosenbrock function:

```
f(x, y) = (10*(y - x^2))^2 + (1-x)^2
```

The graph of f is plotted in Figure 2.

6.4.2 Function objects, derived from `vnl_cost_function`

Running an optimization is a two step process. The first is to describe the function to the program, and the second is to pass that description to one of the minimizers. Functions are described by *function objects*, or "functors", which are classes which provide a method `f(...)` which takes a vector of parameters as input, and returns the error. Such functors are derived from `vnl_cost_function`:

```
struct my_rosenbrock_functor : public vnl_cost_function {
```

The function is a method in the derived class. Here's the continuation of the declaration of `my_rosenbrock_functor`.

```
  double f(vnl_vector<double> const& params) {
    double x = params[0];
```

```
    double y = params[1];
    return vnl_math_sqr(10*(y-x*x)) + vnl_math_sqr(1-x);
  }
```

Because a `vnl_cost_function` can deal with cost functions of any dimension, not just the 2D example here, `my_rosenbrock_functor` must tell the base class the size of the space it's working in. This is done in the constructor as follows:

```
  my_rosenbrock_functor():
    vnl_cost_function(2) {}
```

And we can now close the declaration of `my_rosenbrock_functor`:

```
}
```

6.4.3 Running the minimization

In order to perform the minimization, a `vnl_amoeba` compute object is constructed, passing the `vnl_cost_function`.

```
my_rosenbrock_functor f;
vnl_amoeba minimizer(f);
```

Having provided an initial estimate of the solution in vector `x`, the minimization is performed:

```
minimizer.minimize(x);
```

after which the vector `x` contains the minimizing parameters.

6.4.4 Least-squares problems: The Levenberg-Marquardt algorithm.

The Levenberg-Marquardt algorithm provides only for nonlinear least squares, rather than general function minimization. This means that the function to be minimized must be the norm of a multivariate function. However, this often the case in vision problems, and allows us to use the powerful Levenberg-Marquardt algorithm. The Rosenbrock function can also be written as a 2D-2D least squares problem as follows:

```
f(x, y) = [ 10(y - x^2) ]
          [    1-x      ]
```

In this case, we need to make a class derived from `vnl_least_squares_function`. TODO

6.5 Design issues (Developer Topic)

This section documents some design decisions with which people might disagree. Please let me know how you feel on these issues. It's also a malleable to-do list. The most important consideration has been to provide simple lightweight interfaces that nevertheless allow for maximum efficiency and flexibility.

6.5.1 Computation in constructors

As noted above, a common model in this package is that the compute objects perform computation within the constructors. While this is slightly distasteful from a traditional C++ viewpoint, it offers a number of advantages in both efficiency and ease of use.

The philosophical argument, say in the case of SVD, is that SVD is a noun. The natural description is "The SVD of a matrix M" which is expressed in C++ as `vnl_svd<double> svd(M)` .

Storage for the results of a computation is provided by the compute object which is convenient, allowing client code to access only those results in which it is interested. Local storage is also more efficient, as objects are constructed at the correct size, and initialized immediately. In contrast, passing empty objects to a function will generally involve a resize operation, while returning a structure will incur a speed penalty due to the necessary copy operations.

Namespace clutter is avoided in the `vnl_matrix` class. While svd() is a perfectly reasonable method for a matrix, there are many other decompositions that might be of interest, and adding them all would make for a very large matrix class, even though many methods might not be of general interest.

The model extends readily to n-ary operations such as generalized eigensystems, which combine two objects to produce others. Such operations cannot be methods on just one matrix.

6.5.2 Fixed-size classes

The classes which provide for fast fixed-size matrices and vectors are essential in a system which wants to make claims for efficiency. In addition, a great many uses of these objects *do* know the size in advance. In this case code using say `vnl_double_3` is more efficient (as well as more self-documenting) than the equivalent referring to a `vnl_vector` of unknown size.

6.5.3 Transposing for Fortran

In calling Fortran code, the first difficulty that becomes apparent is that Fortran arrays are stored column-wise, while traditional 'C' arrays are stored row-wise – a trend that is followed by the `vnl_matrix` class. One solution is simply to store C++ arrays column-wise, and this was an early plan for the IUE.

I have not done anything to alleviate this for two reasons – most routines we call are expensive enough (i.e. $O(n^3)$) that the $O(n^2)$ copy operation is only a small performance hit. Secondly, many decompositions satisfy a transpose-equivalence relationship. For example suppose we wish to use a Fortran matrix multiply which has been hand-optimized for some particular machine. Such a routine may be declared

```
mmul(A, B, C) // Computes C = A B, fortran storage
```

To use this with row-stored arrays, we recall the simple identity

```
C = (C')' = (B' A')' = AB
```

and therefore call `mmul(B, A, C)`, reversing the order of parameters A and B. The fortran code will lay down the result of $B'A'$ into the columns of C, thereby computing $C' = B'A'$ from the point of view of the caller.

This however, doesn't apply to the vnl˙svd<double>, as algorithms generally require only the "economy-size" version where size(U) = size(M) in $USV' = M$. This is $O(mn^2)$ flops rather than $O(m^2n)$ for the full size one. Using the transpose-equivalence would mean a doubling of the computation time, as the "economy-size" decomposition is only implemented for $m > n$. If someone does need the full size decomposition, a flag could be added or a new `vnl_svd` class written.

6.6 Frequently Asked Questions

Question 1

> I would like to do constrained optimisation. Can I add constraints to the existing optimisers in '`vnl/algo`'?

Unfortunately, this approach is not likely to solve your problem, since constrained optimisation is not in general solved by adding constraints to an unconstrained optimiser.

The usual approaches to doing constrained optimisation are:

1. Work the constraints into your cost function, heavily penalising any breach in the constraints.
2. Find a mapping T :[your constrained space]->R^n and use f(T^-1(x)) as your const function instead of f.
3. If problem has linear constrains and a simple cost function (linear or quadratic) then there is code out there for your problem, but unfortunately not yet in VXL.

6.7 Future work

Many of the existing methods are unimplemented, or could benefit from optimization. Users can contribute code to address these deficiencies based on the existing examples, and using the conversion hints in Appendix~A. In addition there are many algorithms that ought to be included, listed roughly in order of priority:

- Additional matrix decompositions in the same vein, including an updateable QR, a basic LU, etc.
- Choice of back-end functions—for optimization one might prefer Powell, or even simulated annealing. For matrix decompositions, particular users might prefer to interface to NAG or IMSL routines. These choices must be allowed to be made easily, thereby encouraging the comparison of algorithms and of alternative implementations.
- Many classes are defined as double-only rather than templated. I will use default template arguments when the compilers support them.

7 vil: Imaging

Chapter summary: Load images using `vil_load`. Access them using a `vil_image_view<T>`.

The vxl image library has evolved from the TargetJr and Manchester Image libraries. As with its predecessors, its primary goals is to provide flexible access to all 2D images, including those too large to fit in the address space of a single program or process, and very powerful and fast access to images in memory. In fact, both cases need similar treatment: even in-core images are assumed to be sufficiently large (say a megabyte) that special care must be taken to avoid unnecessary copying of their data. In both cases, the normal requirements of efficiency and ease-of-use apply. The system must allow:

- Beginners to have easy access to an image type. This image type should also be the default image type for an programmer writing image processing code. This image type must be very efficient to use.
- Fast access to images on disk, at no more than a 10% speed penalty for operations on images in memory.
- Fast loading of subsets of the image data. To look at a small portion of a 10000 by 10000 pixel satellite image, one should not have to load the entire 300 megabytes into memory.
- Efficient memory management, both automatic and programmer-mediated. Automatic management is vital during program development, when the code is changing quickly. On the other hand, release builds need the kind of optimisations that only a human can apply.

This vil library is the second VXL image library, and is sometimes referred to as vil2. The original vxl image library vil1 is deprecated.

You can read more about the design philosophy in `$VXLSRC/core/vil/notes.html`

7.1 Loading and saving

Let's look at an example of vil in use. This program makes an image from a disk file, copies it into memory, and prints the pixel at 100,100.

```
#include <vcl_iostream.h>
#include <vxl_config.h>
#include <vil/vil_rgb.h>
#include <vil/vil_load.h>
#include <vil/vil_image_view.h>

int main()
{
  vil_image_view<vil_rgb<vxl_byte> > img;
  img = vil_load("foo.ppm");
  vcl_cerr << "Pixel 100,100 = " << img(100,100) << vcl_endl;
}
```

The first interesting line declares img to be an image. vil_image_view is the basic image type. It represents an image in memory about whose structure, size and pixel type we know everything. Hence we need to specify the pixel type at this point.

Now let's skip to the end to explain the pixel access method.

```
img(100,100)
```

This looks up the pixel at position 100,100 and returns its value. The pixel type was defined on the first line to be an rgb of bytes, and that is what will be displayed.

```
[255 128 128]
```

Where it matters (such as when loading an image in from disk) it is assumed that the image origin is at the top left of the image.

Finally lets look at the middle line. This consists of two parts. The `vil_load` function does a lot of work behind the scenes to determine what the image type is, and then load that image into memory. The second part is the assignment which has several special properties.

- It does not copy the actual image data. A `vil_image_view` object is really a view of some underlying data. The view understands where the real image data is in memory and how to interpret it. When you copy a view, you merely copy this interpretation information, not the actual image data. This is important, because often images are very big, and copying is expensive. The underlying image is managed with smart pointers so when the last view to the underlying data is destroyed, the image data will be too.
- It can do cheap conversions between different views of the same image. `vil_load` by default loads the image as 3 planes, with the pixel type as `vxl_byte`. It is trivial to reconfigure a `vil_image_view` so that it views the same image data as one plane of rgb pixels. The assignment will automatically do any cheap conversion necessary. You may ask then, how is that we know that the pixel type can be viewed as RGB of bytes? Here, we simply know that our image foo.ppm is this type. In general you can either find out what the pixel type is before you load the image, or you can force it to whatever pixel type you want. The latter may involve a relatively expensive pixel by pixel conversion, so this will not happen automatically.

7.1.1 Loading and saving: The threshold program

Anyway, the usual next step in demonstrating an image handling library is to show thresholding, so let's have a look. This program loads an image into memory, forcing it to RGB byte format, and creates a new image where all pixels greater than a threshold value are set to 0.

```
#include <vxl_config.h>
#include <vil/vil_rgb.h>
#include <vil/vil_load.h>
#include <vil/vil_save.h>
#include <vil/vil_image_view.h>
#include <vil/vil_convert.h>

int main(int argc, char **argv)
{
  vil_image_view<vil_rgb<vxl_byte> > img;
  img = vil_convert_to_component_order(
          vil_convert_to_n_planes(3,
                                  vil_convert_cast(vxl_byte(),
                                                   vil_load(argv[1]))));

  for (unsigned j = 0; j < img.nj(); ++j)
    for (unsigned i = 0; i < img.ni(); ++i)
      if (img(i,j).r < 200 && img(i,j).g < 200 && img(i,j).b < 200)
        img(i,j) = vil_rgb<vxl_byte>(0,0,0);

  vil_save(img, argv[2]);
  return 0;
}
```

The call to `vil_save` sends the modified image in *img* to disk. The choice of file format is determined automatically from the extension of the filename. If one wants more control, a string can be appended to specify the format, e.g.

```
vil_save(buf, argv[2], "jpeg");
```

Of course, if your user has chosen a name such as "foo.ppm", you'll have a oddly named image.

7.2 Copying an image

You should know by now that copying `vil_image_view` objects does not duplicate the data they point to. This allows images to be passed into and out of functions efficiently. It also means that modifying the data in one `vil_image_view` might change that in another. Take this example

```
...
vil_image_view<float> a( vil_convert_cast(float(), vil_load("x")) );
vil_image_view<float> b = a;
b(100,100) = 12;
...
```

After the assignment in line 3, both *a(100,100)* and *b(100,100)* are set to the value 12. On the other hand, if we had used `vil_copy_deep`, thus:

```
...
vil_image_view<float> a( vil_convert_cast(float(), vil_load("x")) );
vil_copy_deep(a, b);
b(100,100) = 12;
...
```

or

```
...
vil_image_view<float> a( vil_load("x") );
vil_image_view<float> b( vil_copy_deep(a) );
b(100,100) = 12;
...
```

then *a* is unchanged after the assignment to *b(100,100)*. Note again that the actual copying is done in `vil_copy_deep`; when the return value is assigned to *b*, there is an efficient view copy.

7.3 Image resources

Broadly there are two sorts of image one is interested in

> images in memory, about which everything is known and all parts of which can be accessed directly.
>
> external images (eg in files) which can only be accessed indirectly, or images about which we may currently be missing information (eg pixel type.)

As we have seen the first sort of images are represented by a `vil_image_view<T>` on the data in memory. For some very large images it is not possible or desirable to load them into memory. In this case it is useful to be able to load in a sub-section of the image, manipulate it, and possible write it out again. Alternatively you may want to pass an image about, and process it without knowing its pixel type. vil supports these second sort of images using `vil_image_resource`. You cannot create an image resource object directly, instead you use a creation function which returns a smart pointer to the base class `vil_image_resource_sptr`. When manipulating `vil_image_resource`s it will almost entirely be in terms of `vil_image_resource_sptr`s. There are several types of image resource, with different creation functions:

- Representing an image in a file: e.g. `vil_pnm_image`, `vil_jpeg_image`. These are created using `vil_load_image_resource()`, and `vil_new_image_resource()`.
- `vil_memory_image`: Representing an image in memory This is created using `vil_new_image_resource()`. Alternatively if you want to wrap an existing view up as a vil image resource you can call `vil_new_image_resource_of_view()`
- Representing a filtered version of an image in a file (without loading in memory): e.g. `vil_crop_image_resource` and `vil_decimate_image_resource`. These are created using the equivalent functions: `vil_crop()`, `vil_decimate()`, etc.
- Representing the outcome of an image processing algorithm (see next section) e.g. `vil_convolve_1d_resource`. These are created using the equivalent functions e.g. `vil_convolve_1d()`.

To actually get some image pixels you call the resource's `get_view()` or `get_copy_view()` method. For example, the `vil_load()` function works by creating a `vil_image_resource`, and then calling `get_view()` for the whole image.

```
vil_image_view_base_sptr vil_load(const char *file)
{
  vil_image_resource_sptr data = vil_load_image_resource(file);
  if (!data) return 0;
  return data -> get_view();
}
```

To set image pixels, you call the resource's `put_view()`.

7.3.1 A rule of thumb.

> When developing an image processing algorithm, first write your algorithm in terms of a function for `vil_image_view<T>`. Then, if you need it, write the vil image resource sptr version, using the `vil_image_view<T>` version to do the actual pixel manipulation.

`vil_image_view<T>` is designed for playing with actual pixel values. `vil_image_resource` derivatives are designed to handle all the other stuff associated with images, e.g. choosing pixel types at runtime, splitting an image into blocks so that it fits in memory, dealing with the arbitrary and complex hassles of image IO.

7.3.2 Using `vil_memory_image` to ignore pixel type.

As explained above, you should be using `vil_image_view<T>` to actually manipulate your pixels. However, in some parts of your code, you may want to pass images around without having to decide the pixel type at compile time. This is a role for a `vil_image_resource` derivative, in particular the `vil_memory_image`. You can wrap an existing `vil_image_view<T>` in a `vil_memory_image` by calling `vil_new_image_resource_of_view()`. Reference counting keeps track of the underlying data in memory, so you can let the original view go out of scope without loss.

It may be tempting to use the `vil_image_view_base_sptr` for this purpose instead. That type is only intended for internal use by vil, and it will almost certainly not behave as you want.

The `vil_image_resource` API has been designed to allow efficient access to `vil_memory_image`. In the example below, if the image resource passed in is really a `vil_memory_image`, the `get_view()` returns a view to the underlying data, so no unneeded data copying happens. Similarly, a call to `put_view()`, can return almost immediately, checking only to confirm that the view is still pointing to the same underlying data.

```
void display_view(vil_image_resource_sptr &ir)
{
  switch (ir->pixel_format())
  {
```

```
    case VIL_PIXEL_FORMAT_BYTE: {
     vil_image_view<vxl_byte> v1 = ir->get_view();
     display_byte(v1); }
    case ...
   }
}
```

7.4 Planes, components and stepping.

`vil_image_view` uses a pointer arithmetic style of indexing. The image data is assumed to be a regularly arranged set of pixels in memory. The view keeps a pointer to the pixel at the origin. It also keeps the pointer difference to get to the next pixel to the right, the next pixel down, and the same pixel in the next plane.

In a general image representation a 2d image consists of multiple planes each containing multiple rasters (rows) each containing multiple pixels, and each pixel contains multiple components. The planes and the components are used for the same purpose, to represent different spectral or functional values (e.g. the red, green and blue channels of an RGB image.) In vil it is usually assumed that an image cannot have both multiple planes and multiple components per pixel. This allows `vil_image_view` to view the same a colour image data as either a 3 plane image or a 1 plane RGB image. You can do this explicitly by calling `vil_view_as_planes()` or `vil_view_as_rgb()`.

However the two representations are not equal. The multi-plane representation is more general than the RGB multi-component one. If the underlying data is actually stored RRRR..GGGG..BBBB.. then it is not possible to view that image as a single plane of RGB pixels. For this reason, a lot of vil prefers to view an image as multi-plane single-component. In particular, the vil˙image˙resource derivatives in vil, will treat all images as multi-plane, scalar component images, whether the underlying data is RGBRGBRGB... or RRRR..GGGG..BBBB.. This means if you have switch statement to deal with pixel types in an normal image resource, you need not worry about any types other than than the following

- bool
- vxl˙byte, vxl˙sbyte
- vxl˙int˙16, vxl˙uint˙16
- vxl˙int˙32, vxl˙uint˙32
- float, double
- vcl˙complex<float>, vcl˙complex<double>

Similarly to the planes to components conversion it is possible to perform a whole range of other manipulations. These include `vil_transpose()`, `vil_flip_ud()`, `vil_decimate()`, `vil_crop()`. One further advantage of the arithmetic indexing scheme is that it becomes easy to create a 2d slice view of a 3d image.

7.5 Algorithms and Image Processing

Several image processing functions can be found in the algo subdirectory of vil. Lets look at an example of finding the image gradient using a Sobel filter.

```
#include <vcl_iostream.h>
#include <vxl_config.h> // for vxl_byte
#include <vil/vil_image_view.h>
#include <vil/vil_print.h>
#include <vil/algo/vil_sobel_3x3.h>

int main()
{
```

```
  unsigned ni=8;
  unsigned nj=15;
  unsigned nplanes=1;
  vil_image_view<vxl_byte> image(ni,nj,nplanes);

  for (unsigned p=0;p<nplanes;++p)
    for (unsigned j=0;j<nj;++j)
      for (unsigned i=0;i<ni;++i)
        image(i,j,p) = vxl_byte(i+10*j+100*p);

  vcl_cout<<"Original image:"<<vcl_endl;
  vil_print_all(vcl_cout,image);

  // Objects to hold gradients
  vil_image_view<float> grad_i,grad_j;

  vil_sobel_3x3(image,grad_i,grad_j);

  vcl_cout<<vcl_endl
          <<"Sobel I Gradient:"<<vcl_endl;
  vil_print_all(vcl_cout,grad_i);

  vcl_cout<<vcl_endl
          <<"Sobel J Gradient:"<<vcl_endl;
  vil_print_all(vcl_cout,grad_j);

  return 0;
}
```

There are also algorithms to perform image arithmetic, smoothing, general 1D and 2D convolution, morphological operations, interpolation, and much more.

7.6 Converting from using the old vil1 to vil.

This section explores the major differences between using the old vil1 and using vil, and some of the implications for converting existing code.

The first and most obvious difference is that whilst there is a broad equivalent to `vil1_image`, and its descendants, this class tree has been split in two. The abstract `vil1_image` is now replaced with a smart pointer to a `vil_image_resource`. The concrete `vil1_memory_image_of<T>` is now a `vil_image_view<T>`. Whereas previously, you might have written code in terms of `vil1_image`, it now usually makes sense to write most image manipulations in terms of `vil_image_view<T>`s. With the old `vil1_image`, you either had to do a `get_section` and operate on raw memory, or do a messy switch statement to cast it to its underlying `vil_memory_image_of<T>` type, or do an expensive `vil1_view_as()` conversion. Now with vil, the `vil_image_view<T>` provides a powerful view directly onto your image in memory.

The `vil_image_view` provides such facilities as compile-time type safety and switchable bounds checking. It also acts as a sort of canonicaliser. A wide range of actual memory layouts can all be treated identically and transparently while working through the `vil_image_view`. Previously, in vil1, the image loader often needed read an unblocked resource and to have several filters placed on top of it to do such things as re-order the raster rows and re-order the component order. vil doesn't do this, but instead uses the vil˙image˙view to provide a canonical view of whatever deranged image format your loader finds most efficient to use.

The second important change is that vil provides full support for planes. In many cases accessing different image planes is directly equivalent to accessing different components. In-

deed, it is often preferable to view an image as a multi-planar rather than multi-component. If your algorithms assume a single plane, it is however trivial to provide a wrapper function which takes a multi-planar image and passes one plane at a time to your algorithm. This can be done with virtually no loss in efficiency, and indeed is how some of the code in `vil/algo` is written.

To help convert existing code there is a script (`core/vil/scripts/vil1tovil.pl`) It converts as much code as it can. However, it can really only deal with file and identifier name changes. There are large structural differences between vil1 and vil, with many of the equivalent functions taking different parameters. The output of the conversion script can best be seen as a hint on which types and classes to use and which functions to call. You will almost certainly need to make extensive further edits to your code to get it to compile again.

If you do not want to convert any code, but would rather use an interface to convert between vil1 and vil types at runtime, then take a look at `<vil1/vil1_vil.h>` which has a function for converting between `vil1_memory_image_of` and `vil_image_view`, and a class that wraps a `vil1_image`, and exports a `vil_image_resource` interface.

7.7 Frequently Asked Questions

Question 1

> I'm trying to load a DICOM image, but it doesn't work. `vil_load.cxx` prints an error message that mentions lots of image type but not `.dcm`. What's wrong?

The DICOM loader in VXL is not built by default, because it is large and only medical image people want it.

You will need to rerun CMake and find Cache value called VXL˙BUILD˙DICOM. Turn it on, and rebuild — it won't need to rebuild everything.

Question 2

> I'm having problems trying to use vil˙image˙view˙base˙sptr to process a loaded image without worrying about what type the pixels are.

The designers of vil recommend against using `vil_image_view_base_sptr` explicitly — it is unlikely to behave the way any user might want or expect. vil never processes pixels independently of their type, and `vil_image_view_base_sptr` is just a smart polymorphic pointer to a concrete `vil_image_view<T>` with some actual pixel type `T`. If you want to convert a loaded image into pixels of a particular type, use one of the `vil_convert` functions

```
vil_image_view<vxl_byte> view =
  vil_convert_stretch_range (vxl_byte(), vil_load(my_filename));
```

If you want to store an image in memory without worrying about its pixel type, `See ⟨undefined⟩ [vil_memory_image], page ⟨undefined⟩.`

Question 3

> What co-ordinate system does vil use?

Mostly vil does not assume that the `i` and `j` co-ordinates have any explicit meaning. Instead, any external meaning to the `i` and `j` directions is provided externally by the user. The choice of the letters `i` and `j` was an explicit decision to discourage any assuption of a Cartesian reference frame.

However there are a few places where further assumptions need to be made. When loading an image, the file format generally provides an explicit mapping to up/down and left/right. In such cases, vil assumes that `image(0,0)` is the top-left-most pixel in the image, that increasing `i` moves right, and that increasing `j` moves down. A similar assumption is used by `vil_rotate` to provide a direction to the rotation angle.

If you need an explicit world co-ordinate frame, within which you can embed an image, then take a look at the vimt library in `vxl/contrib/mul/vimt`. That provides an world-to-image co-ordinates transform, that can be efficiently manipulated to provide transforms up to projective complexity.

7.8 Optimising Image Processing Algorithms (Advanced Topic)

The design of `vil_image_view` (being more flexible than the design of vil1,) and the state of modern optimising compilers (not as good as they could be,) means that naive use of vil images may not be as fast as it should be.

The following example shows the original implementation of the image fill method.

```
template<class T>
void vil_image_view<T>::fill(T value)
{
  for (unsigned p=0;p<nplanes_;++p)
    for (unsigned j=0;j<nj_;++j)
      for (unsigned i=0;i<ni_;++i)
        (*this)(i,j,p)= v;
}
```

This implementation has the advantage of being simple, and easy to test.

In an ideal world the compiler would realise that it doesn't have to recalculate the location of each pixel each step, but instead keep a running pointer to the current pixel location. (Of course, in an ideal world we would be programming using natural language and a microphone.) We can make this optimisation explicit.

```
template<class T>
void vil_image_view<T>::fill(T value)
{
  T* plane = top_left_;
  for (unsigned p=0; p<nplanes_; ++p, plane+=planestep_)
  {
    T* row = plane;
    for (unsigned j=0; j<nj_; ++j, row+=jstep_)
    {
      T* p = row;
      for (unsigned i=0; i<ni_; ++i, p+=istep_) *p = value;
    }
  }
}
```

This can halve the run time on some compilers.

The most important rule in code optimisation is to observe how the code behaves in real life, and concentrate your efforts on where the code spends most of its time. In our example, this means the inner most loop Now, it turns out that in many cases, `istep_==1`, because of the default image layout in memory. Because of this common case it would be worth having the compiler generate machine-code for the inner-most loop in this special case. We can do this by explicitly testing for such a special case.

```
template<class T>
void vil_image_view<T>::fill(T value)
{
  T* plane = top_left_;

  if (istep_==1)
  {
```

```
    for (unsigned p=0;p<nplanes_;++p,plane += planestep_)
    {
      T* row = plane-1;
      for (unsigned j=0;j<nj_;++j,row += jstep_)
      {
        int i = ni_ ;
        while (i>=0) { row[i--]=value; }
      }
    }
    return;
  }

  for (unsigned p=0;p<nplanes_;++p,plane += planestep_)
  {
    T* row = plane;
    for (unsigned j=0;j<nj_;++j,row += jstep_)
    {
      T* p = row;
      for (unsigned i=0;i<ni_;++i,p+=istep_) *p = value;
    }
  }
}
```

There are two other optimisations going on here. The first is that we are using the pointer indexing operator `[]`. Most compilers treat `while (++i<n) { *(ptr++)=v; }` differently from `while (++i<n) { ptr[i]=v; }` , with the latter often being significantly faster. This is especially true when `ptr` is a pointer to a character sized type. The other optimisation makes use of the fact that it is faster to count down to 0 than count up to n. This is because it is faster to test against a constant, 0, than against a variable. Sometimes a compiler figures this out itself, but by no means always. One useful refinement that may be possible is to decrement the index counter right at the end of the loop. This allows the compiler to avoid issuing a separate test instruction, since this sort of test is automatically performed by the processor after a decrement or other arithmetic operation.

Since we are performing the same operation on every pixel independent of its absolute or relative position, there is one further optimisation that can be performed. In many cases an image will be stored as a contiguous block of memory. If this is the case, it may make sense just to operate on this block of memory as a single dimensional array. In the case of fill, this may even allow a compiler to issue a specialised single machine instruction which performs the whole fill very very fast. This gives us our final implementation.

```
template<class T>
void vil_image_view<T>::fill(T value)
{
  T* plane = top_left_;

  if (is_contiguous())
  {
    vil_image_view<T>::iterator it = begin();
    vil_image_view<T>::const_iterator end_it = end();
    while (it!=end_it) { *it = value; ++it; }
    return;
  }

  if (istep_==1)
  {
    for (unsigned p=0;p<nplanes_;++p,plane += planestep_)
    {
```

```
        T* row = plane-1;
        for (unsigned j=0;j<nj_;++j,row += jstep_)
        {
          int i = ni_;
          while (i>=0) { row[i--]=value; }
        }
      }
      return;
    }

    for (unsigned p=0; p<nplanes_; ++p, plane+=planestep_)
    {
      T* row = plane;
      for (unsigned j=0; j<nj_; ++j, row+=jstep_)
      {
        T* p = row;
        for (unsigned i=0; i<ni_; ++i, p+=istep_) *p = value;
      }
    }
  }
```

This optimised version was between two and ten times faster than the original depending on the compiler, image structure, and pixel type.

It should always be born in mind that there is a trade-off in testing for special cases. Each test takes time, and this slows the function down for the non-special cases. Limit yourself to only testing for very common cases that have very significant potential speed improvements.

Finally as with all optimisation - be rigorous in comparing the actual times for your original and optimised code. Run enough experiments to measure the statistical spread to see if your improvements are significant. It is quite common for compiler or processor quirks to make your optimised code slower than the original.

7.9 Blocked Images (Advanced Topic)

7.9.1 Basics

It is possible to encounter images that are much larger than available memory. For example, a commercial satellite image can easily exceed several gigabytes in size. The situation is even more dire in the case of ultra high resolution video where up to 16K X 16K pixel resolutions are feasible at two bytes per pixel. It is clearly not practical to handle these images as an in-memory `vil_image_view`. The use of a `vil_image_resource` to supply small views of the image at a time is essential, however the overhead in extracting small views from a large image file can be substantial.

Consider the example of displaying a small image region near the center of the image where the view is zoomed in so that one pixel in the image is mapped to one pixel on the screen. The size of this image patch might be 2K X 1K pixels. In order for the image resource to supply this set of pixels it is necessary to seek past a gigabyte or more of file-resident data to the middle of the image and then pull out the several megabytes of pixels needed to construct the view for display. If the user then wants to pan over a few hundred pixels to view something just off the screen, a full seek and file read must be repeated. Under these circumstances, image viewing performance will be overwhelmingly dominated by disk io bandwidth and seek times.

To mitigate the overhead of disk access, the image can be organized as a set of contiguous rectangular blocks of pixels. Blocks may be randomly scattered within the file, but each block is a contiguous set of pixels. This way, a view can be assembled by seeking to each

block in the view and then reading the block efficiently. Typical block size is 512 X 512 or 1024 X 1024 pixels, so that only a few blocks are needed to display regions of interest at full zoom. To gain even more efficiency, the blocks can be managed in a cache so the most of the pixels being displayed on the screen are already in memory. As the user pans to a new location, those blocks that are now off the screen are replaced by new blocks needed to fill in the new region. Thus, the number of blocks that have to actually be read from the file is significantly reduced.

The blocked image resource interface has the following virtual methods in addition to those already defined in the base resource class:

The block size used to store and retrive pixels.

```
unsigned size_block_i() const
unsigned size_block_j() const
```

The number of blocks in column and row to contain the image.

```
unsigned n_block_i() const
unsigned n_block_j() const
```

Retrieving blocks from the resource. Note that a block is a `vil_image_view` and thus ready for use in processing and visualization operations.

```
vil_image_view_base_sptr
get_block( unsigned  block_index_i, unsigned  block_index_j ) const

bool
get_blocks(unsigned start_block_i, unsigned end_block_i,
           unsigned  start_block_j, unsigned end_block_j,
           vcl_vector< vcl_vector< vil_image_view_base_sptr > >& blocks ) const
```

This blocking structure is used internally to implement the basic method

```
get_copy_view(unsigned i0, unsigned n_i, unsigned j0, unsigned n_j)
```

It is possible that `i0, n_i` and `j0, n_j` are not evenly divisible by `size_block_i` and `size_block_j`, respectively. In this case the blocks are trimmed to extract pixels belonging to the specified image view bounds. In the case of retrieving views near the boundary of the full image, e.g., `n_i=ni(), n_j=nj()`, blocks may lie partially outside the underlying image. In this case the pixel values in the block locations lying outside the full image bounds are undefined.

Similar methods are defined for inserting blocked data into the image resource.

```
bool put_block(unsigned  block_index_i, unsigned  block_index_j,
               const vil_image_view_base& view)

bool
put_blocks(unsigned start_block_i, unsigned end_block_i,
           unsigned  start_block_j, unsigned end_block_j,
           vcl_vector< vcl_vector< vil_image_view_base_sptr > > const& blocks)
```

These methods are used internally to support the virtual

```
put_view(const vil_image_view_base& im, unsigned i0, unsigned j0)
```

method. Note that current vil file-based resources do not support reading and writing on the same open resource. Therefore, a block-oriented image processing algorithm will have an input resource from which blocks are retrieved and an output resource where processed blocks are inserted.

7.9.2 The Facade and Cached Resource

Many of the advantages of blocking can be realized even if the underlying image resource is not intrinsically blocked. The `vil_blocked_image_facade` wraps around any image resource and provides the `vil_blocked_image_resource` class interface. That is, the

facade is a sub-class of `vil_blocked_image_resource`. Internally, reading and writing facade block data is implemented using the usual get and put view methods. In this case the block view dimensions are those defined by the facade blocking geometry.

One might wonder how this simulation of a blocked image structure provides any gain in efficiency for pixel access, since the process relies on an unblocked file format. A significant gain in performance can be gained by the addition of a cache. The `vil_cached_image_resource` is a sub-class of `vil_blocked_image_resource` and provides an in-memory store for most recently retrieved blocks. The size of the cache (in number of blocks) is specified in the constructor:

```
vil_cached_image_resource(vil_blocked_image_resource_sptr bir,
                          const unsigned cache_size)
```

The cache is implemented as a priority queue based on the "age" of a block. The blocks in the queue are given a timestamp as they enter the queue. If a block is retrieved from the cache, then the timestamp is reset to the current time. Otherwise, blocks age as new blocks are entered into the cache. When the cache is full, the oldest block is discarded to make room for a new block. Note that the queue does not participate in writing blocks to a resource.

7.9.3 Using Blocked File Formats

A blocking capability of a resource can be determined by examining the properties of the resource using the method

```
bool get_property(char const* tag, void* property_value = 0) const
```

Two properties are defined for blocked resources:

```
vil_property_size_block_i "size_block_i"
vil_property_size_block_j "size_block_j"
```

To test if a resource supports blocking one can examine the appropriate properties of the resource:

```
vil_image_resource_sptr imgr = vil_load_image_resource("my_filename");
...
unsigned sbi=0, sbj=0;
bool is_blocked =
  imgr->get_property(vil_property_size_block_i, &sbi) &&
  imgr->get_property(vil_property_size_block_j, &sbj);
...
```

If the resource is blocked then `is_blocked` will be true and the variables, `sbi, sbj`, contain the blocking structure for the resource.

The following example shows how to convert an image resource resource to a blocked file resource.

```
vil_image_resource_sptr imgr = vil_load_image_resource("my_filename");
unsigned size_block_i = 256, size_block_j = 256;
vil_blocked_image_resource_sptr bimgr =
        vil_new_blocked_image_resource("my_blocked_filename",
                                       imgr->ni(), imgr->nj(), imgr->nplanes(),
                                       imgr->pixel_format(),
                                       size_block_i, size_block_j, "tiff");
if (!vil_copy_deep(imgr, bimgr))
{ //report trouble
 ...
}
...
```

The new resource, `bimgr`, will store pixels in square, 256 X 256, blocks. `vil_copy_deep` automatically splits the input resource into strips if the image is too large to fit in

memory. However, to insure proper handing of block boundaries it is better to wrap the input resource in a facade with the same blocking structure as the output. That is,

```
...
vil_blocked_image_resource_sptr facr =
    vil_new_blocked_image_facade(imgr,sbi, sbj);

if (!vil_copy_deep(facr, bimgr))
{ //report trouble
 ...
}
...
```

Currently, the `tiff` file format and The National Image Transmission Format (`nitf`) image format provide a `vil_blocked_image_resource`, however the `nitf` format does not yet support writing.

7.10 Pyramid Images (Advanced Topic)

7.10.1 What are Pyramid Images?

As in the previous section on blocked images, the motivation for constructing a pyramid image is to manage large images without having to keep the entire image in memory. Satellite images can easily exceed all available random access memory so it is impossible to display an overview of the image. The blocked image strategy solves the problem of panning through a large image, but it does not solve the problem of zooming between different levels of detail. Even with blocking, the display of a complete overview requires that the entire image must be in memory.

The zooming problem can be solved by constructing a `vil_pyramid_image_resource`. This resource maintains a number of file-based copies of an image at different resolution scales. The original image is called the base image. Each reduced resolution image resource is called a level of the pyramid. Most typically, the levels of the pyramid differ by a factor of two in scale in each dimension. The limit of the size of a pyramid as the number of levels approaches infinity is $1 + 1/4 + .. = 1+1/3$. Thus, the worst case is 33% extra storage to represent all levels of detail.

It is not necessary to have a fixed scale difference between adjacent levels of the `vil_pyramid_image_resource`. When a user requests a `vil_image_view` at a particular scale, a view from the closest scale in the pyramid is generated. The interface for getting a view from a pyramid image is illustrated in the following code example. In this example, the pyramid is stored as a set of image files in a directory.

```
#include <vil/vil_load.h>
#include <vil/vil_pyramid_image_resource.h>
...
{
...
 vil_pyramid_image_resource_sptr pir =
                         vil_load_pyramid_resource("pyramid_dir");
 float actual_scale;
 vil_image_view<unsigned short> level_view =
                         pir->get_copy_view(0.25f, actual_scale);
 ...
}
```

This example shows the basic use of a pyramid resource where a level view 1/4 the scale of the base image is being retrieved. If the pyramid doesn't contain a level with a scale factor of exactly 0.25, the closest scale is returned and the scale of the closest level is returned in

`actual_scale`. The level view only requires 1/16 the number of pixels of the base image and can likely be held entirely in memory. However, the user of the view has to keep in mind that the image has been scaled down and must manipulate it appropriately.

For example, in rendering an image to the screen, the screen display scale factor must be compared to the level scale in order to determine the correct rendering scale. Suppose for example that a display screen has 1000 x 1000 elements and the base image of the pyramid is 15,000 by 15,000 pixels. The required rendering scale factor for the base image is 1/15. Suppose that the closest scale level in the pyramid is 1/16. The resulting level view is then rendered at a scale factor of 16/15 in order to fill the screen. Note however that only one million pixels are being processed instead of 225 million.

7.10.2 Subsampling the base image

Level images are formed by subsampling the original base image. In order to do this subsampling properly, it is necessary to observe the limitations imposed by the Nyquist sampling theorem. The sample rate must be greater than twice the highest spatial frequency in the image. Otherwise aliasing will occur, which appears as interference bands in the down-sampled image. The Nyquist sampling rate constraint can be achieved by spatially smoothing the image using a low pass filter. The filter is designed to remove spatial frequencies that exceed one half the sampling rate corresponding to the scale of the pyramid level.

For example, if the base image is being sampled at a scale of 0.5 (every second pixel in each image dimension) then the image must be pre-smoothed to remove spatial frequencies greater than 1/pixel. A simple filter for achieving this requirement is to form the average of the 2x2 pixel neighborhood in the base image corresponding to each pixel in the downsampled image. This smoothing does not remove all the higher spatial frequencies but they are significantly attenuated. Another common approach is to apply a Gaussian low pass smoothing kernel recursively to each level. Gaussian suppression of higher spatial frequencies is superior to block averaging. The Gaussian is cheap to compute since it is separable and can be formed by applying two 1-d convolutions.

The `vil_pyramid_image_resource` class provides the simple 2x2 averaging method for generating pyramid levels that are a factor of two apart in scale. The user can apply more sophisticated sampling schemes but this method is adequate for display purposes. Each level is generated accordingly by applying the static method `vil_pyramid_image_resource::decimate`.

```
#include <vil/vil_load.h>
#include <vil/vil_pyramid_image_resource.h>
...
{
...
  vil_image_resource_sptr image;

// generate an image at 1/2 the scale

 image =
   vil_pyramid_image_resource::decimate(base_image, "level_filename", "tiff");

// the base_image resource was generated previously
...
}
```

In the current implementation of the decimate method, the pyramid levels are generated as blocked images and so a resource file format that can support blocking must be used. This choice is primarily a matter of decimation processing efficiency and to manage level images that are still too large to fit in memory. In the example, the "tiff" file format is chosen since rectangular block structure is supported. If the input image is blocked then

its native block structure is used. Otherwise a default blocking (256 x 256) structure is used.

7.10.3 Storing the pyramid resource

It is necessary to have a file format that can store the multiple images required for the different resolution levels. The most obvious approach is to store the images as separate files in a directory. This format is called `vil_pyramid_image_list` and is designated by the `vil_file_format::tag()`, "pyil". There is no restriction on the format of the level files but applications of the pyramid are generally more efficient if the base image and the level files are blocked.

A second option for storing image pyramids is the `vil_tiff_pyramid_resource` with `vil_file_format::tag()`,"ptif". In this case, all the pyramid levels are saved in a single tiff file. There is no assumed order to the image headers in the file. The pyramid level scales are sorted by the resource to provide the required interface. The following example shows creating an output resource of each type and inserting the level image resources into each pyramid.

```
#include <vil/vil_new.h>
#include <vil/vil_image_resource.h>
#include <vil/vil_pyramid_image_resource.h>
...
{
// a list of image resources representing the pyramid levels

    vcl_vector<vil_image_resource_sptr> rescs;
...

// Generate a set of resources at multiple scales
...

// Construct a new multiple file pyramid resource

vil_pyramid_image_resource_sptr pyr_image_list =
        vil_new_pyramid_image_resource("pyramid_directory", "pyil");

// Construct a new single file tiff pyramid resource

vil_pyramid_image_resource_sptr pyr_tiff =
        vil_new_pyramid_image_resource("pyramid.tif", "ptif");

// Store image_resources into the pyramids

  for ( vcl_vector<vil_image_resource_sptr>::iterator rit = rescs.begin();
        rit != rescs.end();  ++rit)
  {
    pry_image_list.put_resource(*rit);
    pry_tiff.put_resource(*rit);
  }
...
}
```

Two methods are provided in `vil_new` that generate pyramid images in either the image list or tiff format, `vil_new_pyramid_image_list_from_base` and `vil_new_pyramid_image_from_base`. The following example demonstrates the use of each pyramid builder.

```
{
#include <vil/vil_new.h>
```

```
#include <vil/vil_image_resource.h>
#include <vil/vil_pyramid_image_resource.h>
...

vil_image_resource_sptr base_image;

// base_image is loaded or constructed
...

 unsigned number_of_levels = 7;
 bool copy_base = true;
// Generate a pyramid as an image_list (files in a directory)
 vil_pyramid_image_resource_sptr pyril =
     vil_new_pyramid_image_list_from_base("pyramid_directory_path",
                                          base_image,
                                          number_of_levels,
                                          copy_base,
                                          "tiff",
                                          "R");

// Generate a pyramid as a multi-image tiff file
 vil_pyramid_image_resource_sptr pytif =
    vil_new_pyramid_image_from_base("pyramid_file.tif"
                                    base_image,
                                    number_of_levels,
                                    "ptif",
                                    "temporary_dir_path");
...
}
```

In the image list pyramid the user can specify the format of the level image resource files. In the example the tiff format is specified. The last argument specifies the base name of the pyramid files, e.g., R0, R1, ... Rn-1, in the example. The variable `copy_base` indicates whether or not the base image is already in the directory. If not, then `base_image` is copied as a blocked image resource with default blocking (256 x 256). If a different blocking structure is desired, the base image can be wrapped in a `vil_blocked_image_facade` resource with the new blocking structure.

For the tiff-based pyramid it is necessary to provide a temporary directory to generate pyramid levels prior to inserting them into the single tiff file. Since the pyramid level images can still be too large for memory, they are constructed as file-based resources.

7.11 NITF image reading

The National Imagery Transmission Format (NITF) is a highly flexible and complex format for exchanging digital imagery and its support data. Our NITF implementation includes a framework for defining the "tagged record extensions" and "data extension segments" needed by your application. A framework example, along with the capabilities and current limitations, is summarized here.

The following code demonstrates how to define a tagged record extension:

```
vil_nitf2_tagged_record_definition::define("HISTOA", "Softcopy History")
  .field("SYSTYPE",    "System Type",                    NITF_STR(20))
  .field("PC",         "Prior Compression",              NITF_STR(12))
  .field("PE",         "Prior Enhancements",
     NITF_ENUM(4, vil_nitf2_enum_values()
```

```
      .value("EH08",   "Enhanced 8bpp")
                  ...
      .value("DGHC",   "Digitized hardcopy")
      .value("UNKP",   "Unknown")
      .value("NONE",   "None")))
   .field("REMAP_FLAG", "System Specific Remap",           NITF_INT(1))
   .field("LUT_ID",     "Data Mapping ID from ESD",        NITF_INT(2))
   .field("NEVENTS",    "Number of Processing Events",     NITF_INT(2))
   .repeat("NEVENTS", vil_nitf2_fields_definitions()
      .field("PDATE",   "Processing Date and Time",        NITF_DAT(14))
      .field("PSITE",   "Processing Site",                 NITF_STR(10))
      .field("PAS",     "Softcopy Processing Application", NITF_STR(10))
      .field("NIPCOM",  "Number of Image Proc. Comments",  NITF_INT(1))
      .repeat("NIPCOM", vil_nitf2_field_definitions()
         .field("IPCOM", "Image Processing Comment",       NITF_STR(80)))
      .field("IBPP",     "Image Bit Depth (actual) ",      NITF_INT(2))
                                                                  ...)
```

This code enables the contents of record extension "HIST0A" to be parsed; without it, the unrecognized record would be skipped. Repeating field values, such as "IPCOM", above, are represented as vectors. Conditional and variable-length fields are also supported, and C++ functors are used to evaluate expressions involving tags that specify the length or repetition of other tags.

Currently the library can only read, but not write, NITF 2.0 and 2.1 files, and includes the following capabilities:

- Files larger than 2GB are supported by building with flag USE˙LFS turned on
- All four NITF uncompressed data layouts are supported (IMODE=S, B, P, or R)
- Most NITF image data types, including 8-, 16-, 32- and 64-bit signed and unsigned integers, single- and double-precision floating point numbers, and boolean data are supported. Support for complex float data is implemented but not tested.
- Multiple images per file are supported, as are an arbitrary number of bands per image.
- Blocked images (NBPR > 1 or MBPC > 1) are supported.
- Images with look-up tables (LUTs) will read correctly, but client applications must query the image header for the LUT and apply it to the image data.
- JPEG-2000-compressed imagery is supported via a plug-in, as described in the next section.

The following capabilities are not yet implemented:

- writing NITF files
- parsing graphic segments
- parsing text segments
- bounds checking of numeric field values
- additional structured field formatters (e.g., some geocoordinate formats)
- other compression schemes (e.g., original JPEG, bi-level compression, vector quantization)

7.12 JPEG 2000 Support

VIL can be configured to support the reading of JPEG 2000 image files as well as NITF 2.1 images that are JPEG 2000 compressed. This section describes how to set up this capability.

7.12.1 Install the library

The decompression is handled by a third party library, ECW JPEG 2000 SDK, developed by ER Mapper. The library can be downloaded from www.ermapper.com, and is currently available under three different licensing schemes:

- a "free use" license (even for commercial applications) with the restriction that the compression code supports streams of length 500 MB or less (a moot restriction for VIL, which does not yet support file writing);
- a GPL-like "public use" license with no limitations on reading and writing;
- a "commercial use" license for commercial applications that require the ability to write JPEG 2000 streams longer than 500 MB.

The VXL wrappers around this library were developed using version 3.1 beta of this SDK and have also been tested using version 3.3 RC2, the latest version available on 4 April 2006.

ER Mapper provides ECW JPEG 2000 SDK with a variety of build systems. As described in the next section, VXL has been configured to use the most common one which yields separate NCSEcw and NCSUtil libraries. Most of the testing has taken place using the dynamically linked versions of these libraries, but the static versions should work too.

7.12.2 Configure VXL to use it

Once you have installed and built the ECW JPEG 2000 SDK, you must configure VXL to find it. Specify these three CMAKE variables:

- ECW˙INCLUDE˙DIR: ECW SDK include directory
- ECW˙ncsecw˙LIBRARY: NCSEcw library pathname
- ECW˙ncsutil˙LIBRARY: NCSEcw library pathname

When CMAKE creates your build files it will automatically add the appropriate source files and pre-processor definitions. Once VIL is built, test the JPEG 2000 decompression capability using the test program "test˙file˙format˙read" in project "vil˙test˙all". If things are set up correctly, these test program will report that these two tests passed:

- JPEG 2000 [j2k,jpc]
- NITF 2.1 [nitf] (JPEG 2000 compressed)

Note that if you use the dynamically linked version of the ECW JPEG 2000 SDK, your PATH environment variable must contain the /lib directory that contains NCSEcw and NCSUtil.

8 vgl: Geometry

Chapter summary: This library provides geometric primitive entities, like points and lines and planes. The goal is to provide very lightweight structures than are just slightly more costly than matrices or vectors. At the same time these classes can support all the routine geometric computations that are needed in basic computer vision operations. The idea is that more complex spatial objects would use the vgl operations to carry out basic geometric computations without duplicating operations like line intersection.

8.1 Geometric primitives

The core geometry library `vgl` is intended to provide an environment for geometric primitives, both in Cartesian and homogeneous representations, and for both 2D and 3D.

This includes classes for

- Points, lines and planes.
- 2D conics.
- Rectangular bounding boxes.
- Polygons.
- Direction vectors.
- Region scan iterators.

In addition, the `vgl/algo` library contains functions to perform elementary geometric operations like intersecting two lines, finding the nearest point on a line or a conic, computing the cross ratio of four points, lines or planes, ... For convenience, this functionality is put in a "name space", separate for 2D and 3D, and separate for Cartesian and homogeneous representations.

All representation classes are templated on the computational numeric type, typically `double` or `float`, but it could make sense to use other types like `int` (especially with homogeneous representations) or e.g. `vnl_rational` or `vcl_complex<double>`.

8.1.1 Homogeneous 2D classes and operations

The most general geometric framework is based on projective geometry and homogeneous coordinates. Projective operations arise in the analysis of image geometry and its relationship to the world geometry projected into the image by perspective cameras.

The basic 2D classes using homogeneous (3-argument) representations are:

- `vgl_homg_point_2d<T>`
- `vgl_homg_line_2d<T>`
- `vgl_conic<T>`

Some useful functions can be found in `vgl_distance.h` and `vgl_closest_point.h`:

- `double vgl_distance(vgl_homg_point_2d<T> const& p1, vgl_homg_point_2d<T> const& p2)`
- `double vgl_distance(vgl_homg_point_2d<T> const& p, vgl_homg_line_2d<T> const& l)`
- `vgl_homg_point_2d<T> vgl_closest_point(vgl_homg_line_2d<T> const& l, vgl_homg_point_2d<T> const& p)`

The most useful static functions in namespace `vgl_homg_operators_2d<T>` are:

- `double distance_squared(vgl_homg_point_2d<T> const& point1, vgl_homg_point_2d<T> const& point2)`
- `vgl_homg_line_2d<T> join(vgl_homg_point_2d<T> const& point1, vgl_homg_point_2d<T> const& point2)` to get the line through two points.

- `vgl_homg_point_2d<T> intersection(vgl_homg_line_2d<T> const& line1, vgl_homg_line_2d<T> const& line2)` to get the intersection point of two lines.
- `void unitize(vgl_homg_point_2d<T> &a)` to normalize a point representation (3-tuple) to unit magnitude.
- `double cross_ratio(vgl_homg_point_2d<T> const& p1, vgl_homg_point_2d<T> const& p2, vgl_homg_point_2d<T> const& p3, vgl_homg_point_2d<T> const& p4)`
- `double angle_between_oriented_lines(vgl_homg_line_2d<T> const& line1, vgl_homg_line_2d<T> const& line2)` Return the angle between the (oriented) lines (in radians).
- `double perp_distance_squared(vgl_homg_line_2d<T> const& line, vgl_homg_point_2d<T> const& point)`
- `vgl_homg_line_2d<T> perp_line_through_point(vgl_homg_line_2d<T> const& line, vgl_homg_point_2d<T> const& point)`
- `vgl_homg_point_2d<T> perp_projection(vgl_homg_line_2d<T> const& line, vgl_homg_point_2d<T> const& point)`
- `vgl_homg_point_2d<T> midpoint(vgl_homg_point_2d<T> const& p1, vgl_homg_point_2d<T> const& p2)`
- `vgl_homg_point_2d<T> lines_to_point(vcl_list<vgl_homg_line_2d<T> > const& lines)` to intersect a set of 2D lines to find the least-square point of intersection.
- `vcl_list<vgl_homg_point_2d<T> > intersection(vgl_conic<T> const &c, vgl_homg_line_2d<T> const &l)` to find all real intersection points of a conic and a line (between 0 and 2).
- `vcl_list<vgl_homg_point_2d<T> > intersection(vgl_conic<T> const &c1, vgl_conic<T> const &c2)` to find all real intersection points of two conics (between 0 and 4).
- `vcl_list<vgl_homg_line_2d<T> > tangent_from(vgl_conic<T> const &c, vgl_homg_point_2d<T> const &p)` returns the (at most) two tangent lines that pass through p and are tangent to the conic.
- `vgl_homg_point_2d<T> closest_point(vgl_conic<T> const& c, vgl_homg_point_2d<T> const& p)` returns the point on the conic closest to the given point.

Homogeneous projective geometry can be converted to standard Euclidean or Cartesian coordinates by normalizing by the third homogeneous coordinate. In general there will be need to convert back and forth between the two geometries to carry out operations efficiently. For example intersection of lines in homogeneous coordinates can be carried out simply using the cross-product of vectors.

8.1.2 Cartesian (Euclidean) 2D classes

The basic 2D classes using non-homogeneous (2-argument) representations are:

- `vgl_point_2d<T>`
- `vgl_line_2d<T>`
- `vgl_line_segment_2d<T>`
- `vgl_box_2d<T>`
- `vgl_vector_2d<T>`

A line segment is a bounded part of a line, between two end points. A vector is a directional difference between two points. A box is a rectangular bounding box.

8.1.3 Homogeneous 3D classes and operations

The basic 3D classes using homogeneous (4-argument) representations are:

- vgl_homg_point_3d<T>
- vgl_homg_plane_3d<T>
- vgl_homg_line_3d_2_points<T>

Some useful functions can be found in vgl_distance.h and vgl_closest_point.h:

- double vgl_distance(vgl_homg_point_3d<T> const& p1, vgl_homg_point_3d<T> const& p2)
- double vgl_distance(vgl_homg_plane_3d<T> const& p1, vgl_homg_point_3d<T> const& p2)
- double vgl_distance(vgl_homg_line_3d_2_points<T> const& p1, vgl_homg_point_3d<T> const& p2)
- double vgl_distance(vgl_homg_line_3d_2_points<T> const& p1, vgl_homg_line_3d_2_points<T> const& p2)
- vgl_homg_point_3d<T> vgl_closest_point(vgl_homg_plane_3d<T> const& l, vgl_homg_point_3d<T> const& p)
- vgl_homg_point_3d<T> vgl_closest_point(vgl_homg_line_3d_2_points<T> const& l, vgl_homg_point_3d<T> const& p)
- std::pair<vgl_homg_point_3d<T>,vgl_homg_point_3d<T> > vgl_closest_points(vgl_homg_line_3d_2_points<T> const& l1, vgl_homg_line_3d_2_points<T> const& l2)

The most useful static functions in namespace vgl_homg_operators_3d<T> are:

- T distance(vgl_homg_point_3d<T> const& point1, vgl_homg_point_3d<T> const& point2)
- T distance_squared(vgl_homg_point_3d<T> const& point1, vgl_homg_point_3d<T> const& point2)
- double perp_distance_squared(vgl_homg_line_3d const& line, vgl_homg_point_3d<T> const& point)
- vgl_homg_point_3d<T> intersect_line_and_plane(vgl_homg_line_3d const& , vgl_homg_plane_3d<T> const&) Return the intersection point of the line and plane.
- vgl_homg_point_3d<T> perp_projection(vgl_homg_line_3d const& line, vgl_homg_point_3d<T> const& point) Compute the perpendicular projection point of p onto l.
- double angle_between_oriented_lines(vgl_homg_line_3d const& line1, vgl_homg_line_3d const& line2) Return the angle between the (oriented) lines (in radians).
- vgl_homg_point_3d<T> lines_to_point(vcl_vector<vgl_homg_line_3d> const& line_list)
- vgl_homg_line_3d points_to_line(vcl_vector<vgl_homg_point_3d<T> > const& point_list)
- vgl_homg_line_3d planes_to_line(vcl_vector<vgl_homg_plane_3d<T> > const& plane_list) Return the intersection line of the planes.
- vgl_homg_plane_3d<T> points_to_plane(vcl_vector<vgl_homg_point_3d<T> > const& point_list)
- vgl_homg_point_3d<T> intersection_point(vcl_vector<vgl_homg_plane_3d<T> > const&) double Compute best-fit intersection of planes in a point.

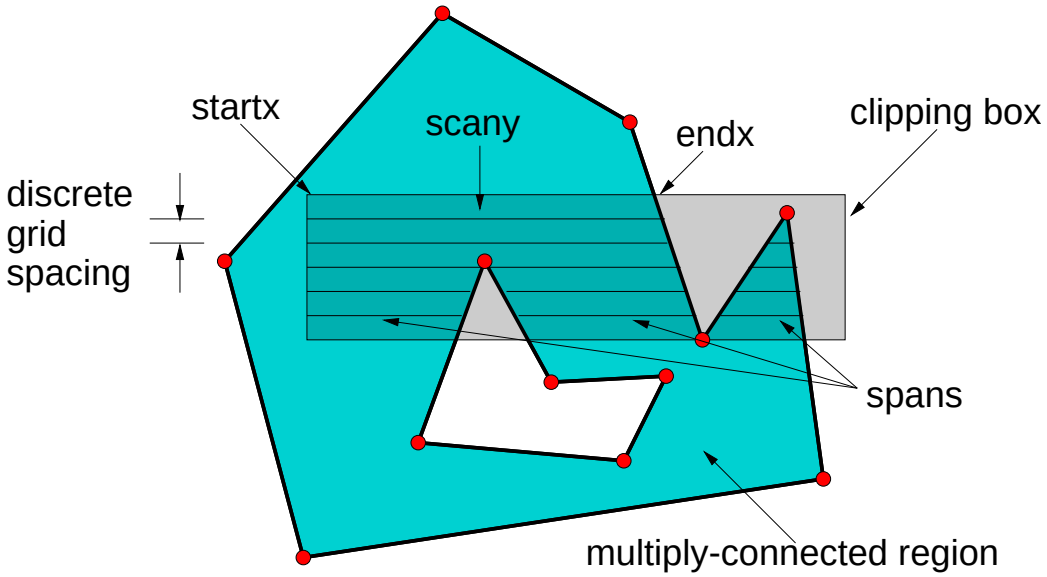

Figure 1: The general situation for a polygon scan iterator. The polygon defines a set of discrete scan lines which are bounded by the polygon or by an optional clipping window.

8.1.4 Cartesian 3D classes

The basic 3D classes using non-homogeneous (3-argument) representations are:

- `vgl_point_3d<T>`
- `vgl_plane_3d<T>`
- `vgl_line_segment_3d<T>`
- `vgl_box_3d<T>`
- `vgl_vector_3d<T>`

8.1.5 Homogeneous 1D classes

For sake of completeness, the following 1D representation classes are present:

- `vgl_homg_point_1d<T>`
- `vgl_1d_basis<T>`

A 1D basis is an arbitrary set of 3 (collinear) points. One receives coordinate 0 or (0,1), one has coordinate infinity or (1,0) and the unit point has coordinate 1 or (1,1). Such a set is an essential ingredient for any projective transformation.

8.2 2D regions and iterators

The `vgl_polygon<T>` class represents a more complex region or area in 2D space. The `vgl_region_scan_iterator` class allows for iterating through regions. More specifically, the derived classes `vgl_polygon_scan_iterator<T>`, `vgl_triangle_scan_iterator<T>`, `vgl_ellipse_scan_iterator<T>` and `vgl_window_scan_iterator<T>` can be used to iterate over the points of a discrete grid that are interior to the region. The general case for the polygon scan is shown in Figure 1.

The scan is initiated by calling `::reset()`. The boolean method `::next()` then iterates over the spans until all spans have been produced. When no more spans are available `::next()` returns false. Thus an iteration over the interior of the polygon is executed as:

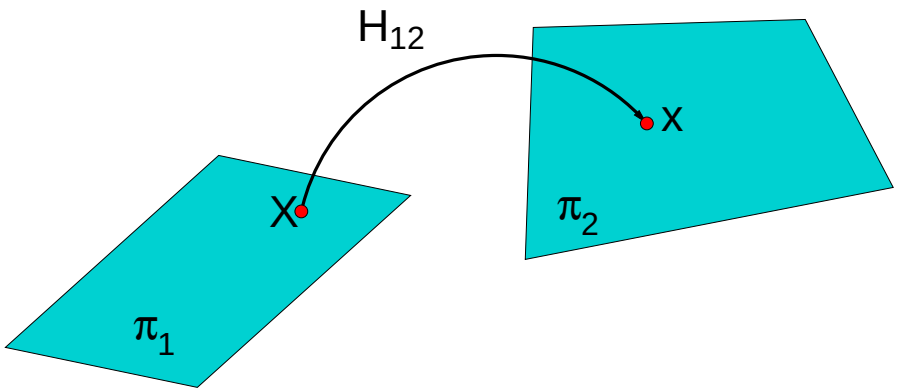

Figure 2: The mapping of points from one projective plane to another. The transformation can be represented by a 3x3 matrix.

```
...
vgl_polygon<double> my_polygon;
...
// do something to define the polygon
...
vgl_polygon_scan_iterator<double> psi(mypolygon);
psi.set_include_boundary(true); // optional flag, default is true
for (psi.reset(); psi.next(); ) {
  int y = psi.scany();
  for (int x = psi.startx(); x <= psi.endx(); ++x)
    ....
    // do something with x and y, e.g. compute the center of gravity of
    // the interior points.
}
```

The `vgl_polygon_scan_iterator<T>` also supports the specification of an optional clipping window. The window is intersected with the polygon to define scan region as shown in Figure 1. The window is specified by an alternative constructor:

```
vgl_polygon_scan_iterator<float>(vgl_polygon<float> const& face, bool boundaryp,
                                 vgl_box_2d<float> const& window)
```

Note that the boundaryp flag is defined to determine if points on the boundary of the polygon or window are to be included in the scan.

The area of a polygon can be determined with the `vgl_area` function.

8.3 Projective transformations

One of the goals of vgl is to support the basic operations of projective geometry. This goal inevitably entails the use of projective transformations which are typically represented as square matrices of dimension n+1, where n is the dimension of the geometric space in which a point set is embedded. Because of the strict rules of core libraries, vgl is not permitted to require other core libraries in order to carry out its operations. At the same time there is no justification to re-invent a numerical library within vxl simply to avoid library cross-linking. The solution is to define a `vgl/algo` library that can link to vnl and thus make use of the necessary vnl functions. A user can cleanly link to very basic vgl classes without including vnl, but the full operations of projective geometry will require the use of vnl matrix algorithms.

8.3.1 2-d projective transformations

Much of the discussion here will center on the projective plane (2-d points and lines) but a partial set of operations are available for both 1-d and 3-d geometry. The basic operation is to transform a point or line, and in the projective plane this corresponds to multiplying the vector of corresponding homogeneous coordinates by a 3x3 transformation matrix. This basic operation is illustrated in Figure 2. The projective transformation between planes is often called a planar homography, thus the symbol h, or H, is used to represent transformations.

The class `vgl_h_matrix_2d<T>` defines the basic operations of a planar homography. Points and lines are transformed by the operator `()` or the `*` operator as for example,

```
...
vnl_matrix_fixed <double, 3,3> M;
...
//define M somehow
...
vgl_h_matrix_2d <double> H(M);
vgl_homg_point_2d<double> X(1.0, 0.0, 1.0), x1, x2;
x1 = H(X);
x2 = H*X;
...
```

and x1 ~ x2, where ~ indicates that the two points are projectively equivalent. That is, their three homogeneous coordinates are within a scale factor of each other.

The method `preimage` effects the inverse transformation. So continuing with our example,

```
...
vgl_homg_point_2d <double> Xpre;
Xpre = H.preimage(x1);
..
```

and X ~ Xpre. This operation is carried out by inverting the forward transformation matrix.

The process is similar for lines, but it should be noted that the transformation for lines requires a different matrix then that for points. It can be shown that,

H˙line = (H˙point)^-t

where ^-t indicates the transpose of the inverse of a matrix. Thus, the forward transformation of a line from plane 1 to plane 2 requires a matrix inverse. On the other hand, the pre-image operation is easier, only requiring a matrix transpose.

The ambiguity as to whether an `vgl_h_matrix_2d<T>` is a point mapping or a line mapping is avoided by the convention that the class only refers to point mappings. The matrix inverse could be cached to avoid extra computation, but currently it is not, since there hasn't been performance issues to motivate the extra machinery. Still, the user should be aware that inversion is occurring on every forward line transformation.

8.3.2 3-d projective transformations

`vgl/algo` also provides a basic capability for carrying out 3-d to 3-d projective transformations, based on a 4x4 linear matrix multiplication. The most common application is to implement Euclidean transformations (rotation and translation) in a unified, compact framework. For example the following code defines a Euclidean transformation based on a given axis of rotation:

```
vgl_h_matrix_3d <double> Hrot, Htrans, H;

//Setup the rotation transformation
vnl_vector_fixed<double, 3> axis(0.0,0.0,1.0);//The z axis
```

```
double angle = vnl_math::pi/4.0;//45 degree rotation
Hrot.set_rotation_about_axis(axis, angle);

//Set up the translation transformation
Htrans.set_translation(1.0, 2.0, 3.0);

//compose the two.  The rotation is applied first and then the translation
H = Htrans*Hrot;
...
// Transform a 3-d homogeneous point
vgl_homg_point_3d<double> X(1.0, 0.0, 0.0, 1.0), x;
x = H(X);
//
//The resulting transformed point
// x = (1.707, 2.707, 3.0, 1.0)
```

In 3-d projective space, points and planes hold the same dual relationship as points and lines do in 2-d projective space. In 3-d, planes are transformed by H^-t, where H is a 3-d projective transformation on points.

Lines in 3-d are considerably more complicated than points and planes. The representation called Plucker coordinates which allows computations involving lines to be carried out using matrix and vector operations. It is planned to introduce Plucker geometry into `vgl/algo` when the need arises.

8.4 Computing 2-d projective transformations

A standard coordinate frame is defined in projective geometry, called the *projective basis*. The basis is constructed from four points and in the coordinate frame of the basis these points have coordinates as follows:

p[0]	p[1]	p[2]	p[3]
1	0	0	1
0	1	0	1
0	0	1	1

Note that the first two points are *points at infinity*, or *ideal* points that indicate the direction of the x and y coordinate axes. The third point is the origin and the last point is called the *unit* point and is at Euclidean coordinates (1,1). The method `bool projective_basis(vcl_vector<vgl_homg_point_2d<T> > const & four_points)` sets the transformation so as to map points from their projective plane to the plane defined by the canonical basis.

The more general case is based on classes that compute a plane projective transformations based on sets of corresponding points or lines. For example,

```
vgl_h_matrix_2d_compute_linear hcl;
vcl_vector <vgl_homg_point_2d <double> > point_set1, point_set2;
...
//fill these two vectors with corresponding points
//taken from two projective planes,
//e.g. a world plane and the image plane.
//
vgl_h_matrix_2d <double> H = hcl.compute(point_set1, point_set2);
// H represents the homography that
// transforms points from plane1 into plane2.
```

This functionality is very useful in tracking planar surfaces in images and in the calibration of perspective cameras. Currently, only linear algorithms are available for finding the

homography that best fits the mapping between two sets of corresponding points or two sets of corresponding lines. It is planned to add non-linear compute methods, such as Levenberg-Marquardt.

9 vsl: Binary I/O

Chapter summary: This section describes how to save and restore objects using a binary stream. It details how to add the appropriate functions to each class to make use of this facility.

All objects in VXL should be able to save themselves to a stream (eg a file) and restore (load) themselves from a stream (file). The main functions provided for this purpose are `vsl_b_write(os,object);` and `vsl_b_read(is,object&);`.

The binary IO for the core libraries (vbl, vil, vgl and vnl) is implemented in 'clip-on' libraries which live in the `io` subdirectories of each library (thus the declaration of the function `vsl_b_write(vsl_b_ostream&,const vnl_vector&);` lives in the file '`vnl/io/vnl_io_vector.h`'.

However, it is recommended that I/O for other libraries be provided by writing `b_write(os);` and `b_read(is);` functions in each class. See the 'Design Notes' section below.

9.1 Supported Platforms

The binary I/O code is known to work across the following hardware/OS/compiler combinations, but probably also works on most other platform/compiler combinations:

1. Intel - Linux - gcc-2.95 and gcc-3.0
2. Intel - WindowsNT - vc++
3. SGI - MIPS - MipsPRO CC
4. Sun - Solaris - gcc-2.95
5. DEC alpha - OSF - gcc-2.95 and gcc 3.0 (64 bit!)

Thus binary files produced by any of the above should be readable by any other of the above. There is of course a minor exception: large numbers (like integers larger than 4294967295) saved on a 64-bit platform cannot be read on a 32-bit platform.

9.1.1 caveats

The code has been designed to work on as many platforms as possible. However if your platform uses any of the following, then it will probably not work (as presently coded.)

1. A middle endian word encoding scheme.
2. Chars of length other than 8 bits.
3. Non-IEEE format floats and doubles.

9.2 Using Binary I/O

To save an object to a file, simply do the following:

```
vxl_myclass my_object;

// Fill my_object

vsl_b_ofstream bfs("my_object.bvl");
if (!bfs)
{
    vcl_cerr<<"Failed to open my_object.bvl for output."<<vcl_endl;
}
else
```

```
{
    vsl_b_write(bfs,my_object);
    bfs.close();
}
```

To load/restore the object from a file:

```
vxl_myclass my_new_object;

vsl_b_ifstream bfs("my_object.bvl");
if (!bfs)
{
    vcl_cerr<<"Failed to open my_object.bvl for input."<<vcl_endl;
}
else
{
    vsl_b_read(bfs,my_object);
    bfs.close();
}
```

It is recommended that the default extension name for your binary files is `.bvl`. This extension does not appear to be used by any other program. In many cases however, you will want to pick a new extension to indicate the contents of a file. For example, we store active shape model objects with ending `.asm`.

The classes `vsl_b_ifstream` and `vsl_b_ofstream` are simple wrappers around real `vcl_ifstream` and `vcl_ofstream` objects. These wrappers ensure that you open a file with CR/LF conversion turned off, and they should also allow lots of common misuses to be caught at compile time.

The functions `vsl_b_write(os,X)` and `vsl_b_read(is,X)` are defined for all reasonable cases, including all inbuilt types, most classes in vcl and the classes in the core vxl libraries.

When you write a new class, you should add the appropriate functions to allow easy use of binary I/O (see below).

Or for simplicity we provide the utility functions which would allow you to write:

```
#include <vsl/vsl_quick_file.h>
vxl_myclass my_object,my_new_object;

vsl_quick_file_save("my_object.bvl",my_object);
vsl_quick_file_load("my_object.bvl",my_new_object);
```

9.2.1 Saving multiple objects

One can use exactly the same approach to save a set of objects

```
vxl_myclass my_object;
vxl_my_other_class my_other_object;

// Fill objects
// ...

vsl_b_ofstream bfs("my_object.bvl");
if (!bfs)
{
    vcl_cerr<<"Failed to open my_object.bvl for output."<<vcl_endl;
}
else
{
    vsl_b_write(bfs,my_object);
```

```
    vsl_b_write(bfs,my_other_object);
    bfs.close();
}
```

(and similarly for loading them).

A standard rule for ensuring trouble free I/O is

> Always write the input and output code in tandem - the output should precisely mirror the input.

9.2.2 Binary I/O by baseclass pointer

When using polymorphism, there are frequently times when one needs to save and restore an object just using a base class pointer to it. `vsl` provides facilities to do this.

Assuming class `my_derived` is derived from class `my_base`, the following will work.

To save an object by baseclass:

```
my_derived d;

my_base *b = &d;

vsl_b_ofstream bfs("data.bvl");
vsl_b_write(bfs,b);
...
```

To restore an object:

```
// Make application aware of possible classes that it might see in the file

vsl_add_to_binary_loader(my_derived());
vsl_add_to_binary_loader(my_derived2());
...


my_base *b = 0;

vsl_b_ifstream bfs("data.bvl");
vsl_b_read(bfs,b);
// b now points to the correct class which has been created
// on the heap and filled with the data from bfs
...
```

Note that the read function will only work if the application has been made aware of each of the possible derived classes that it might come across in the file. This is done using calls to `vsl_add_to_binary_loader(my_derived())` (see appendix for details).

To reduce the pain of doing this, many libraries have a function that adds all the relevant derived classes (eg `xxxx_add_all_binary_loaders()` where `xxxx` is the library name).

9.2.3 Which files do I need to include/link?

In general the vsl˙b˙read and vsl˙b˙write functions use Koenig Lookup - that is the location of their declaration depends on their parameters.

The `vsl_b_stream` objects and `vsl_b_write` and `vsl_b_read` functions for fundamental data types are declared in `<vsl/vsl_binary_io.h>`. If you want to load or save a `vcl_vector`, the appropriate `vsl_b_write` and `vsl_b_read` functions will be in `<vsl/vsl_vector_io.h>`. Likewise for most of the other vcl classes. The `vsl` library contains the implementation of all of this.

When reading/writing by baseclass pointer, you need to include `vsl/vsl_binary_loader.h`.

If you want to load or save a `vgl_point_2d`, you will need to include `<vgl/io/vgl_io_point_2d.h>` and similarly for all other Level-1 VXL libraries. You will need to include the vgl_io library. For Level-2 libraries, the situation varies. If binary io has been defined at all for a level-2 library, it might be included in the library itself, e.g. the io functions for vpdfl_gaussian are declared in the same file as the Gaussian, `vpdfl/vpdfl_gaussian.h>`. Alternatively, it might be in a clip-on library in the same form as the Level-1 libraries above.

9.2.4 How to save templated objects

The situation for templated objects is the same as above, except that you need to ensure that the appropriate (templated) `vsl_b_read` and `vsl_b_write` functions are explicitly instantiated. This instantiation is achieved by placing a file in the relevant "Templates" folder.

An example template file, is shown below. It enables saving of a 2d array of `"hjk_model"`s (a completely made up plain class).

```
// file = my_module/hjk/Templates/vbl_array_2d_io+hjk_model~-.cxx
#include <vbl/io/vbl_io_array_2d.txx>
#include <hjk/hjk_model.h>
VBL_IO_ARRAY_2D_INSTANTIATE(hjk_model);
```

The `vbl_io_array_2d.txx` file contains the `VBL_IO_ARRAY_2D_INSTANTIATE` macro and the `hjk_model.h` file contains the io header declarations for a plain class.

Another example template file, allowing the saving of a vector of `vgl_point_2d` objects, is shown below.

```
// file = my_module/hjk/Templates/vsl_vector_io+vgl_point_2d~-.cxx
#include <vsl/vsl_vector_io.txx>
#include <vgl/io/vgl_io_point_2d.h>
VSL_VECTOR_IO_INSTANTIATE(vgl_point_2d<double>);
```

The `vsl_vector_io.txx` file contains the `VSL_VECTOR_IO_INSTANTIATE` macro and the `vgl_io_point_2d.h` file contains the io header declarations for `vgl_point_2d<double>`.

You should now be able to load and save templated objects with lines such as:-

```
vcl_vector<hjk_model> hjk_model_vec;
vsl_b_ofstream bfs("hjk_model_vec.bvl");
if (!bfs)
{
    vcl_cerr<<"Failed to open hjk_model_vec.bvl for output."<<vcl_endl;
}
else
{
    vsl_b_write(bfs,hjk_model_vec);
    bfs.close();
}
```

NB, the template instantiation files should be placed in your own libraries (ie here "hjk") to avoid creating unnecessary and unused versions of a given templated function.

9.3 Tidy Printing with `vsl_indent`

The utility functions and class in `vsl_indent` give a way of putting indentation into output streams to give more legible printed output.

If each class implements it's printing (print(os) or print_summary(os)) in such a way that at the beginning of each new line one inserts an indentation using

```
os<<vsl_indent()<<"Rest of stuff.."<<vcl_endl;
```

and increases and decreases the current indentation for the stream with `vsl_indent_inc(os)` and `vsl_indent_dec(os)`, then one can easily generate readable output for complex nested sets of classes.

It's use is best described by example:

```
vcl_cout<<vsl_indent()<<"No Indent"<<vcl_endl;
vsl_indent_inc(vcl_cout);
vcl_cout<<vsl_indent()<<"1 Indent"<<vcl_endl;
vsl_indent_inc(vcl_cout);
vcl_cout<<vsl_indent()<<"2 Indent"<<vcl_endl;
vsl_indent_dec(vcl_cout);
vcl_cout<<vsl_indent()<<"1 Indent"<<vcl_endl;
vsl_indent_dec(vcl_cout);
vcl_cout<<vsl_indent()<<"No Indent"<<vcl_endl;
```

This produces output of the form

```
No Indent
  1 Indent
    2 Indent
  1 Indent
No Indent
```

Example of use in class output:

```
class Fred
{
public:
  void print(vcl_ostream& os) const { os<<vsl_indent(os)<<"Fred's data"; }
};

vcl_ostream& operator<<(vcl_ostream& os, const Fred& fred)
{
  os<<"Fred: "<<vcl_endl;
  vsl_indent_inc(os);
  fred.print(os);
  vsl_indent_dec(os);
  return os;
}

class Jim
{
private:
  Fred fred_;
public:
  void print(vcl_ostream& os) const
  {
     os<<vsl_indent()<<fred_<<vcl_endl;
     os<<vsl_indent()<<"Jim's other data";
  }
};

vcl_ostream& operator<<(vcl_ostream& os, const Jim& jim)
{
  os<<"Jim: "<<vcl_endl;
  vsl_indent_inc(os);
  jim.print(os);
  vsl_indent_dec(os);
  return os;
```

```
}

main()
{
  Jim jim;
  vcl_cout<<jim<<vcl_endl;
}
```

This produces output:

```
Jim:
  Fred's data
  Jim's other data
```

If Jim were then included as a member of another class, Harry, one could get output of the form

```
Harry:
   Harry's basic data
   jim1:
     Fred's data
     Jim's other data
   jim2:
     Fred's data
     Jim's other data
```

and so forth. The author humbly suggests that this makes the summaries quite readable.

9.4 Error Detection

IO is often prone to errors beyond the control of the programmer. In particular, files can be come corrupted, given to programs that can't read a new format, read on platforms that do not support large enough numbers.

vsl attempts to detect as many error conditions as possible. It prints an error message to `vcl_cerr` and sets the fail bit on the input stream. Any objects that were being loaded when the error occurred should be consistent at least as far as being able to delete the object safely.

During the opening of a binary input stream, vsl also checks for a schema version number, and magic number that confirm that the stream was written by vsl.

It is easy to detect the error condition as the example shows

```
vsl_b_ifstream bfs_in(path);
if (!bfs_in)
{
  vcl_cout << "Could not open " << path
           << " for reading as binary IO" << vcl_endl;
  return;
}
vsl_b_read(bfs_in, my_obj);
if (!bfs_in)
{
  vcl_cout << "Unable to read my_obj" << vcl_endl;
  return;
}
bfs_in.close();
```

Figure 1: A `vgui` image display application.

10 vgui: Graphical User Interface

Chapter summary: `vgui` is a user interface library for computer vision applications `vgui` supports the following general functions:

- Menus
- Displaying Images
- Displaying/Creating Geometric Features

The `vgui` design is based on the OpenGL graphics library, and is intended to be platform independent and adaptable to a wide range of GUI toolkits. The central `vgui` class is the `tableau` which is a region (or regions) of the screen for carrying out display and event processing. Various tableaux can be assembled and layered to create a complex GUI application. At the same time, each tableau is relatively simple and can often be used independently in a small application such a popup image displayer.

This chapter is concerned with basic vgui programming and does not consider the issues associated with adapting vgui to a new window system and GUI toolkit. The examples are demonstrated using the mfc implementation of `vgui`.

10.1 A First Example

A simple example will be useful to illustrate some of the basics of vgui. The appearance of an image displayer is shown in Figure 1.

The program, `basic01_display_image.cxx` that produced this display is provided in the examples directory of the `vgui` library, `vgui/examples/`. The code is reproduced below.

```
#include <vcl_iostream.h>
#include <vgui/vgui.h>
#include <vgui/vgui_image_tableau.h>
#include <vgui/vgui_viewer2D_tableau.h>
#include <vgui/vgui_shell_tableau.h>
int main(int argc, char **argv)
{
  vgui::init(argc, argv);
  if (argc <= 1)
  {
    vcl_cerr << "Please give an image filename on the command line\n";
    return 0;
  }

  // Load image (given in the first command line param)
  // into an image tableau.
  vgui_image_tableau_new image(argv[1]);

  // Put the image tableau inside a 2D viewer tableau (for zoom, etc).
  vgui_viewer2D_tableau_new viewer(image);

  // Put a shell tableau at the top of our tableau tree.
  vgui_shell_tableau_new shell(viewer);

  // Create a window, add the tableau and show it on screen.
  return vgui::run(shell, image->width(), image->height());
}
```

It will be useful to go through this example carefully since it brings out some of the important characteristics of programming with tableaux.

`vgui::init(argc, argv)`: The class `vgui` is the base management class for the GUI toolkits and handles overall operations. In this example, the desired toolkit is being potentially selected by arguments on the command line. Normally, the kit is not specified by the user but is the first element in a registry of toolkits. For example, on windows, the mfc toolkit is used by default. Currently `vgui` supports the following toolkits:

OpenGL Utility Toolkit (GLUT) - http://www.xmission.com/~nate/opengl.html

Qt Toolkit - http://doc.trolltech.com

Gnu Toolkit (Gtk) - http://www.gtk.org

Microsoft Foundation Classes (MFC) - http://msdn.microsoft.com/library

`vgui_image_tableau_new image(argv[1])`: The `vgui_image_tableau` is a very basic tableau that mainly handles the display of pixels on the screen and can provide properties of the image being displayed. We see a somewhat strange construction in the term `vgui_image_tableau_new`. The idea is that all the tableaux support smart pointers (see vbl/vbl_smart_ptr.h). However it is desirable to be able to cast up and down the tableaux class hierarchies. The machinery needed to do this is maintained by the `xxx_new` form of construction, rather than calling the tableau constructor directly. This mechanism will be discussed later in the context of building a new sub-tableau.

An equivalent, and perhaps clearer, form is:

```
vgui_image_tableau_sptr image = vgui_image_tableau_new(argv[1]);
```

The constructor reads the image file specified by `argv[1]` and then inserts it into the image tableau, image.

If an image is already available, then the construction can be carried out as follows:

```
vil_image img;
... // get the image somehow
vgui_image_tableau_sptr image = vgui_image_tableau_new(img);
```

If the `image_tableau` already exists, then one can change the image being displayed:

```
vil_image img1, img2;
... //get the images somehow
//construct the image_tableau with img1
vgui_image_tableau_sptr image = vgui_image_tableau_new(img1);
//change the image to img2.
image->set_image(img2);
```

`vgui_viewer2D_tableau_new viewer(image)`: Next, the `vgui_image_tableau`, `image`, is added to a `vgui_viewer2D_tableau`. The viewer is responsible for manipulating the pan and zoom states of the tableaux being viewed, i.e, those *below* the viewer in the tableaux hierarchy. In our simple example, only the image tableau is under the control of the viewer. Again, an alternative form for the construction is:

```
vgui_viewer2D_tableau_sptr viewer = vgui_viewer2D_tableau_new(image);
```

The viewer responds to a variety of events such as key presses, mouse motion and mouse clicks. See ⟨undefined⟩ [events], page ⟨undefined⟩. The current implementation has the following menu of event processing:

(CTRL + left mouse)	`zoom in`
(CTRL + middle mouse)	`pan`
(CTRL + right mouse)	`zoom out`
(CTRL + 'c')	`center`
(CTRL + 'x')	`resize`
(CTRL + '-')	`lower zoom factor`
(CTRL + '=')	`raise zoom factor`
('n')	`toggle aliasing`
('z')	`toggle zoom type`
('d')	`sweep zoom`

Run the `basic01_display_image` example and try the various event actions. Note that, on windows, a middle mouse is often hard to come by since the middle mouse button does not produce a middle mouse button event by default on Windows. If you are not getting middle mouse button events then look at Start->Settings->Control Panel->Mouse. Check on the "Button Actions" tab that your middle mouse button is set to "Middle" Alternatively, the use of scroll bars can substitute for panning by middle button mouse movement.

`vgui_shell_tableau_new shell(viewer)`: The shell tableau is a composite of three tableaux, including the viewer tableau just described. The second tableau is called `vgui_clear_tableau` and clears the display area on each draw operation. Without this clear function, the image display will include old renderings of the image in the background.

The third tableau is a `vgui_tview_launcher_tableau` which prepares and displays a graph illustrating the tableau layout. This graph can be displayed using the 'G' key-press event. The tableau graph for this example is shown in Figure 2. Extra annotations have been added for illustration. This graph is useful for debugging complex tableau configurations. If the mouse is clicked above a node in the graph, information about the tableau will be streamed to vcl˙cout.

`vgui::run(shell, image->width(), image->height())`: This last expression causes the shell tableau hierarchy to be displayed on the screen and to continuously process events. The window size is determined by the last two arguments. Note that the borders of the window are included in these values so, for small images, the margin widths can be significant and the entire image is not visible when the window is displayed.

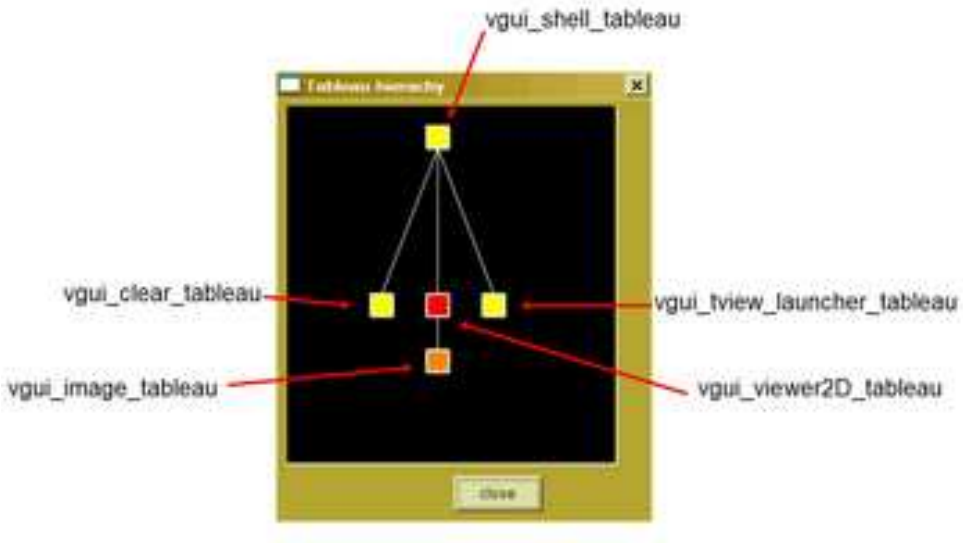

Figure 2: The nested tableaux form a tree structure. This structure can be displayed by the `vgui_tview_launcher_tableau` which is part of the composite `vgui_shell_tableau`. The edges in the graph are represented by instances of the class `vgui_parent_child_link`.

In programming with tableaux, it is often necessary to retrieve a particular tableau in a hierarchy such as the shell ↦ viewer ↦ image stack in the example. This access is provided by the method, `vgui_tableau_sptr::vertical_cast(vgui_tableau_sptr const& tab)`. The following code fragment will illustrate its use:

```
vgui_image_tableau_sptr get_image_tab(vgui_tableau_sptr const& tab)
{
  vgui_image_tableau_sptr i_tab;
  if (tab)
    itab.vertical_cast(vgui_find_below_by_type_name(tab,
                       vcl_string("vgui_image_tableau")));
  return i_tab;
}
```

If the input tableau, `tab`, is above an image tableau in the hierarchy then this routine will return it, otherwise the returned tableau will be null. One can also keep smart pointers to each tableau as members in an application class, which provides convenient access.

10.2 Tableaux des Tableaux

10.2.1 The Grid

In many computer vision applications, it is useful to be able to display multiple images in the same window. The `vgui_grid_tableau` is designed for this purpose. The each element of the grid holds a sub-hierarchy of tableaux. A simple example of multiple panes is provided by `vgui/examples/basic01a_multiple_panes.cxx`, reproduced below. The result of executing this program with two image paths supplied on the command line is shown in Figure 3.

```
#include <vcl_iostream.h>
#include <vgui/vgui.h>
#include <vgui/vgui_image_tableau.h>
#include <vgui/vgui_viewer2D_tableau.h>
#include <vgui/vgui_shell_tableau.h>
#include <vgui/vgui_grid_tableau.h>
int main(int argc, char **argv)
{
  vgui::init(argc, argv);
  if (argc <= 2)
  {
    vcl_cerr << "Please give two image filenames on the command line\n";
    return 0;
  }
  // Load two images(given in the first two command line args)
  // and construct separate image tableaux
  vgui_image_tableau_new image_tab1(argv[1]);
  vgui_image_tableau_new image_tab2(argv[2]);

  //Put the image tableaux into viewers
  vgui_viewer2D_tableau_new viewer1(image_tab1);
  vgui_viewer2D_tableau_new viewer2(image_tab2);

  //Put the viewers into a grid
  vgui_grid_tableau_sptr grid = new vgui_grid_tableau(2,1);
  grid->add_at(viewer1, 0,0);
  grid->add_at(viewer2, 1,0);
  // Put the grid into a shell tableau at the top the hierarchy
  vgui_shell_tableau_new shell(grid);

  // Create a window, add the tableau and show it on screen.
  int width = image_tab1->width() + image_tab2->width();
  int height = image_tab1->height() + image_tab2->height();
  return vgui::run(shell, width, height);
}
```

The use of tableaux is very similar to the first example, except that two image tableaux and two viewer2D tableaux are constructed. A 2x1 `vgui_vgrid_tableau` is constructed and the viewers are inserted in the left and right panes. Note that the order of the indices in the method, `vgui_grid_tableau::add_at(unsigned col, unsigned row)` is transposed from the order normally used for matrices, i.e. rows then columns. constructed.

In the case of multiple panes, it becomes an issue as to which pane is considered active. That is, suppose we wanted to replace the image in a pane selected by the user. How does the user indicate what pane is to be updated? A simple approach would be to have the user input the column and row of the grid cell to be updated using a menu (we will discuss menus in a later section). There are several additional grid methods that help define the grid cell that is to be operated on.

`void get_active_position(unsigned* col_pos,unsigned* row_pos)`: This method returns the column and row of the cell which is under the mouse cursor.

`void get_last_selected_position(unsigned* col_pos,unsigned* row_pos)`: This method returns the column and row of the last cell where the left mouse key was clicked.

An application can then use these selections to operate on the desired pane. For example, if a user wants to load and image from a file into a particular pane, they would click on

Figure 3: The `vgui_grid_tableau` supports the display of multiple panes.

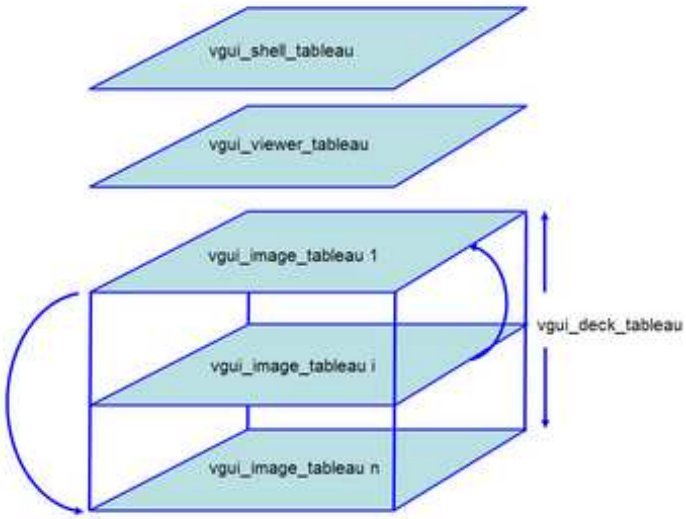

Figure 4: The `vgui_deck_tableau` supports the display of a stack of child tableaux. A typical application is to be able to page through a sequence of images. Only the tableau root on the top of the stack responds to events, such as pan and zoom.

the desired pane and then push the load-image menu. The menu callback routine would use the last˙selected˙position method to identify the appropriate `vgui_image_tableau`.

10.2.2 The Deck

Another useful capability is to stack displays in a pane. Then the application can "page" through the displays. Only the tableau hierarchy on the "top" of the deck responds to events, such as zooming and panning controls. The creation of a deck is illustrated by the example, `vgui/examples/basic01b_deck.cxx`. The concept of a deck is shown in Figure 4. The example code is:

```
#include <vcl_iostream.h>
#include <vnl/vnl_math.h>
#include <vgui/vgui.h>
#include <vgui/vgui_image_tableau.h>
#include <vgui/vgui_viewer2D_tableau.h>
#include <vgui/vgui_shell_tableau.h>
#include <vgui/vgui_deck_tableau.h>

int main(int argc, char **argv)
{
  vgui::init(argc, argv);
  if (argc <= 2)
  {
    vcl_cerr << "Please give two image filenames on the command line\n";
    return 0;
  }
  // Load two images(given in the first two command line args)
  // and construct separate image tableaux
  vgui_image_tableau_new image_tab1(argv[1]);
  vgui_image_tableau_new image_tab2(argv[2]);

  //Put the image tableaux into a deck
  vgui_deck_tableau_sptr deck = vgui_deck_tableau_new();
  deck->add(image_tab1);
  deck->add(image_tab2);

  vgui_viewer2D_tableau_new viewer(deck);

  // Put the deck into a shell tableau at the top the hierarchy
  vgui_shell_tableau_new shell(viewer);

  // Create a window, add the tableau and show it on screen.
  int width = vnl_math_max(image_tab1->width(), image_tab2->width());
  int height = vnl_math_max(image_tab1->height(), image_tab2->height());

  //Add 50 to account for window borders
  return vgui::run(shell, width+50, height+50);
}
```

The deck tableau responds to `vgui_PAGE_UP` and `vgui_PAGE_DOWN` events, which advance or backup the deck sequence.

10.3 Displaying Geometry

10.3.1 Displaying 2-d Features

`vgui` supports the *subject-view* programming pattern. That is, there is a clear separation between a class object, such as a line segment, and its *view* which is the manner in which it is rendered on the screen. Indeed, one can have many different views for a given object. The same line can be displayed with different line widths or colors, or even in an entirely different form such as a point in an image of Hough space, (rho, theta). Note, this subject-view approach is perversely called document-view in MFC.

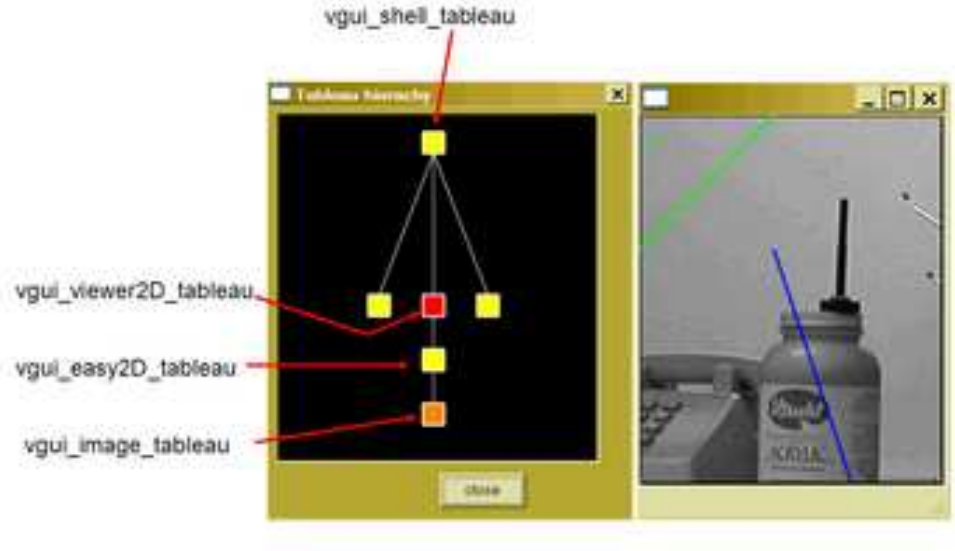

Figure 5: Geometry overlaid on the image is shown on the right. The tableau graph for this example is shown on the left.

The following example illustrates the ability of vgui to display geometric figures, and its result is shown in Figure 5.

```
#include <vcl_iostream.h>
#include <vgui/vgui.h>
#include <vgui/vgui_image_tableau.h>
#include <vgui/vgui_easy2D_tableau.h>
#include <vgui/vgui_viewer2D_tableau.h>
#include <vgui/vgui_shell_tableau.h>

int main(int argc, char **argv)
{
  vgui::init(argc, argv);
  if (argc <= 1)
  {
    vcl_cerr << "Please give an image filename on the command line\n";
    return 0;
  }

  // Load an image into an image.tableau
  vgui_image_tableau_new image(argv[1]);

  // Put the image.tableau into a easy2D tableau
  vgui_easy2D_tableau_new easy2D(image);

  // Add a point, line, and infinite line
  easy2D->set_foreground(0,1,0);
  easy2D->set_point_radius(5);
  easy2D->add_point(10, 20);

  easy2D->set_foreground(0,0,1);
  easy2D->set_line_width(2);
  easy2D->add_line(100,100,200,400);

  easy2D->set_foreground(0,1,0);
  easy2D->set_line_width(2);
  easy2D->add_infinite_line(1,1,-100);

  // Put the easy2D tableau into a viewer2D tableau:
  vgui_viewer2D_tableau_new viewer(easy2D);
  vgui_shell_tableau_new shell(viewer);

  // Create a window, add the tableau and show it on screen:
  return vgui::run(shell, image->width(), image->height());
}
```

The first new code we encounter in the example is:

```
vgui_easy2D_tableau_new easy2D(image);
```

or equivalently,

```
vgui_easy2D_tableau_sptr easy2D = vgui_easy2D_tableau_new(image);
```

The `vgui_easy2D_tableau` is responsible for rendering 2-d geometric shapes on top of its child, a `vgui_image_tableau`. The commands for inserting various geometric elements are of the form `add_xxx(...)`. `vgui_easy2D_tableau` assumes an elemental form of geometric specification, where the points and lines are directly specified by their parameters. The definitions for each add method used in the example are:

```
vgui_soview2D_point* add_point(float x, float y)
```

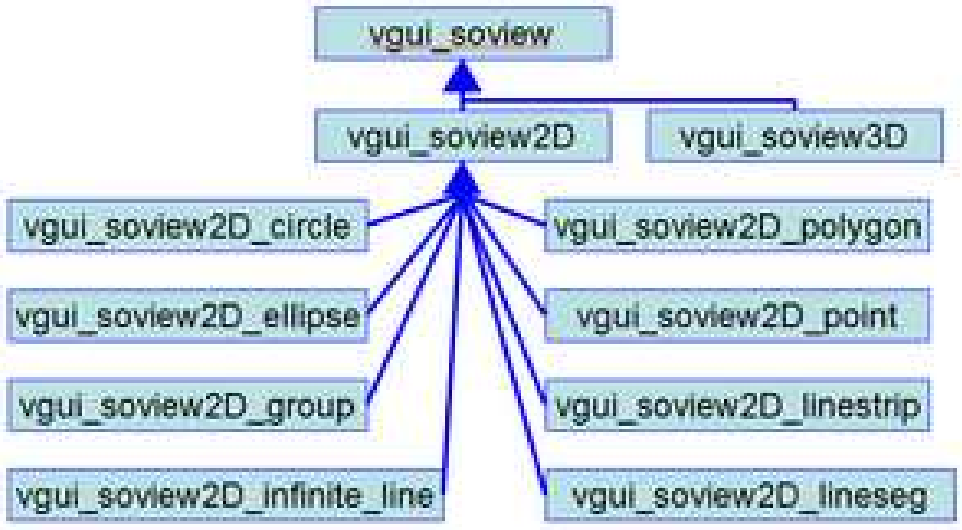

Figure 6: The hierarchy for 2-d soviews. `vgui` also has some support for 3-d rendering.

```
vgui_soview2D_lineseg* add_line(float x0, float y0, float x1, float y1)
vgui_soview2D_infinite_line* add_infinite_line(float a, float b, float c)
```

The point is defined by its location. The line segment is specified by the endpoints. The infinite line is specified by its line coefficients: $ax + by + c = 0$.

The appearance (or *style*) of the display is controlled by the following style specifiers:

```
void set_foreground(float r, float g, float b)
void set_line_width(float w)
void set_point_radius(float r)
```

The term *foreground* refers to the color of the displayed geometry. The style of each element added to `vgui_easy2D` after the `set_foreground`, `set_line_width`, and `set_point_radius` commands is assigned according to their specification until a new style command is issued.

A particular view of a geometric entity is specified by the class, `vgui_soview` which has the hierarchy shown in Figure 6. The constructor for a `vgui_soview` extracts the necessary information from the object to specify OpenGL rendering commands. The commands then add to the OpenGL display list to be rendered. While the current set is adequate for a wide range of computer vision programming, more advanced users will want to create their own `vgui_soview` subclass to provide convenient display interfaces for their objects, or to achieve special viewing capabilities.

10.3.2 Interactive Drawing of 2-d Features

It is often necessary to create geometric features such as a box to define a region of interest for image processing or a line for sampling pixels to provide an intensity plot. The following example shows how to create an interactive tool for drawing lines and circles. Interactive drawing consists of a tight loop of mouse position tracking and rendering so that the feature point tracks the mouse movements. This loop is called rubber-banding, since the feature seems to stretch and pull as the mouse moves. The rubberband loop is usually terminated by an event, such as a left mouse click. An example of a rubberband application is provided in `vgui/examples/basic10a_rubberband.cxx`. For this example, a left mouse click starts rubber-banding the feature and another left click terminates the rubber-banding and inserts the feature into the `vgui_easy2D_tableau`.

```
#include <vcl_iostream.h>
#include <vgui/vgui.h>
#include <vgui/vgui_menu.h>
#include <vgui/vgui_image_tableau.h>
#include <vgui/vgui_easy2D_tableau.h>
#include <vgui/vgui_rubberband_tableau.h>
#include <vgui/vgui_viewer2D_tableau.h>
#include <vgui/vgui_shell_tableau.h>

//global pointer to the rubberband tableau
static vgui_rubberband_tableau_sptr rubber = 0;

//the menu callback functions
static void create_line()
{
  rubber->rubberband_line();
}
static void create_circle()
{
  rubber->rubberband_circle();
}

// Create the edit menu
vgui_menu create_menus()
{
  vgui_menu edit;
  edit.add("CreateLine",create_line,(vgui_key)'l',vgui_CTRL);
  edit.add("CreateCircle",create_circle,(vgui_key)'k',vgui_CTRL);
  vgui_menu bar;
  bar.add("Edit",edit);
  return bar;
}

int main(int argc, char ** argv)
{
  vgui::init(argc,argv);
  if (argc <= 1)
  {
    vcl_cerr << "Please give an image filename on the command line\n";
    return 0;
  }
  // Make the tableau hierarchy.
  vgui_image_tableau_new image(argv[1]);
  vgui_easy2D_tableau_new easy(image);
  vgui_rubberband_easy2D_client* r_client =
    new vgui_rubberband_easy2D_client(easy);
  rubber = vgui_rubberband_tableau_new(r_client);
  vgui_composite_tableau_new comp(easy, rubber);
  vgui_viewer2D_tableau_new viewer(comp);
  vgui_shell_tableau_new shell(viewer);

  // Create and run the window
  return vgui::run(shell, 512, 512, create_menus());
}
```

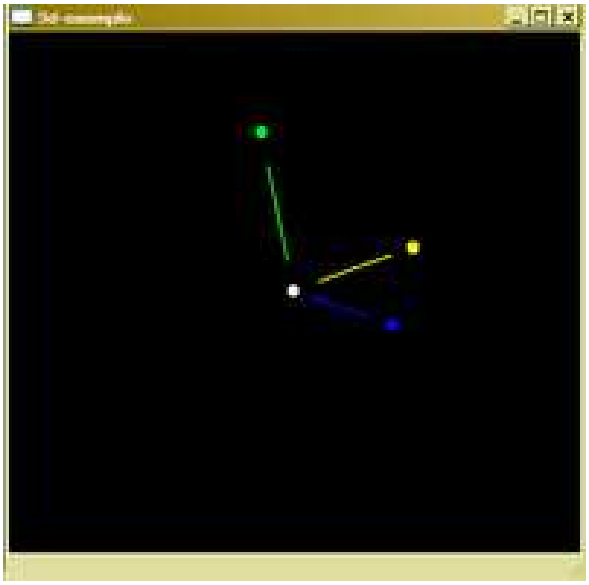

Figure 7: vgui's 3-d display.

This example introduces several new coding aspects to discuss.

`vgui_rubberband_tableau_new(new vgui_rubberband_easy2D_client(easy))`: The tight loop between mouse tracking and drawing requires a means for rendering the feature as its parameters are continuously adjusted by the mouse. The application could use OpenGL commands directly, but it is much simpler to use the drawing commands provided by the `vgui_easy2D_tableau` that was described earlier. See ⟨undefined⟩ [easy2D], page ⟨undefined⟩.

In order to link the rubberbanding and the drawing it is necessary to specify a class called the `vgui_rubber_band_client`. The client is assumed to be capable of drawing all the shapes that can be rubberbanded by the rubberband tableau. A subclass called `vgui_rubberband_easy2D_client` is defined in the `vgui_rubberband_tableau.h` header file and handles all the currently rubberbandable shapes.

`vgui_composite_tableau_new comp(easy, rubber)`: The rubberband tableau and its drawing client are included together in a composite tableau which forms the complete rubberband drawing capability.

10.3.3 Rendering 3-d Features

The 3-d display capabilities in vgui are not well-developed. Currently one can only display 3-d points and lines. The 3-d viewer does provide a "trackball" mode of interaction in viewing the 3-d geometry. An example of the `vgui` 3-d display is shown in Figure 7.

10.4 Menus

10.4.1 Basic Menus

`vgui` supports basic menu capabilities such as menu bars and popup menus as illustrated in the following example, `vgui/examples/basic05_menubar.cxx`:

```
#include <vcl_iostream.h>
#include <vgui/vgui.h>
#include <vgui/vgui_menu.h>
#include <vgui/vgui_image_tableau.h>
#include <vgui/vgui_viewer2D_tableau.h>
#include <vgui/vgui_shell_tableau.h>

// Set up a dummy callback function for the menu to call (for
// simplicity all menu items will call this function):
static void dummy()
{
  vcl_cerr << "Dummy function called\n";
}

// Create a vgui menu:
vgui_menu create_menus()
{
  vgui_menu file;
  file.add("Open",dummy,(vgui_key)'O',vgui_CTRL);
  file.add("Quit",dummy,(vgui_key)'R',vgui_SHIFT);

  vgui_menu image;
  image.add("Center image",dummy);
  image.add("Show histogram",dummy);

  vgui_menu bar;
  bar.add("File",file);
  bar.add("Image",image);

  return bar;
}

int main(int argc, char ** argv)
{
  vgui::init(argc,argv);
  if (argc <= 1)
 {
    vcl_cerr << "Please give an image filename on the command line\n";
    return 0;
 }

  // Make our tableau hierarchy.
  vgui_image_tableau_new image(argv[1]);
  vgui_viewer2D_tableau_new viewer(image);
  vgui_shell_tableau_new shell(viewer);

  // Create a window, but this time we also pass in a vgui_menu.
  return vgui::run(shell, 512, 512, create_menus());
}
```

The appearance of this program is shown in Figure 8.

By now, most of this code should follow a familiar pattern. The new element is the function `vgui_menu create_menus()`. The menu structure is assembled hierarchically

Figure 8: A `vgui` example involving menus. The menu bar on the top has sub-menus as indicated in the figure. In the example, all the menu choices call the same dummy function.

where the top-level menus have sub-menus which can have, sub-menus etc. The basic menu construction pattern is illustrated by the line:

```
file.add("Open",dummy,(vgui_key)'O',vgui_CTRL);
```

The first argument `"Open"` is a string representing the label of the menu item in the menu. The second argument is the name of the function to be called when the menu is selected. The last two arguments define a key-press configuration that will select the menu item without clicking on it with the mouse. In this case, the function `dummy()` is called by pressing the key combination, `CTRL + 'o'`.

10.4.2 Pop-up Menus

The appearance of a `vgui` pop-up menu is shown in Figure 9. A pop-up menu is launched by pressing the right mouse button over the active application window. Typically the role of the pop-up menu is to present operations that are relevant to the context present when the right button is pressed. For example, if we are displaying an image, there would be image display or image processing operations presented in the menu. Another mode might dominate when the tableau contains only geometric features. In that case, the menu items might present geometric operations such as translation or rotation.

This display was created by the example `vgui/examples/basic06_popup.cxx`:

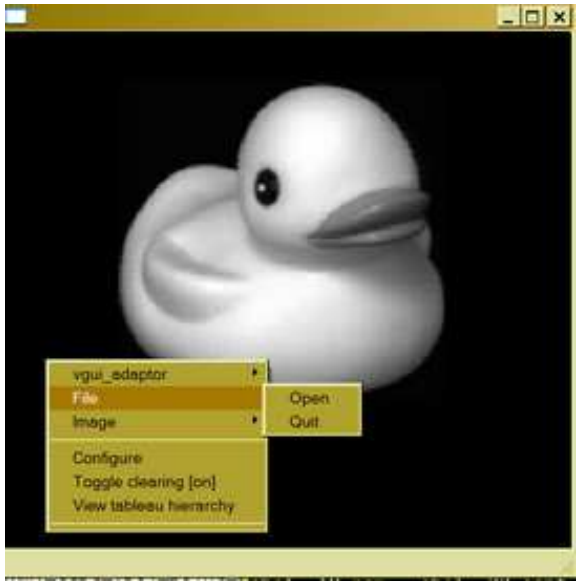

Figure 9: A `vgui` pop-up menu example. The menu shown was launched by pressing the right mouse key. A pop-up menu can have sub-menus, as shown in the figure.

```
#include <vcl_iostream.h>
#include <vgui/vgui.h>
#include <vgui/vgui_window.h>
#include <vgui/vgui_adaptor.h>
#include <vgui/vgui_menu.h>
#include <vgui/vgui_image_tableau.h>
#include <vgui/vgui_viewer2D_tableau.h>
#include <vgui/vgui_shell_tableau.h>
// Set up a dummy callback function for the menu to call (for
// simplicity all menu items will call this function):
static void dummy()
{
  vcl_cerr << "Dummy function called\n";
}

// Create a vgui_menu:
vgui_menu create_menus()
{
  vgui_menu file;
  file.add("Open", dummy);
  file.add("Quit", dummy);

  vgui_menu image;
  image.add("Center image", dummy);
  image.add("Show histogram", dummy);

  vgui_menu bar;
  bar.add("File",file);
  bar.add("Image",image);

  return bar;
}
```

```
int main(int argc, char ** argv)
{
  vgui::init(argc,argv);
  if (argc <= 1)
  {
    vcl_cerr << "Please give an image filename on the command line\n";
    return 0;
  }
  // Load an image into an image tableau:
  vgui_image_tableau_new image(argv[1]);
  vgui_viewer2D_tableau_new viewer(image);
  vgui_shell_tableau_new shell(viewer);

  // Create a window and add the tableau:
  vgui_window *win = vgui::produce_window(512, 512);
  win->get_adaptor()->set_tableau(shell);

  // Add our menu items to the base pop-up
  // (this menu appears when the user clicks
  //  the right mouse button on the tableau)
  win->get_adaptor()->include_in_popup(create_menus());
  win->get_adaptor()->bind_popups();
  win->show();
  return vgui::run();
}
```

This example contains a few new elements that should be discussed.

`vgui_window *win = vgui::produce_window(512, 512)`: In this case the widow is created before the display is launched. In the previous examples, the window was created at the time of launch using the `vgui::run(..)` command. Here the window is initialized to size 512x512 display resolution elements.

`win->get_adaptor()->include_in_popup(create_menus())`: This line introduces the class, `vgui_adaptor`. The idea of the adaptor is to provide a uniform interface for events across all toolkits. The adaptor also dispatches draw requests to the tableau hierarchy. A key role for the adaptor is to attach itself to a tableau using the method, `vgui_adaptor::set_tableau`. Then events received by the adaptor can then be passed down the tableau hierarchy. In the example, the `vgui_shell_tableau` is the root of the tableau hierarchy, and the adaptor is attached to the shell.

Each tableau in the hierarchy can add to the pop-up menu. In this example the specified menu is being included at the top-most level, i.e. the adaptor. It is also possible to define pop-up menu entries in each tableau in the hierarchy. To see an example of adding items to the pop-up menu by a tableau lower in the hierarchy, take a look at `vgui_clear_tableau::add_popup(..)`.

`win->get_adaptor()->bind_popups()`: This command binds the appropriate button and modifier to launch the pop-up menu. This method depends on the particular toolkit and is defined by the adaptor sub-class in the `vgui/impl/` sub-directory for the toolkit being invoked.

`win->show()`: This code causes the window to be exposed on the screen.

`vgui::run()`: This command tells the window to process all events until it is terminated. Unlike the previous examples, the window has been constructed in advance of the run command.

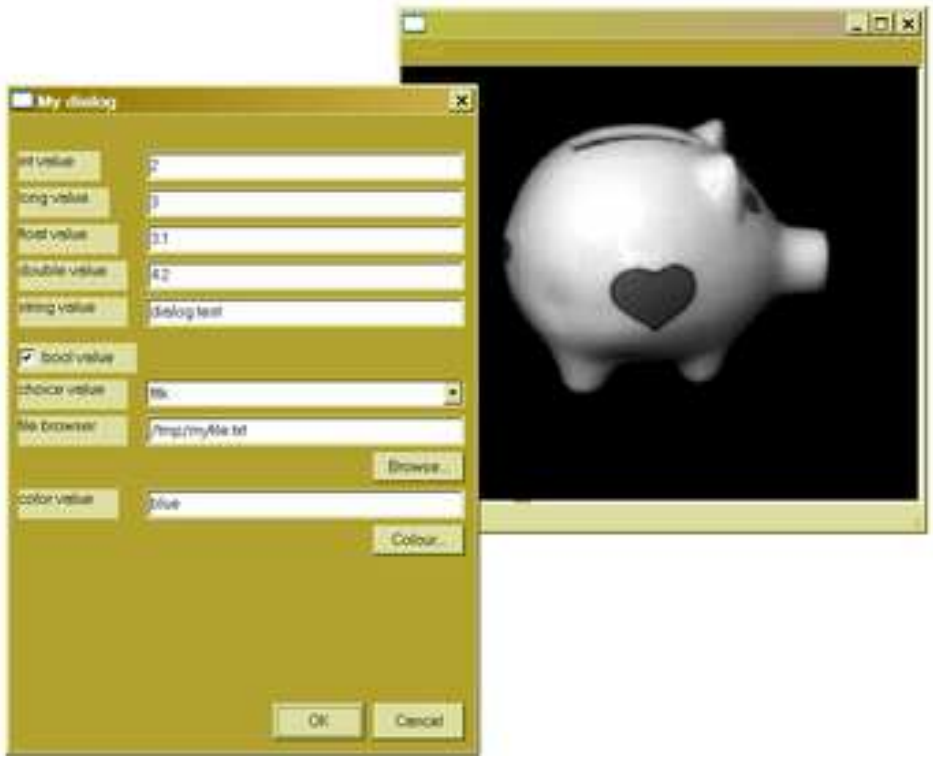

Figure 10: The dialog menu appears when a menu item is pressed that calls a function containing a dialog. The dialog displays names and values as well as boolean check boxes. The state of the boxes and values can be changed and when the dialog is dispatched, the altered values are bound to the variables in the dialog specification. In this simple example, the latest values are output to `vcl_cerr`.

10.5 Dialog Menus

It is often necessary to obtain values for parameters, such as edge detection thresholds or scales before applying the operation to an image. Also it is frequently necessary to obtain a file path string in order to read data such as image. These functions are satisfied by the *dialog menu*. An example of a dialog is shown in Figure 10, which corresponds to `vgui/examples/basic07_dialog.cxx`.

The following example illustrates these points.

```
#include <vcl_iostream.h>
#include <vbl/vbl_bool_ostream.h>
#include <vgui/vgui.h>
#include <vgui/vgui_menu.h>
#include <vgui/vgui_dialog.h>
#include <vgui/vgui_image_tableau.h>
#include <vgui/vgui_viewer2D_tableau.h>
#include <vgui/vgui_shell_tableau.h>
// Make a vgui_dialog:
static void test_dialog()
{
  static int int_value = 2;
  static long long_value = 3;
  static float float_value = 3.1f;
  static double double_value = 4.2;
  static vcl_string string_value = "dialog test";
  static bool bool_value = true;
  static vcl_string file_value = "/tmp/myfile.txt";
  static vcl_string regexp = "*.txt";
  static vcl_string color_value = "blue";

  static int choice_value = 1;
  vcl_vector<vcl_string> labels;
  labels.push_back(vcl_string("fltk"));
  labels.push_back(vcl_string("motif"));
  labels.push_back(vcl_string("gtk"));
  labels.push_back(vcl_string("glut"));
  labels.push_back(vcl_string("glX"));

  vgui_dialog mydialog("My dialog");
  mydialog.field("int value", int_value);
  mydialog.field("long value", long_value);
  mydialog.field("float value", float_value);
  mydialog.field("double value", double_value);
  mydialog.field("string value", string_value);
  mydialog.checkbox("bool value", bool_value);
  mydialog.choice("choice value", labels, choice_value);
  mydialog.inline_file("file browser", regexp, file_value);
  mydialog.inline_color("color value", color_value);

  if (mydialog.ask())
  {
    vcl_cerr << "int_value : " << int_value << vcl_endl
             << "long_value : " << long_value << vcl_endl
             << "float_value : " << float_value << vcl_endl
             << "double_value : " << double_value << vcl_endl
             << "string_value : " << string_value << vcl_endl
             << "bool_value : "
             << vbl_bool_ostream::true_false(bool_value) << vcl_endl
             << "choice_value : " << choice_value << ' '
             << labels[choice_value] << vcl_endl
             << "file_value: " << file_value << vcl_endl
             << "color_value: " << color_value << vcl_endl;
  }
}
```

```
static void test_dialog2()
{
  vgui_dialog mydialog("My dialog2");
  vgui_image_tableau_new image("c:/house11_small.jpg");
  vgui_viewer2D_tableau_new viewer(image);
  mydialog.inline_tableau(viewer, 512, 512);

  mydialog.message("A picture");

  vcl_string button_txt("close");
  mydialog.set_ok_button(button_txt.c_str());
  mydialog.set_cancel_button(0);
  mydialog.ask();
}

// Create a vgui.menu with an item which shows the dialog box:
vgui_menu create_menus()
{
  vgui_menu test;
  test.add("Dialog", test_dialog);
  test.add("Dialog2", test_dialog2);

  vgui_menu bar;
  bar.add("Test",test);

  return bar;
}
```

A dialog pops up when the associated menu item is selected. The dialog interface is reasonably self-explanatory, but it will be useful to discuss some of the main elements.

`static int choice_value = 1`: It is desirable to have the values in the dialog persist from one invocation to the next. This persistence is enabled through the use of static variables. Also note that the indexing of the choices starts at 1, i.e., "fltk" is the first element of the choice list. The appearance of the choice sub dialog is shown in Figure 11 a).

`mydialog.inline_file("file browser", regexp, file_value)`: This file browser dialog element is used extensively in applications. The interface is: `void inline_file(const char* label,vcl_string& regexp,vcl_string& filepath)`

The argument `label` is the displayed name of the dialog slot attached to the file browser. The string `regexp` defines a filter on the file extensions so that only a class of files will appear in the browser. For example, if only JPEG images are to be selected the argument assignment would be, `static vcl_string regexp = "*.jpg"`. The third argument is the result and returns the path to the selected file.

`if (mydialog.ask())`: This function pops up the dialog and waits for the `ok` or `cancel` button to be pushed. If `ok` is pushed then the function returns *true*, and the dialog values can be processed by the users application.

`mydialog.inline_tableau(viewer, 512, 512)`: Dialogs can contain an embedded hierarchy of tableaux. In this example an image viewer with pan and zoom capability is included in the dialog as shown in Figure 12. Any tableau can be inserted, such as `vgui_easy2D_tableau`, and its capability could be used to display geometric objects.

10.6 Event Processing

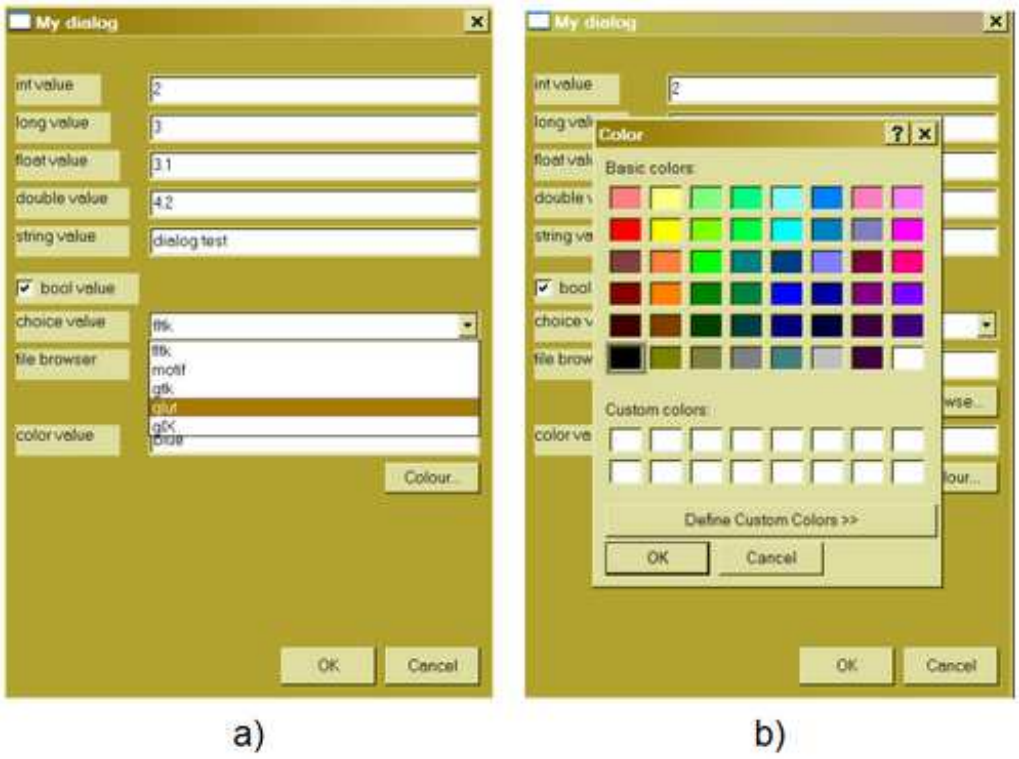

Figure 11: a) The choice option enables a selection from a set of alternative values. b) The color option enables a selection from a pallet of colors.

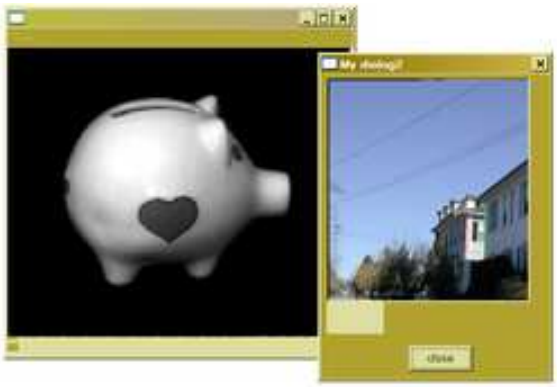

Figure 12: Dialogs can also contain embedded images or geometric figures.

As applications become more sophisticated, the programmer will need to be able to process events in a manner appropriate to customized interactive tasks. The `vgui` design has a simple interface for handling events as is illustrated by `vgui/examples/basic09_mouse_position`. The code for this example:

```
#include <vcl_iostream.h>
#include <vgui/vgui.h>
#include <vgui/vgui_image_tableau.h>
#include <vgui/vgui_viewer2D_tableau.h>
#include <vgui/vgui_shell_tableau.h>
//--------------------------------------------------------------
// A tableau that displays the mouse position
// when left mouse button is pressed.
struct example_tableau : public vgui_image_tableau
{
  example_tableau(char const *f) : vgui_image_tableau(f){ }

  ~example_tableau() { }

  bool handle(const vgui_event &e)
  {
    if (e.type == vgui_BUTTON_DOWN &&
        e.button == vgui_LEFT && e.modifier == 0)
    {
      vcl_cout << "selecting at " << e.wx << ' ' << e.wy << vcl_endl;
      return true; // event has been used
    }

    //  We are not interested in other events,
    //  so pass event to base class:
    return vgui_image_tableau::handle(e);
  }
};
//--------------------------------------------------------------
// Make a smart-pointer constructor for our tableau.
struct example_tableau_new : public vgui_image_tableau_sptr
{
  example_tableau_new(char const *f) : vgui_image_tableau_sptr(
    new example_tableau(f)) { }
};
```

```
//-------------------------------------------------------------
//The first command line argument is expected
// to be an image filename.
int main(int argc,char **argv)
{
  vgui::init(argc, argv);
  if (argc <= 1)
  {
    vcl_cerr << "Please give an image filename\n";
    return 0;
  }

  // Load an image into my tableau
  // (derived from vgui_image_tableau)
  vgui_tableau_sptr my_tab = example_tableau_new(argv[1]);

  vgui_viewer2D_tableau_new viewer(my_tab);
  vgui_shell_tableau_new shell(viewer);

  // Start event loop, using easy method.
  return vgui::run(shell, 512, 512);
}
```

This example contains a number of new concepts that are important to building custom applications. The example illustrates how to go about creating a new tableau, which is a subclass of an existing tableau.

`struct example_tableau : public vgui_image_tableau`: The new tableau is a child of the `vgui_image_tableau`. The tableau is being defined as a `struct`, however it could also be a class.

`struct example_tableau_new : public vgui_image_tableau_sptr`: vgui makes use of smart pointers extensively to prevent memory leaks. The smart pointer maintains a reference count so that an object (or struct) can be deleted when the reference count goes to zero, i.e., no one is referencing a pointer to the object. A drawback to the smart pointer approach is that inheritance is not maintained, that is, the smart pointer of a child of a class is not a sub-class of the smart pointer of the class. `vgui` has implemented a solution to this problem by creating a hierarchy of smart pointers that match the tableau hierarchy.

`bool handle(const vgui_event &e)`: The base class for all tableaux, `vgui_tableau` defines the virtual method, *handle*, which is called as events are passed down the tableau hierarchy. The event is initially captured by the `vgui_adaptor` which then passes it to its base tableau. If there is no handle method defined for a tableau, then the event is processed by its parents.

`return vgui_image_tableau::handle(e)`: After a particular handle method has processed an event it, can mark it as used (*true*) or unused (*false*). If an event is considered used by a tableau then it is eliminated and not presented to any other tableau. If the event is unused, then it is further dispatched to child tableaux or returned to the parent.

`if (e.type == vgui_BUTTON_DOWN..)`: `vgui` represents a spectrum of event types, which are summarized at this point for convenient reference.

It is often required to include the definition of a tableau smart pointer in other class implementations. When a new tableau is created it is convenient to define the smart pointer in a '`xxx_tableau_sptr.h`' file as follows:

```
#include <vgui/vgui_tableau_sptr.h>
```

```
class xxx_tableau;
typedef vgui_tableau_sptr_t<xxx_tableau> xxx_tableau_sptr;
```

10.6.1 vgui Events

The general types of events handled by vgui are:

- drawing
- mouse button down
- mouse button up
- mouse motion
- key press (and key modifiers)
- timers

The following is a partial list of the event enum symbols defined in vgui_event.h. Not all of the events enumerated in vgui_event are described here. Some of them seem to be vestigial and not exploited in code. A few comments will be added where the event function is not obvious.

vgui_BUTTON_DOWN=vgui_MOUSE_DOWN=vgui_MOUSE_PRESS

vgui_BUTTON_UP=vgui_MOUSE_UP=vgui_MOUSE_RELEASE

vgui_DRAW, vgui_DRAW_OVERLAY and vgui_RESHAPE These events cause the OpenGL display list to be re-rendered. The draw events are generated by the virtual tableau methods, vgui_tableau::post_redraw() and vgui_tableau::post_overlay_redraw(). When a tableau creates or modifies some displayable features, it is necessary to post the redraw event to stimulate the OpenGL display list to be re-rendered. At present, the use of overlay˙redraw events is not extensive in vgui. The design idea was that one may not want to redraw the entire display list. In the current implementation, all these events are treated by redrawing the display list.

vgui_ENTER and vgui_LEAVE These events are triggered when the mouse enters/leaves the window.

vgui_HSCROLL and vgui_VSCROLL When a window has specified the addition of scroll bars, these events are emitted when the scroll bars are moved.

vgui_KEY_PRESS=vgui_KEY_DOWN

vgui_KEY_RELEASE=vgui_KEY_UP

vgui_MOTION This event occurs whenever the mouse is in motion over the window.

vgui_TIMER vgui allows users to set a timer and then have an event issued when the time interval is completed. The timer is set by the vgui_adaptor::post_timer(..) method. For example, the code e.origin->post_timer(100.0) retrieves the vgui_adaptor and issues a timer post of 100 milliseconds.

vgui_DESTROY This event is caused by issuing a vgui˙adaptor::post˙destroy() command. The purpose is to allow exit processing to be carried out before the application quits.

10.6.2 vgui Buttons, Keys and Modifiers

The following table defines the vgui enum symbols for buttons, keys and modifiers:

===== Buttons =====	
vgui_LEFT	left mouse button
vgui_MIDDLE	middle mouse button
vgui_RIGHT	right mouse button
===== Keys =====	
vgui_ESC	ascii 27
vgui_TAB	'\t'

vgui_RETURN	'\r'
vgui_NEWLINE	'\n'
vgui_F1	0x100 + 1
vgui_F2	0x100 + 2
vgui_F3	0x100 + 3
vgui_F4	0x100 + 4
vgui_F5	0x100 + 5
vgui_F6	0x100 + 6
vgui_F7	0x100 + 7
vgui_F9	0x100 + 8
vgui_F10	0x100 + 9
vgui_F11	0x100 + 10
vgui_F12	0x100 + 11
vgui_CURSOR_LEFT	0x100 + 12
vgui_CURSOR_UP	0x100 + 13
vgui_CURSOR_RIGHT	0x100 + 14
vgui_CURSOR_DOWN	0x100 + 15
vgui_PAGE_UP	0x100 + 16
vgui_PAGE_DOWN	0x100 + 17
vgui_HOME	0x100 + 18
vgui_END	0x100 + 19
vgui_DELETE	0x100 + 20
vgui_INSERT	0x100 + 21
===== Modifiers =====	
vgui_NULL	0x0
vgui_CTRL	0x1
vgui_SHIFT	0x2
vgui_META	0x4
vgui_ALT	0x8

When a key press event is dispatched it carries information that specifies the key and modifier structure. This code fragment illustrates the use of modified keys:

```
bool handle(vgui_event const & e)
{
  vgui_key k = e.key;
  vgui_modifier m = e.modifier;
  if (m & vgui_CTRL)
    if (k == 's')
     {
      // Do something appropriate for CTRL + 's'
      ...
     }
  return true;
}
```

The bit corresponding to the modifier is tested to see if further action switched by the actual key is warranted. Note that the key in a `vgui_event` is always lower case. This eliminates the ambiguity that might arise in the use of the SHIFT key and upper case vs lower case characters. The following table will illustrate the effect of various modifier combinations.

key press	modifier	key	ascii character
=======	=======	=======	=======
a	vgui_NULL	'a'	'a'
CTRL+a	vgui_CTRL	'a'	'^a'
SHIFT+a	vgui_SHIFT	'a'	'A'
/	vgui_NULL	'/'	'/'
?	vgui_SHIFT	'/'	'?'

If one wants to work directly with the actual ascii character pressed, then use `e.ascii_char`.

10.6.3 Event Condition

A convenient class, `vgui_event_condition` is defined to represent the occurrence of a particular event configuration. Its use is best illustrated by an example:

```
bool my_tableau::handle(const vgui_event &e)
{
   vgui_event_condition g0(vgui_LEFT, vgui_CTRL, false);
   if (g0(e))
      vcl_cout << "saw a left mouse button release with CTRL pressed event\n";

   // pass the event back to the parent tableau
   return vgui_my_parent_tableau::handle(e);
}
```

In this case a test for the indicated event condition is constructed and can be used to filter events passing into a tableau's handle method. The event condition class provides a compact and tidy way of expressing complex logic on modifiers, keys and buttons.

10.6.4 Mouse Position

An event passes back the position of the mouse when the event occurred. As was illustrated in the `basic09_mouse_position` example. However this position is in the coordinate system of the display window. Most computer vision applications require positions referenced to the coordinate system of the image being displayed, and expressed in pixels. The class `vgui_projection_inspector` provides methods for transforming between the window and image coordinate systems.

The transformation is illustrated by this code fragment:

```
// Get X,Y mouse position to display on status bar
// in image coordinates
bool my_image_tableau::handle(const vgui_event &e)
{
  if (e.type == vgui_MOTION && button_down == false)
   {
    float pointx, pointy;
    vgui_projection_inspector p_insp;
    p_insp.window_to_image_coordinates(e.wx, e.wy, pointx, pointy);
    int intx = (int)vcl_floor(pointx), inty = (int)vcl_floor(pointy);
    vgui::out << '(' << intx << ' ' << inty << ")\n";
   }
  return vgui_image_tableau::handle(e);
}
```

10.7 Building an Application

The essential elements to create a GUI application have been presented. In this section, a typical design for a main program and associated GUI management classes will be described.

10.7.1 The Manager

There is a tendency to pile a lot of menu callbacks and menu constructors into the main program. It is easy and convenient, but this approach doesn't stand up to evolution of

the program over time. The main program quickly becomes hopelessly cluttered with a tangle of processing code, menu callbacks and tableau specifications.

A much better approach is to separate the methods used to process callbacks and event handling into a singleton class called the *manager*. The manager can be a sub-class of the top-level tableau and thus provide custom processing of events by defining its own `::handle` method. The methods on the manager provide the implementation for the menu callbacks.

It is also better to separate the menu construction class from the main program, since menus also tend to grow in number and complexity as the application evolves.

The following example will illustrate these design principles. The manager class looks like:

```
#include <vil/vil_image.h>
#include <vgui/vgui_image_tableau_sptr.h>
#include <vgui/vgui_wrapper_tableau.h>

class basic_manager : public vgui_wrapper_tableau
{
 public:
  ~basic_manager();
  static basic_manager *instance();
  void quit();
  void load_image();
  void init();
  virtual bool handle(vgui_event const &);

 private:
  basic_manager();
  vil_image img_;
  vgui_image_tableau_sptr itab_;
  static basic_manager *instance_;
};
```

Some elements of the class design require explanation:

`static basic_manager *instance()`: This method returns a unique single instance of the `basic_manager` class. This design is called a *singleton* pattern and is used when the class must maintain a unique consistent state across applications accessing the class. This requirement frequently arises in event handling where global access to the same event process is required. In our example, All events are funneled through the same `basic_manager` instance.

`basic_manager()`: The constructor for this class is made private because the only way the class should be invoked is through the `::instance` method.

`virtual bool handle(vgui_event const &)`: Since the manager is a sub-class of a `vgui_wrapper_tableau` it inherits the `::handle` method. This inheritance enables the manager to implement its own event processing and then pass unused events onto the manager's tableau children.

`vgui_image_tableau_sptr itab_` : To provide convenient access, one can cache pointers to intermediate tableaux.

`void load_image()`: A typical menu callback method. This manager can be accessed by many different applications that need to load an image. Thus, the image GUI management code doesn't have to be continually rewritten.

The implementation of the `basic_manager` class is as follows:

```
#include <vcl_cstdlib.h> // for vcl_exit()
#include <vcl_iostream.h>
#include <vil/vil_load.h>
#include <vgui/vgui.h>
#include <vgui/vgui_dialog.h>
#include <vgui/vgui_viewer2D_tableau.h>
#include <vgui/vgui_shell_tableau.h>
#include <vgui/vgui_image_tableau.h>
#include "basic_manager.h"

//static basic_manager instance
basic_manager* basic_manager::instance_ = 0;

//insure only one instance is created
basic_manager *basic_manager::instance()
{
  if (!instance_)
  {
    instance_ = new basic_manager();
    instance_->init();
   }
  return basic_manager::instance_;
}

// constructor/destructor
basic_manager::basic_manager():vgui_wrapper_tableau(){}

basic_manager::~basic_manager(){}

void basic_manager::init()
{
  itab_ = vgui_image_tableau_new();//keep the image tableau handy
  vgui_viewer2D_tableau_sptr viewer = vgui_viewer2D_tableau_new(itab_);
  vgui_shell_tableau_sptr shell = vgui_shell_tableau_new(viewer);
  this->add_child(shell);
}
```

Most of this code should be clear. One subtle point is the method `void basic_manager::init()`. When the instance of `basic_manager` is created, the parent class, `vgui_wrapper_tableau`, is constructed using its default constructor. After it comes into existence, the rest of the tableau hierarchy can be inserted as a child of `basic_manager`. With this approach, there is no assumption required about the order of constructors.

```
//the event handler
bool basic_manager::handle(vgui_event const & e)
{
  if (e.key == 'b')
    vgui::out << "I saw a 'b'\n";
  //pass the event to the shell
  return this->child.handle(e);
}

void basic_manager::quit()
{
  vcl_exit(1);
}

void basic_manager::load_image()
{
  vgui_dialog load_image_dlg("Load image file");
  static vcl_string image_filename = "";
  static vcl_string ext = "*.*";
  load_image_dlg.file("Image Filename:", ext, image_filename);
  if (!load_image_dlg.ask())
    return;
  img_ = vil_load(image_filename.c_str());
  itab_->set_image(img_);
}
```

The manager has a simple basic handle method that looks for the letter '`b`'. All events are then passed to the child (shell) tableau for further processing. This routine could return `true` on the detection of the '`b`' event if it were desired not to have any of the child tableaux react.

The methods to support menu callbacks are implemented in the manager. For example, `basic_manager::load_image()` illustrates the use of a dialog which pops up when the "Load Image" menu is selected.

10.7.2 The Menus

A menu class is defined to package up the static callback functions required in the `vgui_menu` assembly. The `basic_menu` class is:

```
//basic_menus.h
class basic_menus
{
 public:
  static void quit_callback();
  static void load_image_callback();
  static vgui_menu get_menu();
 private:
  basic_menus(){};
};
//basic_menus.cxx
#include <vgui/vgui.h>
#include <vgui/vgui_key.h>
#include <vgui/vgui_modifier.h>
#include <vgui/vgui_menu.h>
#include "basic_manager.h"
#include "basic_menus.h"

//Static menu callback functions

void basic_menus::quit_callback()
{
  basic_manager::instance()->quit();
}

void basic_menus::load_image_callback()
{
  basic_manager::instance()->load_image();
}

//basic_menus definitions
vgui_menu basic_menus::get_menu()
{
  vgui_menu menubar;
  vgui_menu menufile;

  //file menu entries
  menufile.add( "Quit", quit_callback,(vgui_key)'q', vgui_CTRL);
  menufile.add( "Load Image", load_image_callback, (vgui_key)'l', vgui_CTRL);

  //main menu bar
  menubar.add( "File", menufile);
  return menubar;
}
```

Note that the menu callback functions are paired with methods on the manager. These menus can be exported to other GUI libraries so that the same menu functionality can be re-used. However, keep in mind that under Windows special measures must be taken to export static items.

10.7.3 The Main Program

The main program for the basic˙manager application is:

```
#include <vgui/vgui.h>
#include <vgui/vgui_adaptor.h>
#include <vgui/vgui_window.h>
#include <vgui/vgui_command.h>
#include <vgui/vgui_shell_tableau.h>
#include <vgui/internals/vgui_accelerate.h>
#include "basic_menus.h"
#include "basic_manager.h"

int main(int argc, char** argv)

  vgui::init(argc, argv);
  vgui_menu menubar = basic_menus::get_menu();
  unsigned w = 512, h = 512;
  vcl_string title = "REALLY BASIC";
  vgui_window* win = vgui::produce_window(w, h, menubar, title);
  basic_manager* bas = basic_manager::instance();
  win->get_adaptor()->set_tableau(bas);
  win->set_statusbar(true);
  win->enable_vscrollbar(true);
  win->enable_hscrollbar(true);
  win->show();
  return vgui::run();
```

Note that the basic manager instance is attached to the adaptor in order to receive events, by the expression, `win->get_adaptor()->set_tableau(bas)`. Note that the main program is now very simple and will stay uncluttered as the application grows.

Several new features have been included in the construction of this window:

- a title which appears at the top left of the window
- a status bar, which displays messages at the bottom of the application window.
- horizontal and vertical scroll bars for panning the tableaux below a `vgui_viewer2D_tableau`.

10.8 Summary of `vgui` Tableaux

The following is a summary of the tableau defined in `vgui`.

`active_tableau`	This tableau (or rather a tableau derived from it) can appear visible or invisible, and active or inactive by calling `toggle_active` and `toggle_visible`.
`blackbox_tableau`	A tableau for event record and playback.
`blender_tableau`	To use this tableau make a `vgui_image_tableau` containing one of the images to blend and a `vgui_blender_tableau` containing the other. Put them both in a `vgui_composite_tableau`. Set alpha to be less than one to see the blended image.
`clear_tableau`	A tableau that performs OpenGL clearing upon receipt of a `vgui_DRAW` event. It has no child tableau.

`composite_tableau`	The `vgui_composite_tableau` class can have any number of children, indexed from 0 upwards. The draw action of `vgui_composite_tableau` is to draw each of its children, in order, into the current context. Events reaching the `vgui_composite_tableau` are passed on to each child in turn, till it is handled, so that child 0, the first added, is the "top" tableau.
`deck_tableau`	For holding an ordered collection of child tableaux, only one of which is passed all events that the `vgui_deck_tableau` receives. The effect is a flick-book of tableaux where the currently active tableau can be changed using PageUp and PageDown.
`displaylist2D_tableau`	Display of two-dimensional geometric objects - a builder tableau usually sub-classed.
`displaylist3D_tableau`	Display of three-dimensional geometric objects - a builder tableau usually sub-classed.
`drag_tableau`	A drag event occurs when the user moves the mouse with one of the mouse buttons pressed down. In `vgui` there is no `vgui_DRAG` event (there is only `vgui_MOTION` for when the mouse is moving). So if you want to capture drag events you may find this tableau handy.
`easy2D_tableau`	Easy interface for displaying two-dimensional geometric objects (see `vgui_soview2D`) such as lines, points, circles, etc. can be added using add, or `add_point`, `add_line`, `add_circle`, etc.
`easy3D_tableau`	Easy interface for displaying three-dimensional objects (see vgui˙soview3D) can be added using add, or `add_point`, `add_line`, etc.
`enhance_tableau`	Magnify/display another tableau in a region around the mouse pointer. Useful for a roaming image-processing sub-window.
`function_tableau`	Allows a user to insert custom functions that are called when events such as draw, mouse up, motion .. etc occur.
`grid_tableau`	A tableau that renders its child tableaux as a rectangular grid.
`image_tableau`	A tableau that renders the given image using an `image_renderer`.
`listmanager2D_tableau`	A tableau that manages a set of `vgui_displaylist2D_tableau` children.
`loader_tableau`	A tableau which (optionally) loads given values for the projection and modelview matrices before passing control to its child. This is typically used to initialize GL before rendering a scene.
`poly_tableau`	A tableau which renders its children in sub-rectangles of its viewport. The `grid_tableau` is a sub-class of `poly_tableau`.
`quit_tableau`	A tableau which quits the application on receiving '`q`' or ESC.

`roi_tableau` — A tableau which makes an ROI of an image act like a whole image.

`rubberband_tableau` — A tableau for interactive drawing of lines, circles, boxes, etc.

`satellite_tableau` — Turns a non-tableau into a multi-tableau, or puts one tableau into two parts of the hierarchy simultaneously. Example: We are displaying two images, each in its own zoomer and we'd like to have a tableau which takes mouse events from one image and draws a line on the other; introduces a "crossover" in the tree which is difficult to handle without `vgui_satellite_tableau`.

`shell_tableau` — A shell tableau is a handy collection of things one often wants at the very top of one's tableau hierarchy. It is essentially an acetate with N utility tableaux at the bottom.

`text_tableau` — A tableau for rendering text. Each piece of text is associated with an integer handle through which it can be retrieved, moved about, changed or removed. This tableau will not display any text unless you have compiled with GLUT.

`tview_launcher_tableau` — A tableau that pops up tableau tree (tview) on '`G`'.

`tview_tableau` — Displays a tableau tree.

`viewer2D_tableau` — A tableau for zooming and panning 2-d renderings.

`viewer3D_tableau` — A tableau for manipulating 3-d rendered scenes (not completed).

`wrapper_tableau` — A base class tableau which insures only a single child. Useful as a base class for managers.

10.9 Advanced Topic: Image Display Range

Prior to Jan 2005, image display in `vgui_image_tableau` was limited to 256 (`vxl_byte`) levels per pixel component. Images with pixel data types having a larger dynamic range were clamped to the range of [0 255]. The `vgui_image_tableau` interface now has the method,

```
void set_mapping(vgui_range_map_params_sptr const& rmp)
```

which defines how images with a dynamic range larger than one byte are to be displayed. If the pointer `rmp` is null, then the previous vgui display process is carried out as the default, otherwise range mapping is invoked.

For example if the image has `unsigned short` pixels, the intensity can be anything in the range [0 65535]. To generate a meaningful display, a range, [min max], is specified such that all pixel intensities less or equal to min are mapped to 0 and all pixel intensities greater or equal to max are mapped to 255. Intensities inside the range are mapped to the [1 254] remaining display levels according to a gamma or inversion function, as will be described below.

The parameters of the mapping are:

`n_components_` — The number of components in the image. A grey level image has one component, a typical RGB image has 3 components.

`min_L_`	The minimum range value (luminance) for a grey level image.
`max_L_`	The maximum range value for a grey level image.
`gamma_L_`	The gamma factor in the exponential mapping of image intensity.
`invert_`	If true, then the image display is inverted to form a negative image.
`min_R_`	The minimum range value for the red channel in a color image.
`max_R_`	The maximum range value for the red channel in a color image.
`gamma_R_`	The gamma for red channel mapping.
`min_G_`	The minimum range value for the green channel in a color image.
`max_G_`	The maximum range value for the green channel in a color image.
`gamma_G_`	The gamma for green channel mapping.
`min_B_`	The minimum range value for the blue channel in a color image.
`max_B_`	The maximum range value for the blue channel in a color image.
`gamma_B_`	The gamma for green channel mapping.
`use_glPixelMap_`	If `true` the range map is processed by hardware when available.
`cache_mapped_pix_`	Under panning and zooming operations, it is not necessary to re-map the pixel intensities. The range mapped display can be cached to avoid mapping computation by setting `cache_mapped_pix_` to `true`.

The `gamma` function is defined as

$$\frac{Ig}{Imax} = \left(\frac{I}{Imax}\right)^{\frac{1}{gamma}}$$

Assume that the pixel intensity has been mapped to the range [0 1.0], e.g., I/Imax. The normalized intensity is raised to the power 1/gamma. The rationale for this definition is that a typical CRT display monitor has a non-linear response with exponential factor gamma. This correction compensates for the monitor response and achieves an overall linear intensity display.

In typical operation, the user will interactively adjust the min max values in a loop that displays the mapped image until a satisfactory display is produced. The loop should re-instantiate the parameter block on each iteration since the update is triggered by a change in the value of the `rmp` pointer. For example,

```
vgui_image_tableau_sptr itab = vgui_image_tableau_new();
...
//set an image on itab
...
//set up a mapping parameter block
unsigned short min_val = 10000, max_val = 40000;
float gamma = 1.0;
bool invert = false;
bool use_glPixelMap = true;
bool cache_buffer = true;

vgui_range_map_params_sptr rmp =
   new vgui_range_map_params(min_val,max_val, gamma,
                             invert, use_glPixelMap, cache_buffer);
//start range mapping
itab->set_mapping(rmp)
itab->post_redraw();

//change the range
rmp->min_val_ = 15000;
itab->post_redraw();

//the image display will be updated with the new range min value
```

An example of mapping is shown in Figure 13. An example of the inversion mapping for a color image is shown in Figure 14.

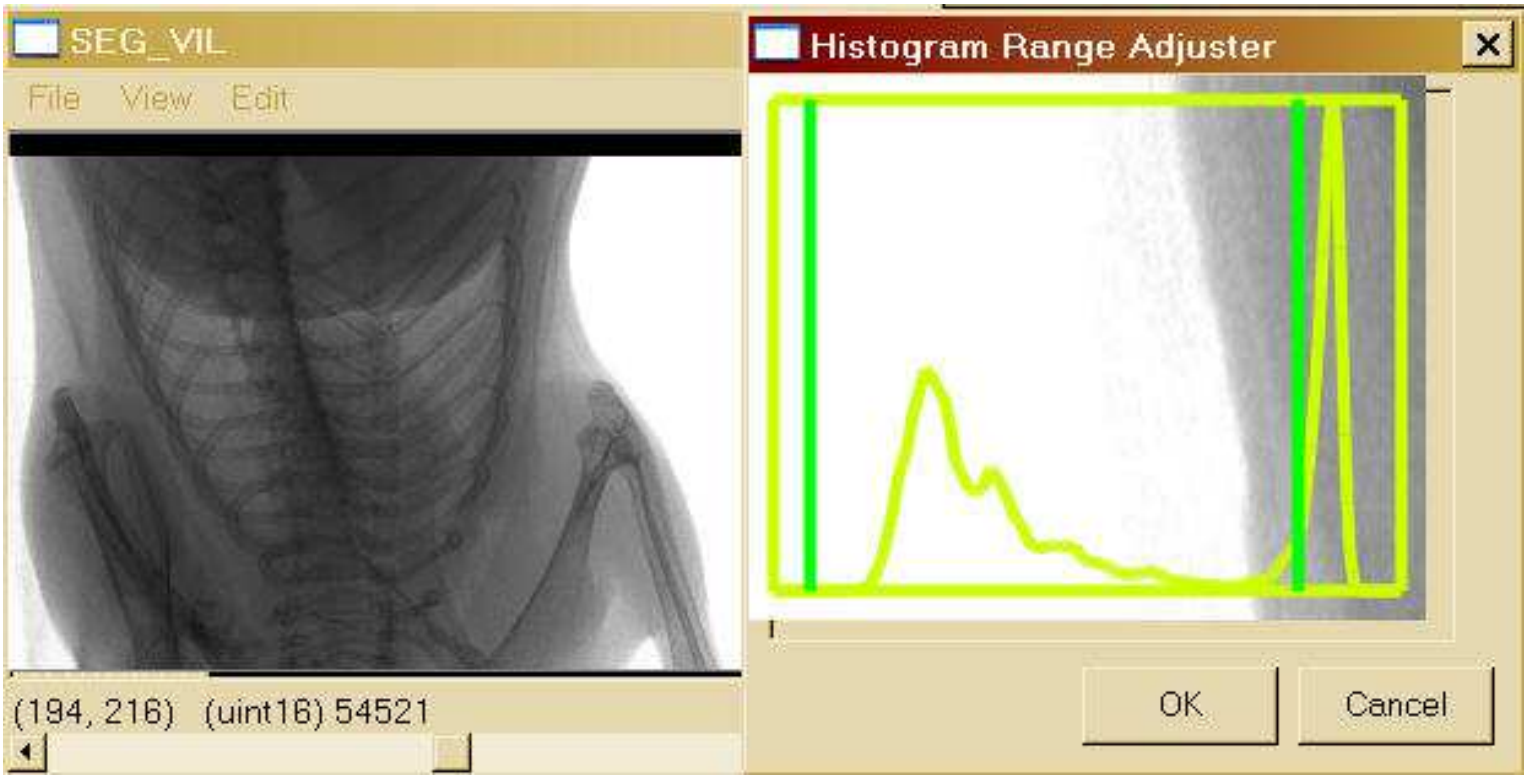

Figure 13: A display of an x-ray image with 16 bit unsigned short pixels. An inline tableau is used to adjust the range by moving the bars via mouse interaction. The image contrast is displayed simultaneously with the mouse motion. A histogram is also displayed to guide the user. Note the mouse position / image intensity display at the lower left indicates the pixel value in the proper units and range.

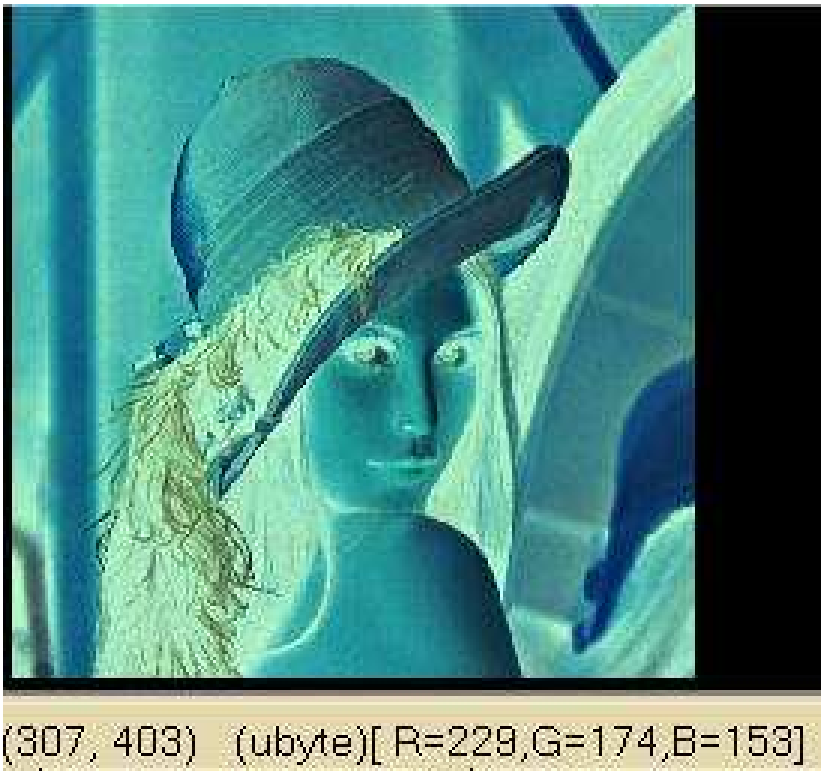

Figure 14: The use of range mapping to invert the image color channels. Note again that the mouse position /pixel intensity display provides the appropriate values corresponding to the original image, not the displayed image.

11 vidl: Video Access

Chapter summary: `vidl` is a library for managing video data. `vidl` supports the following general functions:

- Reading Video Files
- Saving Video Files
- Indexing through Video Files

`vidl` has the central notion of a *movie*, which is turn is a sequence of clips. A *clip* is simply a sequence of frames. A *frame* can return a `vil_image_view` corresponding to the frame, using the coder/decoder *codec* for a given video format.

Large videos can be read a frame at a time from the codec so it is not necessary to load an entire video into memory in order to access individual frames.

Currently vidl supports the following codecs:

- List of images
- AVI
- MPEG (not mature and read only)

There are plans to extend to DVD with a read-only capability

11.1 The vidl˙movie structure

The main interface class in `vidl` is `vidl_movie` it contains a list of clips. The class is meant to reflect a movie structure which is typically segmented into short independent frame sequences, i.e. clips. The movie structure hides the clip segments from the user by providing an iterator that plays through the whole movie. However, most research experiments involve single clip sequences. Currently a movie file is loaded as a single clip.

The standard interface for getting frames from the movie is through an iterator as illustrated in the following example:

```
#include <vidl/vidl_io.h>
#include <vidl/vidl_frame.h>
#include <vidl/vidl_movie.h>
//load the movie
vidl_movie_sptr  my_movie = vidl_io::load_movie("video.avi");

//loop through the frames
vidl_movie::frame_iterator fit(my_movie);
for (fit = my_movie->first(); fit != my_movie->last(); ++fit)
  {
     vil_image_view<vxl_byte> image = fit->get_view();
     // do something with the image
    ...
  }
```

Most users will operate in this way in accessing the frames of a movie. The `frame_iterator` allows the user to set forward and backward through the movie or to access a specific frame.

11.2 The Codec

Access to video file formats is provided through a standardized interface called the *codec*, which stands for coder/decoder. The key tasks of the codec are to load, save and probe

a video file. The probe detects if the format of a given file corresponds to any of the available codecs.

The codecs are automatically registered at compile time. There is no need to explicitly register codecs as there was in the past. All codecs that are compiled will be registered.

It is difficult to find portable codecs that are open source code for the popular video formats. We are working on an MPEG codec (the current implementation doesn't read the header information) that will work on both unix and windows. Two AVI codecs are also available. One uses native windows dlls for handling AVI files and compiles in windows. The other uses the open source avifile library and compiles in linux. Unfortunately, the avifile library also uses the windows dlls and therefore is only available on x86 machines.

11.3 File I/O

As we have already seen the `vidl` library supports reading and writing video files. These functions are handled by the class `vidl_io`. The key methods are illustrated by the following example:

```
#include <vidl/vidl_io.h>
#include <vidl/vidl_movie.h>
...
  vidl_movie_sptr avi_movie = vidl_io::load_movie("movie.avi");
  if (!avi_movie) {
    vgui_error_dialog("Failed to load movie");
    return;
  }
...

//later we save the avi movie as a set of images in the indicated directory

 if (!vidl_io::save(avi_movie, "image_directory" , "ImageList"))
  {
    vgui_error_dialog("Failed to save movie");
    return;
  }
...
```

11.4 Conclusion

The `vidl` interface is simple and hides most of the difficult aspects of dealing with video files. Getting MPEG to work at least in read mode across platforms is an important next step.

12 vcsl: Co-ordinate systems

Chapter summary: Metric entities, co-ordinate systems and transformations.

This non-core Level 2 library `vcsl` is intended to provide an environment for representing co-ordinate systems, transformations, dimensions, and metric units.
This includes classes for

- Radians, degrees, and meters.
- 2D and 3D Cartesian co-ordinate systems
- Various polar co-ordinate systems
- Various geographic co-ordinate systems.
- Various Transformations

12.1 Contents

12.1.1 Metric and Dimensional classes

- `vcsl_dimension` - Abstract dimension
  - `vcsl_length`
  - `vcsl_angle`
- `vcsl_unit` - Abstract unit associated to a dimension
  - `vcsl_length_unit`
    - `vcsl_meter` - This is the standard length unit
  - `vcsl_angle_unit`
    - `vcsl_radian` - This is the standard angular unit
    - `vcsl_degree`

12.1.2 Co-ordinate Systems and Transformations

- `vcsl_coordinate_system`
  - `vcsl_cartesian_2d`
  - `vcsl_cartesian_3d`
  - `vcsl_cylindrical`
  - `vcsl_polar`
  - `vcsl_spherical`
  - `vcsl_geographic`
    - `vcsl_geocentric` - Intended to represent a location relative to the earth
    - `vcsl_geodetic` - Intended to represent a location relative to the earth
    - `vcsl_lambertian` - Lambert Conformal Conic Projection
    - `vcsl_utm` - Universal Transverse Mercator projection
- `vcsl_spatial_transformation`
  - `vcsl_cylindrical_to_cartesian_3d`
  - `vcsl_perspective`
  - `vcsl_rotation` - Rotation about an axis through the origin
    - `vcsl_displacement` - Rotation about a general axis
  - `vcsl_scale`
  - `vcsl_translation`
  - `vcsl_composition` - Composition of transformations
- `vcsl_graph` - Represent multiple co-ordinate systems and transformation between them.

12.2 Examples

The following example shows how to use and build a set of co-ordinate systems.

```
#include <vcsl/vcsl_cartesian_3d.h>
#include <vcsl/vcsl_rotation.h>
#include <vcsl/vcsl_graph.h>
...
  //: Convenient indices for vectors representing points, etc.
  enum AXES { X, Y, Z, T };

  //: Graph of defined CS (Co-ordinate system)
  // All CS must be added to this graph as they are created.
  vcsl_graph_sptr graphCS = new vcsl_graph;

  //: World Coordinate System
  // Equivalent to the Top view, by definition.
  //
  // A CS for which no parent is defined is absolute.  Logically,
  // there must be one absolute CS which all other directly or
  // indirectly relate to.
  vcsl_spatial_sptr WCS = new vcsl_cartesian_3d;

  // Add WCS to the global map of coordinate systems.
  WCS->set_graph(graphCS);

  // New CS can be defined relative to any point (translation) or
  // axis (rotation) in the parent CS.  But it is so common to do
  // so relative to the X, Y, or Z axes, that it is convenient to
  // define the corresponding vectors:

  //: x-axis vector
  vnl_vector_fixed <double, 3> xA;
  xA[X]=1; xA[Y] = 0; xA[Z] = 0;

  //: y-axis vector
  vnl_vector_fixed <double, 3> yA;
  yA[X]=0; yA[Y] = 1; yA[Z] = 0;

  //: z-axis vector
  vnl_vector_fixed <double, 3> zA;
  zA[X]=0; zA[Y] = 0; zA[Z] = 1;

  //: 90 degree rotation about WCS y-axis
  // Transforms from WCS to right CS
  vcsl_rotation rightXF;
  rightXF.set_static(vnl_math::pi_over_2, yA);

  //: WCS rotated 90 degrees about the y-axis to produce right hand view/CS
  vcsl_spatial_sptr right = new vcsl_cartesian_3d;
  right->set_graph(graphCS);
  right->set_unique(WCS, &rightXF);

  //: Corner of a box with opposite corner at origin of WCS.
  vnl_vector_fixed <double, 3> corner;
```

```
corner[X] = 1; corner[Y] = 2; corner[Z] = 3;

// By inspection, corner should be (-3,2,1) in 'right' CS
vnl_vector<double> cornerXF = WCS->from_local_to_cs(corner, right, 0);

vcl_cout << cornerXF.x() << ", " << cornerXF.y() << ", " << cornerXF.z() << '\n';
```

12.3 Further Work

1. Write conversion between classic coordinate systems in the same than the class vcsl˙cylindrical˙to˙cartesian˙3d (singleton pattern)
   - `vcsl_spherical_to_cartesian_3d`
   - `vcsl_polar_to_cartesian_2d`
   - `vcsl_geocentric_to_cartesian_3d`
   - `vcsl_geodetic_to_cartesian_3d`
   - `vcsl_lambertian_to_cartesian_3d`
   - `vcsl_utm_to_cartesian_3d`
2. Add other units and dimensions
3. In `vcsl_spatial_transformation`, add

```
//: May 'this' have a matrix representation ?
virtual bool is_linear(void) const=0;

//: Homogeneous matrix of 'this' at time 'time'
// REQUIRE: is_valid()
// REQUIRE: is_linear()
virtual vnl_matrix<double> matrix_value(const double time, bool type) const;
```

Appendix A Filenames and structure

A.1 Basic structure

The directory structure of vxl is below

```
$VXLSRC/vcl
        core/vnl
            /vul
            /vbl
            /vil
            /vgl
            /vsl
           ...
        v3p/netlib
           /jpeg
           ...
        config/cmake
        contrib/
```

'`vcl`' is the C++ compatibility layer (see ⟨undefined⟩ [vcl], page ⟨undefined⟩).

'`core`' contains the core vxl libraries.

'`v3p`' (v-third-party) contains code that is available elsewhere and is used in implementing vxl. `v3p/netlib` contains cleaned-up C versions of the high-quality Fortran code. `v3p/jpeg` and others contain image file format code used by vil. In theory, you don't need any of of the v3p libraries if you have locally installed versions. In practice, you need `v3p/netlib` but not the others. the

'`config/cmake`' contains the configuration files used by the CMake build system (see ⟨undefined⟩ [CMake], page ⟨undefined⟩).

A.2 Other packages

On the repository and some vxl distributions, you will find several other packages in the '`contrib`' directory.

'`gel`' Published libraries from General Electric Global Research.

'`oxl`' Published libraries from the Robots Research Group at Oxford University.

'`mul`' Published libraries from Imaging Science and Biomedical Engineering at the University of Manchester.

'`rpl`' Published libraries from Rensselaer Polytechnic Institute

'`brl`' Published libraries from Brown University

'`tbl`' A set of image processing libraries.

'`oul`' Published libraries from Otago University

'`conversions`' A set of routines for converting between various vxl and non-vxl types.

These modules are presented as useful code, with even less guarantees than vxl (which is not to suggest that we guarantee anything about vxl). In particular they may not work as advertised. They may not be documented. If you find them useful, and think they deserve full vxl-status please contact the library's author to see about having the code tidied up and promoted to vxl.

You can find any overview documentation about these modules from the vxl homepage (`http://vxl.sourceforge.net/`) by following the "documentation" link.

A.3 CVS Repository

The VXL repository is provided by SourceForge—Thanks guys! The instructions for downloading vxl are documented at `http://vxl.sourceforge.net/#download`.

Appendix B Build Systems

After building VXL, you will want to write some code which links against your shiny new libraries. The details of this process vary depending on how you are building your own code. In general, as well as all the system standard libraries, you will need to tell your compiler and linker the following:

- For the include directories, you will need to use `$VXLSRC/vcl`, `$VXLSRC/core`, `$VXLBIN/vcl` and `$VXLBIN/core`. If you are using one of the contributed non-core libraries, e.g. the spatial objects library `vsol` in `contrib/gel`, then you need to add the relevant contrib directory, e.g. `$VXLSRC/contrib/gel`.
- With the default build options in VXL, all the libraries are placed in `$VXLBIN/lib`.
- You obviously need to decide which libraries to link against. As explained before, VXL is a collection of libraries. You will almost certainly have to link against `vcl`. You should also link against any libraries you explicitly use. e.g. if your code has a `#include <vnl/vector.h>` then link against `vnl`. Any missing libraries can be added as the linker complains they are missing. VXL's simple naming scheme should make it very easy to identify the missing library from the error message.
- You may also need to link against some extra system libraries, e.g. on Unix, VXL can use libm (for math stuff), libpng, libjpeg, libtiff (for image loading), libnsl (for Solaris sockets.) On Windows you may need to link against ws2˙32.dll (for sockets stuff.)

B.1 CMake

If keeping track of include directories, etc. is too much hassle, or if you are building a project to run on multiple platforms, then we suggest using the CMake system to build your project. CMake automatically communicates the necessary settings between VXL and your project. If you are unable to get your non-CMake Makefile of project file to work properly, then please try CMake — it will likely solve your problem for you.

VXL uses the CMake configuration system to generate Makefiles (for GNU Make or NMake) and MSVS Project files. There are other build systems as well, but most users use CMake. This section shows by example how to write '`CMakeLists.txt`' CMake files for VXL. The intent here is to show how to write these configuration files and to give detailed reasons for the various methods used in these files in VXL. This section should be used in conjunction with the CMake User's Manual (`http://www.cmake.org`).

CMake is available from `http://www.cmake.org` where you'll find pre-compiled versions and instructions for getting the source code from an anonymous CVS server. VXL is designed to be configured by the latest released version of CMake. At the time of writing, this is version 1.8.1. When obtaining CMake via CVS, you can use the tag `LatestRelease`. Cmake documentation can be found at `http://www.cmake.org/HTML/Documentation.html`.

You can choose not to build certain portions of the VXL source tree by setting options in CMake.

Examples given below are for simple applications and libraries. Guidelines for the top level VXL '`CMakeLists.txt`' file and intermediate '`CMakeLists.txt`' files (which include many `SUBDIRS` commands) are not given here.

There are many good examples of '`CMakeLists.txt`' files in the VXL source, but they are often in a state of flux. The intention is for this document to show how the '`CMakeLists.txt`' files should be constructed and to be verbose about the reasons, and to also be a style guide for these files.

It is recommended that the build directory be separate from the source directory.

B.1.1 Example 1: An application that uses VXL libraries

A single, complete '`CMakeLists.txt`' file follows, for a fictional application named foobar. The file is mostly comments, which are prefixed by '#' in CMake.

```
# This sample CMakeLists.txt file will configure an application named
# foobar that is built from foo.cxx, bar.cxx and some header files.

# The ADD_EXECUTABLE command configures a new executable.  There may
# be any number of these commands in the file.  The first argument to
# ADD_EXECUTABLE is the name of the executable file (without the
# possible .exe suffix) that you want to generate.  The rest of the
# arguments, up to the closing ")", are file names, where spaces,
# comments and newlines may be used at will.

# Always give the filename suffix for source files in the
# ADD_EXECUTABLE and ADD_LIBRARY commands.  CMake will try to guess
# the suffix if none is given, but this can cause problems when object
# files, or other files with the same name but different suffix, are
# present in the source directory.

# In addition to the C and C++ source files, all header files are
# included in the ADD_EXECUTABLE command so that they become part of
# the project in MS Visual Studio.  List all header files for this
# application, but not headers from other libraries you are using.
# The reason for this is that users of a GUI build system like MSVS
# will then see every source file in the file list, and can click on
# them to view and possibly edit them.

# Every source file (except for template instantiation files residing
# in Templates subdirectories) should be present in exactly one build
# rule, i.e., in either an ADD_EXECUTABLE or an ADD_LIBRARY rule in
# exactly one CMakeLists.txt file (normally the one in the same
# directory, but sometimes it could be the CMakeLists.txt file of the
# parent directory, cf. core/vil/file_formats).  Template instantiation
# files residing in Templates subdirectories are not included
# specifically in any CMakeLists.txt file, but are included via the
# AUX_SOURCE_DIRECTORY CMake command.  The AUX_SOURCE_DIRECTORY
# command is demonstrated in Example 2, in which a library is built,
# but can also be used for executables.

# Finally, we get to an actual CMake command.

ADD_EXECUTABLE( foobar
  defines.h
  foo.cxx foo.h
  bar.cxx bar.h
)

# If this executable is using libraries other than vcl, then an
# INCLUDE_DIRECTORIES is needed so that include files will be found
# (vcl is automatically included via the top level VXL CMakeLists.txt
# file).  This line is needed to include files from the core vxl and
# gel packages.  Most executables will, of course, need to include the
# core vxl sections.
```

```
INCLUDE_DIRECTORIES(
  ${VXLCORE_INCLUDE_DIR}
  ${GEL_INCLUDE_DIR}
)

# For INCLUDE_DIRECTORIES, the preferred practice is to specify a
# directory exactly one level below ${allvxl_SOURCE_DIR} as an include
# directory, and then the further qualify include files in the source
# code with commands like #include <vsol/vsol_box_2d.h>.  Do not do
# any of the following.

# NO # INCLUDE_DIRECTORIES( ${vxl_SOURCE_DIR} )
# NO # INCLUDE_DIRECTORIES( ${GEL_INCLUDE_DIR}/vsol )
# NO # INCLUDE_DIRECTORIES( ${vxl_SOURCE_DIR}/contrib/gel/vsol )

# Indicate which VXL libraries foobar depends on with the
# TARGET_LINK_LIBRARIES command.  Include each library that is
# directly used by this application.  Do not include libraries that
# are subsequently used by the libraries that are directly used.
# CMake will automatically handle the tree of dependencies so the link
# command will have all required libraries.

# The libraries can be listed in any order.  However, if you know that
# library "A" depends on library "B", then it is best to list library
# "A" before library "B".  Doing so helps to reduce the number of
# duplicate library listings in the link command generated by CMake.

TARGET_LINK_LIBRARIES( foobar vnl_algo vul vil vcl )

# Before CMake 1.4, LINK_LIBRARIES commands were used to add
# libraries.  The LINK_LIBRARIES command should no longer be used in
# VXL in ordinary circumstances.
```

B.1.2 Example 2: A library that uses VXL libraries

```
# This CMakeLists.txt file will configure a library named vfbl that is
# built from vfbl_foo.cxx, vfbl_bar.cxx, possibly vfbl_zip.cxx
# (optional, used if zlib is available) and several header files.

# This fictional vfbl library can optionally use zlib.  We will find
# out whether zlib is available by including this module file, and use
# it if it is.

INCLUDE( ${MODULE_PATH}/FindZLIB.cmake )

# For this library the list of source files will be built up with SET
# commands.  Before CMake 1.4 the SOURCE_FILES command was used for
# this purpose, but SOURCE_FILES is now deprecated and the SET command
# is preferred.  The SET command can be used to create source file
# lists for use with both the ADD_EXECUTABLE and ADD_LIBRARY commands.
# This is necessary when certain sources are conditionally included,
# as we see here, and preferred for long source file lists.
```

```
# The SET commands for a source list variable must come before the
# ADD_LIBRARY or ADD_EXECUTABLE command that uses them.  Because we
# are using SET and not SOURCE_FILES, the source list variable will be
# referenced with "${vfbl_sources}" as opposed to simply
# "vfbl_sources".  The ability to use ${vfbl_sources} to reference
# source lists is new in CMake 1.4, and is now preferred.

SET( vfbl_sources
  vfbl_defines.h
  vfbl_foo.cxx vfbl_foo.h
  vfbl_bar.cxx vfbl_bar.h
  vfbl_baz.txx vfbl_baz.h
)

# Use AUX_SOURCE_DIRECTORY when there are .cxx files in a Templates
# directory that instantiate templates in .txx files.  This command
# causes each file in the subdirectory Templates to be added to a list
# of source files.

AUX_SOURCE_DIRECTORY( Templates vfbl_sources )

# Include the source file vfbl_zip.cxx when building the library only
# if ZLIB is available.  Notice how vfbl_sources is set to its current
# value, plus the new source file "vfbl_zip.cxx".

IF( ZLIB_FOUND )
  SET( vfbl_sources ${vfbl_sources} vfbl_zip.cxx )
ENDIF( ZLIB_FOUND )

# The ADD_LIBRARY command configures a new library.  The first
# argument is again the target name, i.e., the name of the library to
# be created, but without any possible suffix such as ".lib",
# ".a" or ".so" and without the possible prefix "lib" (used in Unix).
# The other arguments are either file names, or (as here) a variable
# holding a list of files, specified before the ADD_LIBRARY command
# with one or more SET commands.

ADD_LIBRARY( vfbl ${vfbl_sources} )

# In general, include FindXXX.cmake to determine whether package XXX
# is available.  In this example, XXX is ZLIB.  VXL has its own
# specialised version of some of the FindXXX.cmake Module files
# instead of the ones that come with the current version of CMake.
# These specialised versions are there to deal with v3p, or copies of the
# files in the CMake CVS snapshot, but not available in the current
# release of CMake.

# The variable XXX_FOUND is set to "YES" if the package is available
# and undefined or "NO" if it is not.  If the package is available,
# XXX_INCLUDE_DIR and XXX_LIBRARIES variables are set, and you must
# use them in INCLUDE_DIRECTORIES and TARGET_LINK_LIBRARIES commands
# as shown below.
```

```
IF( ZLIB_FOUND )
  INCLUDE_DIRECTORIES( ${ZLIB_INCLUDE_DIR} )
  ADD_DEFINITIONS( ${ZLIB_DEFINITIONS})
  TARGET_LINK_LIBRARIES( vfbl ${ZLIB_LIBRARIES} )
ENDIF( ZLIB_FOUND )

# As in Example 1, we will configure to include VXL header files.

INCLUDE_DIRECTORIES( ${VXLCORE_INCLUDE_DIR} )

# Just as in Example 1, use the TARGET_LINK_LIBRARIES command to
# specify each library that is directly used by this library.  CMake
# will remember that vfbl needs these libraries.  When an application
# links with vfbl, CMake will automatically make sure it is linked
# with each of these libraries.  This linking is recursive, so that
# for example, the vcl in the line below is unnecessary - using vul
# and vil will ensure that vcl is used.  However, it is still a good
# practice to specify vcl here is the library directly uses vcl.

TARGET_LINK_LIBRARIES( vfbl vnl_algo vul vil vcl )

# The LINK_LIBRARIES command should not be used when configuring a
# library.  This would not propagate library dependencies to
# applications that use this library.

# Tests go in the tests subdirectory.  Whether the tests are built
# depends on BUILD_TESTING.

IF( BUILD_TESTING )
  SUBDIRS( tests )
ENDIF( BUILD_TESTING )

# Same goes for examples.

IF( BUILD_EXAMPLES )
  SUBDIRS( examples )
ENDIF( BUILD_EXAMPLES )
```

B.1.3 Example 3: Building outside of the VXL tree

It is a good practice to keep your local source code which uses VXL, but will not become part of VXL, outside of the VXL source tree. CMake offers support for the configuration of a local build tree that will use the VXL build tree. The CMake commands below, used at the beginning of the top-level '`CMakeLists.txt`' in your local source code tree will make it easy for your tree to use VXL.

```
# Your local source tree must have its own project name.  This project
# will be a "client" project of VXL.  It will import configuration
# information and use parts of VXL.

PROJECT( foobar )

# The CMake command FIND_PACKAGE(VXL) attempts to find VXL binary
# installation.  CMake will look in the directory specified by the
```

```
# CMake variable VXL_DIR.  Normally, CMake will initially not find
# VXL, will warn you that it could not find VXL, and then give you the
# chance to set the variable VXL_DIR and reconfigure.  VXL_DIR now
# replaces VXL_BINARY_PATH.

FIND_PACKAGE(VXL)

# Whether FIND_PACKAGE(VXL) worked is stored in the variable
# VXL_FOUND.  If VXL was found we then include 'UseVXL.cmake'
# which will set many variables prefixed with VXL_ that your project
# can use to determine what parts of VXL are present and how to use
# them.  'UseVXL.cmake' holds all the necessary definitions that
# CMake needs to use VXL.

IF(VXL_FOUND)
  INCLUDE(${VXL_CMAKE_DIR}/UseVXL.cmake)
ENDIF(VXL_FOUND)

# 'UseVXL.cmake' will in turn include the file
# 'VXLConfig.cmake' from the top of the VXL build tree.  You
# should look at this file to see what CMake variables are imported.

# To set VXL_DIR, you could rely on an environment variable:
#   SET( VXL_DIR $ENV{VXLBIN} )
# However, you must then make sure that VXLBIN is set correctly every
# time that cmake is run.  This is often a source of much frustration
# and is not recommended.
#
# Another option is to set the path directly:
#   SET( VXL_DIR /usr/local/vxl-1.0-beta2/bin )
# This has the drawback that the CMakeLists.txt file (this file) has
# to be modified to suit your particular setup, which may cause
# trouble.  If this file is under CVS control, for example, you may
# inadvertently commit it with your local setup hard-coded, thus
# playing havoc with other users.

# The rest of this CMakeLists.txt file could contain commands as seen
# in the previous examples, or might just be SUBDIRS commands.

# Remember that all CMake variables defining the VXL configuration
# begin with "VXL_".  If you want to use VGUI in a client project you
# should use something like this.

IF( VXL_VGUI_FOUND )
  INCLUDE_DIRECTORIES( ${VXL_VGUI_INCLUDE_DIR} )
  SET( foo_sources ${foo_sources}
    foo_gui.cxx foo_gui.h
  )
  TARGET_LINK_LIBRARIES( foo ${VXL_VGUI_LIBRARIES} )
ENDIF( VXL_VGUI_FOUND )

# If you want to directly use the mpeg2dec library that VXL chose to
# use, do something like this.  You will link either to a system
# mpeg2dec library that VXL used or the one VXL built for itself and
```

```
# used.

IF( VXL_MPEG2_FOUND )
  INCLUDE_DIRECTORIES( ${VXL_MPEG2_INCLUDE_DIR} )
  SET( foo_sources ${foo_sources}
    foo_video.cxx foo_video.h
  )
  TARGET_LINK_LIBRARIES( foo ${VXL_MPEG2_LIBRARIES} )
ENDIF( VXL_MPEG2_FOUND )

# In the file 'UseVXL.cmake' in the VXL build directory you will
# see the xxx_FOUND, xxx_INCLUDE_DIR, and xxx_LIBRARIES variables set
# for all optional portions of VXL and all 3rd party libraries used by
# VXL.
```

In projects external to VXL you should not use the '`UseVGUI.cmake`', '`FindMPEG2.cmake`' or similar files in the VXL CMake Modules directory. These files are for VXL internal use only. Rely on the variable settings from '`UseVXL.cmake`'.

B.1.4 CMake Pearls

You can tell CMake to be a bit more picky and to warn about the use of deprecated syntax with the following command.

```
CMAKE_MINIMUM_REQUIRED( VERSION 1.6 )
```

If your programs or libraries are to use libraries that are outside of the VXL tree and outside of the normal system locations, you must tell CMake where to find the include files and libraries with commands such as these.

```
INCLUDE_DIRECTORIES( /usr/local/include/libgimp )
LINK_DIRECTORIES( /usr/local/lib/gimp )
```

If your are building in a tree outside of the VXL tree, as in Example 3, and want libraries installed to a specific directory, you can use one of these commands. Here "foobar" is the name of the project (see Example 3).

```
SET( LIBRARY_OUTPUT_PATH ${foobar_BINARY_DIR}/lib )
```

or

```
SET( LIBRARY_OUTPUT_PATH /usr/local/lib )
```

B.1.5 CMake Variables and Preprocessor Macros

Listed below are CMake variables commonly used in VXL CMake files and similar preprocessor macros. The CMake variable are prefixed by "CMake:" and the equivalent (usually identically named) C/C++ preprocessor macros (for `#ifdef`, etc.) are prefixed by "Preprocessor:". The means by which each variable is set is given in parenthesis.

To determine the compiler being used at compile time, use macros defined in '`vcl/vcl_compiler.h`'. The '`FindXXX.cmake`' files are in '`$VXLSRC/config/cmake/Modules/`'.

- CMake: `WIN32` (defined by CMake, not in a file)
- Preprocessor: `VCL_WIN32` ('`vcl_compiler.h`', included by each '`vcl_*.h`' file)
  The Windows API is available. Keep in mind that `WIN32` is defined under Cygwin, so it does not imply use of MSVC++. Make sure you don't really want `HAS_MFC` or `VCL_VC` (or `VCL_VC_6` or `VCL_VC_DOTNET`) instead.
- CMake: `UNIX` (defined by CMake, not in a file)
- Preprocessor: system dependent
  Some kind of Unix API is present. The CMake variable `UNIX` is also is defined under Cygwin.

- CMake: `CYGWIN` (defined by CMake, not in a file)
- Preprocessor: `__CYGWIN__` (gcc under Cygwin)
  The Cygwin API is present.
- CMake: `BORLAND` (defined by CMake, not in a file)
- Preprocessor: [???]
  The Borland C++ compiler is being used.
- CMake: `SOLARIS` (VXL top level '`UseVXL.cmake`' file)
- Preprocessor: [???]
  The OS is Solaris. This variable is set in the top level '`CMakeLists.txt`' file.
- CMake: `MODULE_PATH` (VXL top level '`CMakeLists.txt`' file)
- Preprocessor: (none)
  Directory holding the VXL versions of the '`FindXXX.cmake`' files.
- CMake: `MFC_FOUND` (Module file '`FindMFC.cmake`')
- Preprocessor: `HAS_MFC` (Occasionally defined where needed)
  Microsoft Foundation Classes are available.
- CMake: `GLUT_FOUND` (Module file '`FindGLUT.cmake`')
- Preprocessor: `HAS_GLUT` (Occasionally defined where needed)
  The GLUT library is available.
- CMake: `GTK_FOUND` (Module file '`FindGTK.cmake`')
- Preprocessor: `HAS_GTK` (Occasionally defined where needed)
  The GTK library is available.
- CMake: `JPEG_FOUND` (Module file '`FindJPEG.cmake`')
- Preprocessor: `HAS_JPEG` (Occasionally defined where needed)
  The JPEG library is available.
- CMake: `MPEG_FOUND` (Module file '`FindMPEG.cmake`')
- Preprocessor: `HAS_MPEG` (Occasionally defined where needed)
  The MPEG library is available.
- CMake: `NETLIB_FOUND` (Module file '`FindNetlib.cmake`')
- Preprocessor: none (could easily be added if and where needed)
  The NetLib library is available.
- CMake: `OPENGL_FOUND` (Module file '`FindOpenGL.cmake`')
- Preprocessor: `HAS_OPENGL` (Occasionally defined where needed)
  The OpenGL library is available.
- CMake: `PNG_FOUND` (Module file '`FindPNG.cmake`')
- Preprocessor: `HAS_PNG` (Occasionally defined where needed)
  The PNG library is available.
- CMake: `QT_FOUND` (Module file '`FindQt.cmake`')
- Preprocessor: none (could easily be added if and where needed)
  The Qt library is available.
- CMake: `QV_FOUND` (Module file '`FindQv.cmake`')
- Preprocessor: `HAS_QV` (Occasionally defined where needed)
  The Qv library is available.
- CMake: `TIFF_FOUND` (Module file '`FindTIFF.cmake`')
- Preprocessor: `HAS_TIFF` (Occasionally defined where needed)
  The TIFF library is available.
- CMake: `X11_FOUND` (Module file '`FindX11.cmake`')
- Preprocessor: `HAS_X11` (Occasionally defined where needed)
  The X11 library is available.
- CMake: `ZLIB_FOUND` (Module file '`FindZLIB.cmake`')
- Preprocessor: `HAS_ZLIB` (Occasionally defined where needed)
  The ZLIB library is available.

B.1.6 Historical CMake Include Files

VXL used to have '`CMakeListsLink.txt`' include files to handle library dependencies. The library dependency problem is now dealt with by CMake (since version 1.4) and the use of '`CMakeListsLink.txt`' files is deprecated.

In the past, VXL used '`CMakeListsHeaders.txt`' include files that were only included for Win32 builds, but this is no longer needed and is deprecated. The reason for having a '`CMakeListsHeaders.txt`' file was that older versions of CMake did not allow .h or .txx files in a build rule (except with MSVS), but this has been fixed now (CMake versions 1.2 and later). It is better to have everything in a single file, since that makes the whole system more manageable.

New VXL source code directories for executables and libraries will need a '`CMakeLists.txt`' file, but not a '`CMakeListsLink.txt`' and not a '`CMakeListsHeaders.txt`' file.

B.2 Other Build Systems for Building VXL.

Your copy of VXL may contain control files for other build systems such as make or Developer Studio. Copies of these control files (e.g. makefiles or .dsp files) do exist in the repository, although they may not be quite as up to date. They are tucked away in the `build-makefiles` and `build-dsps` branches of the repository.

B.3 Frequently Asked Questions

Question 1

> I have successfully built VXL under MS Visual Studio, and am now trying to build a simple program that uses the libraries. The Linker complains that there are multiple definitions (error `LNK2005`) of several things from the C++ Standard Library, or locally defined symbols being imported (error `LNK4049`). What is going wrong?

These kinds of errors often indicate that you are linking against different C++ run-time libraries with which you compiled VXL (release vs debug, static vs DLL stdlib, etc). The flags like `/MP` and `/MD` much match exactly, in VXL's build and your program's build. These flags are set in Visual Studio under Project->Properties (or Settings under MSVC6.0)->C/C++->Code Generation->Use run-time library. You should set your project to (Debug) Multithreaded DLL to match the default settings for VXL.

The easiest way to link against VXL, we've found, is to use CMake for your project too. CMake will then make sure the flags match, or else will give you an error or warning.

Question 2

> I followed the VXL installation documentation, but found that some of the entries in the '`CMakeCache.txt`' file say that CMake could not find programs, paths, etc. Is this a problem?

No. VXL needs very little to build correctly (only a C/C++ compiler at worst), however it can make use of various system provided libraries, rather than build its own versions. CMake also looks for lots of system tools so that it can understand the environment it is in. So having lots of `NOTFOUND` entries is quite normal. There is no need to worry unless CMake displays a warning or error message while it runs.

Question 3

> I used CMake to create a MSVC project with `BUILD_SHARED_LIBRARY=ON`. But when I try to build, it gives message `"cannot open vcl.lib"`. What's wrong?

The CMake "Shared Library" feature doesn't work with MSVC. You have to use static libraries. This is mostly due to MSVC's requirements to have either a complete list of every exportable identifier or a decoration of every identifier in the code. We are too lazy (or appalled at this "feature") to try to fix this. If you absolutely need DLLs you can build your code using Cygwin, which can produce shared libraries in a "normal" manner. Alternatively, if you only want to export a small number of classes or functions, then you can manually list them. See the MSVC tool documentation for further details.

Appendix C Coding Standards

C.1 Coding Standards

As has been mentioned before, it is VXL's philosophy that any rules should be pragmatic. The following is a list of standards that have either been explicitly agreed, or implicitly observed, and are here to make our lives easier. If they are making life harder, then we will change them.

Of course, these standards only apply to code in the VXL repository. Although some of they may be useful, there is absolutely no need for users of VXL to follow them at all.

- See ⟨undefined⟩ [VXL Philosophy], page ⟨undefined⟩.
- See ⟨undefined⟩ [VCL Requirements], page ⟨undefined⟩.
- See ⟨undefined⟩ [CMake Build System], page ⟨undefined⟩.
- All code should build, preferably warning-free, on the platforms on our dashboard. This is probably the single most important coding standard. If your commit breaks an existing build, you are responsible for fixing it as quickly as possible. If you don't understand the error, contact the relevant platform's maintainer for help.
- The code should have good coverage with regression testing, and no reported memory leaks, etc.
- Anyone who is modifying code should keep a watch on the `vxl-maintainers` email list.

C.2 Style Guide

The following vxl coding standards were originally agreed at the Zurich meeting in March 2001. They are mandatory for libraries in the core module. Even if you are not writing code which you expect to enter vxl/core, then it is still worth following these coding standards, as they have been designed to minimise confusing and help avoid potential problems.

- All global scope identifiers are lowercase with underscores separating words. (Rejected `vxl_Image_template` by vote Zurich, March 2001.)
- Each class, file, function and member variable should have a one line Doxygen description (brief line) in addition to the optional long description. See `core/doc/vxl_doc_rules.*` for further information. Each library should have a file in its directory called `introduction_doxy.txt`, containing at least the following:

```
/*! \mainpage wibble: Succinct Description of wibble.
*/
```

- Private member variables should have a underscore as a suffix, e.g. `value_`. (An underscore as a prefix `_value` is not allowed by the C++ standard. Rejected by vote: prefix "m˙", e.g. `m_value`.)
- Accessor functions for a variable `x_` should be called `x()` and `set_x()`. In particular

```
T x;
...
void set_x(T x); // or const T&, or whatever
T x();
```

  (Using `get_x()` is redundant and simply causes extra typing.) `set_x()` should return void. (It should never return the old value, as this can be highly confusing.)

```
void x(T val)  x_ = val;   // WRONG!
```

- Any forward declarations in a library `val` should be consolidated in `val_fwd.h` at the top level of the library. However, there is no requirement to provide forward declarations.

- A file may have more than one class, provided they "belong". The goal of the filename=class name is to make it easy to find class/function declarations and definitions. As long as this holds, a file may declare more than one class.
- One tab (0x08) *always* expands to 8 spaces
- Indentation:
  - may use any number of indent spaces (>=2), but must be consistent.
  - we rejected proposal to fix the number of indent spaces
  - may only use spaces for indentation (i.e. no tab characters, which means 10 spaces of indentation may *not* be replaced by one tab and two spaces). (Reason: Visual Studio (and other editors) allow the developer to set tab=n. Most people have done this to their own taste. These environments convert TAB to n spaces, instead of 8.)
  - When editing other people's sources, you must use the same indentation as they have.
- Maximum number of characters per line: 132. (Number extracted primarily from vtk guidelines). 80 character lines preferred when possible.
- Source file endings:
  - C source files should end `.c`
  - C++ ordinary source files should end in `.cxx`
  - Header files should end in `.h`
  - C++ Template implementation code should end in `.txx`

  (These choices minimise problems and maximise sensible default behaviour over many compilers. Agreed by consensus on vxl-maintainers May 2002.)

C.3 How to contribute a new class or library

If you have computer vision code that fits neatly into VXL platform, or a VXL-friendly interface to an external library, the VXL community would welcome your contribution.

For a single class, or few functions, it is worth seeing which existing VXL library your code might neatly fit into. Whole libraries should be added to a subdirectory of the contrib directory. See ⟨undefined⟩ [`vxl/contrib`], page ⟨undefined⟩. Get yourself an account on SourceForge, and we'll give you commit rights.

If you have already developed your code in a CVS repository, we can arrange for your RCS files to get added directly to the repository.

It isn't expected that your code meet the Style Guide or Coding Standards on its first commit to the repository. But you will have to put some effort into making your code build cleanly on the dashboard as soon as possible. You should aim for your code to meet the coding standards and style guide over time.

C.4 How to elevate a library to the core.

- All the "should"s in the above standards, will need to be treated as "must"s, unless there is a good reason.
- A second institution or company, must be happily using your library. (It is unclear whether that second institution needs to be a well known member of the VXL community.) The second institution may well be the one pushing for the library's elevation to the core.
- There must be a texinfo chapter for the VXL book, that provides a beginners introduction and tutorial to the library.
- There must be a designated owner for the library, who is responsible for co-ordinating changes to that library.

C.5 On the Use of Templates

VXL makes heavy use of the C++ template mechanism. In order to ensure efficient compilation and linking a scheme has been adopted in which the declaration of each templated class is placed in a `.h` file, and its definition (implementation) is separated into a `.txx` file. The latter is only read a small number of times, when a particular instantiation of a template is created.

Each specialisation of the templated class must be compiled explicitly. This is usually done by creating an explicit instantiation in a file in the Templates subdirectory.

For example, `vgl_point_2d<T>` has a header file, `vgl/vgl_point_2d.h`, and an implementation in `vgl/vgl_point_2d.txx`. At the end of the latter is a macro which can be used to explicitly instantiation a particular specialisation of `vgl_point_2d`, together with all the associated templated functions:

```
File: vgl/vgl_point_2d.txx:
...
#define VGL_POINT_2D_INSTANTIATE(T) \ template class vgl_point_2d<T >; \
template vcl_ostream& operator<<(vcl_ostream&, const vgl_point_2d<T >&); \
template vcl_istream& operator>>(vcl_istream&, vgl_point_2d<T >&); \
```

To instantiate a a `vgl_point_2d<double>` there is a file in the a `vgl/Templates` directory:

```
File: vgl/Templates/vgl_point_2d+double-.cxx:

// Instantiation of vgl_point_2d<double>
#include <vgl/vgl_point_2d.txx>
VGL_POINT_2D_INSTANTIATE(double);
```

If you decided to use an unusual specialization, eg `vgl_point_2d<my_sponge>`, your code should compile cleanly, but would not link unless somewhere you explicitly instantiate the class and compile it somewhere suitable using a call:

```
VGL_POINT_2D_INSTANTIATE(my_sponge);
```

When you write a new templated class (or function), you should put the implementation in a suitable `.txx`, together with an appropriate `INSTANTIATE` macro. Study examples in existing classes for more details.

C.6 Frequently Asked Questions

Question 1

> Can I import a useful third-party library into `vxl/v3p`?

The general rule is that v3p is for third-party libraries that are need by VXL core libraries. Getting libraries in v3p working on all platforms is a difficult and tedious task, so we want to have no more than necessary.

Question 2

> I have a FindFoo.cmake module that is imports the very useful Foo library. Where can I put it?

You should submit it to the CMake project, where it will see wider use than just the VXL community. For very special cases, such as a new library needed by something in core, you can temporarily put it in `config\cmake\Modules\NewCMake`. However you should also submit it to CMake, and remove it from the VXL repository as soon as it becomes available in a CMake distribution.

Appendix D Adding to vnl˙algo

The strategy adopted for converting and wrapping the fortran files is a little involved. Some routines are simple to do, others very tricky. The general procedure is as follows. These steps are elaborated upon in the example below.

D.1 Overview

1. Use GAMS to find the module name, in SLATEC if possible, although CMLIB and TOMS routines are also public domain and good.
2. Convert the fortran to C using f2c
3. Add the routine to the Imakefile in the netlib library.
4. Encapsulate the routine in a class in vnl, after determining a suitable interface.
5. Read the module documentation and determine the calling sequence.
6. In the calling method, create all necessary workspace arrays and temporary variables that the call requires, call the external routine, and convert the results into the classes that VXL expects.
7. After the call, interpret the error code, and handle accordingly.

D.2 Problems

There are a few potential sources of difficulty, mostly in item 5, but in general I find that gritting one's teeth and guessing is a surprisingly good strategy. The main points to remember are:

1. All scalar variables are passed by reference. This means that you need to store all constants in variables and pass their addresses or declare the routines as accepting references. I do the latter for input variables, and the former for outputs.
2. Fortran arrays start from 1 rather than 0. This is actually a non-problem, as f2c generates code which interfaces zero-based to one-based arrays using the Numerical Recipes trick of decrementing the pointer, but is mentioned here for the benefit of fortran programmers.
3. Fortran arrays are stored column-wise rather than row-wise. Class `vnl_fortran_copy` provides an easy and efficient way to transpose matrices before calling.

In addition to these fortran specifics, it is important to be aware of the sorts of design patterns seen in numerical code. Many routines are coded for maximum generality and efficiency, which can make reading the descriptions heavy going. Common conventions are:

- An array is passed with three dimensions: number of rows in the physical array, number of rows to use for computation and number of columns. This allows the routines to be used on any submatrix of a larger matrix.
- Output results overwrite the input matrix.
- Output results are stored in some compact form, which must be decoded before use. Note however that in many cases routines are supplied to perform further computations using the encoded representation for time and space efficiency. The new QR class will demonstrate how to take advantage of this.
- The results of pivoting are generally returned in vectors of integers (say `ipvt`), where `ipvt[i]` is the position to which row/column `i` has been moved. These permutations which must be applied to the results in order to complete the process.

D.3 Example conversion – QR decomposition

Given the need for an algorithm that is not yet included in the vnl package, say a routine to compute the QR decomposition, your first stop is the GAMS decision tree. Class "D" is Linear Algebra, and class "D5" is QR decomposition. The SLATEC implementation is called DQRDC (Double precision QR DeComposition). Download the source, or obtain it from a local SLATEC distribution. Convert it to a C source file and a prototype file using

```
f2c -P dqrdc.f
```

and from the prototype file `dqrdc.P` we find that the function prototype is

```
int dqrdc_(doublereal *x, integer *ldx, integer *n, integer *p,
           doublereal *qraux, integer *jpvt, doublereal *work,
           integer *job);
```

At this point, the header of the fortran file `dqrdc.f` is examined in order to determine the meaning of the parameters. Considering parameter `X`, we find

```
X     DOUBLE PRECISION(LDX,P), where LDX .GE. N.
      X contains the matrix whose decomposition is to be computed.
```

This means that X is a LDX row by P column matrix, and that we require a decomposition of the first N rows. This is a common convention in fortran programs which allows computation on subblocks of matrices. In general, we will assume that we wish to work on the full matrix, and therefore that LDX = N. To create the required transformed copy of the matrix, use class `vnl_fortran_copy`:

```
vnl_fortran_copy Xtranspose(X);
```

Now, the function may be called as

```
int n = X.rows();
int p = X.columns();
vnl_vector<int> jpvt(p);
jpvt.fill(0); // Mark all columns as pivotable
vnl_vector<double> work(p);
int do_pivoting = 1;
vnl_vector<double> qraux(p);
dqrdc_(Xtranspose, &n, &n, &p,
       qraux.data_block(), jpvt.data_block(), w.data_block(),
       &do_pivoting);
```

Appendix E vsl Developer Topics

E.1 Binary IO Design Notes

The aim of the binary IO is to provide a consistent mechanism for saving and restoring all VXL objects to streams/files, in a cross platform manner.

E.1.1 Structure

The original intention was that ALL classes with data should implement `b_write(os)` and `b_read(is)` as member functions, and that the external `vsl_b_write(os,object);` would be implemented using the member function. The advantages of this scheme are that

- Member functions can save all the state of a class without making everything public
- Keeping the IO in the class makes it more likely that it will be kept up to date

Unfortunately a problem arises with templated containers, such as `vbl_array_2d<T>`. Creating a container of type T requires that `vsl_b_write(os,T)` is implemented, otherwise the `b_write` for the array will not compile. This means that adding such code to the library would break existing code, and force un-necessary implementation of reading/writing functions (even when creating arrays of objects which are unlikely ever to be saved, such as GUI Widgets).

A related problem is that to keep the core libraries independent, one would have to duplicate the basic binary IO of built in types in every library. When templating over a class outside the library, one might have to write a specialisation of the IO functions to get them to compile.

The simplest solution is to write the code as 'clip-on' libraries, as has been done.

However, for level-2 libraries, in general, neither of these problems will arise. In this case it is strongly recommended by the authors that IO be built into the class.

It is interesting to note that after vsl was written, and entirely independently as far as these authors are aware, a section on IO was added to the comp.lang.c++ FAQ. The recommendations in the FAQ match the design of vsl very closely, in the design of binary formats, the serialisation scheme, and the base-class loader scheme.

E.1.2 Format of Fundamental types

All values are stored in little endian format (as used by Intel and DEC alpha processors.) We had to choose one, and we mostly use Intel platforms.

E.1.2.1 Format of Integer types

The original code from which `vsl` was derived was cross platform only in the sense that it worked on either big-endian 32-bit platforms or little-endian 32-bit platforms. This was not an unreasonable assumption at the time it was written. However during the requirements capture for `vsl`, it became clear that being able to handle 64-bit fundamental types would be necessary. For example the long on an alpha-64 is 64 bits.

It is fundamentally impracticable to deal quietly and correctly with the problems this causes. In particular, a number greater than 4G can be represent by a 64-bit long. It can be successfully saved, but a platform with a 32-bit long simply cannot represent the value. Nevertheless it is necessary to deal with the problems in a predictable manner. There were several options.

1. Fix the size of the various types.
2. Record the size of the various types in a header at the start of the file, and then save values in native type.

3. Use an encoding scheme which works for any size.

The first option has the advantage of being the simplest (on the platform which matches the sizes chosen.) It has the disadvantage of either being twice as large as required on some platforms, and not able to represent the full range of values on others.

The second option, has the advantages of doing the correct thing on all platforms, and being very efficient when files are loaded and saved on the same platform. This is especially true if the endian-ness was also flagged in the header and values saved in native format. The disadvantage is that the code for reading files on each platform must know about every other possibility. So if there are n platforms supported, the numbers of options that need to be programmed is n^2.

The third option, suggested by Peter Vanroose, has the advantage that it does the correct thing on all platforms. Also, there is only one format, and so the amount of code to write only grows linearly with the number of platforms rather than quadratically as in option 2. The final advantage is that the store file will likely be smaller, since numbers are represented in as few bytes as possible. It has the disadvantage of some extra computation and memory overhead when both reading and writing.

The performance of the encoding and decoding required for option 3 was measured. On an Intel Pentium 3, using MSVC 6.0 in full optimisation mode, 25 million integers were encoded or decoded at once. The results do not include memory allocation overhead but does include some OS overhead.

Type	Range of values	Range of encoded size / bytes	Encoding time /clock cycles	Decoding time /clock cycles
unsigned	0 -> 127	1	18	24
int	-64 -> 63	1	19	28
unsigned	0 -> 4G	1,2,3,4,but mostly 5	37	95
int	-2G -> 2G	1,2,3,4,but mostly 5	47	105

This means that a 850 MHz PC can encode about 50 million small valued integers per second, and decode about 35 million. The decoding takes longer due to the need for testing for integer overflow.

We consider these speeds to be fast enough. and so option 3 was chosen.

The arbitrary length format stores the number in a series of bytes. Of each byte, 7 bits are used to represent the number and the most significant bit is used to flag the end of the number. The flag bit is 0 if there are more bytes left for this integer, and 1 if this is the last byte. The bytes are stored in little endian order, and the signed numbers are stored using 2s complement notation.

A consequence of this choice is that for example a number can be written from a short, and read into a long. We strongly advise against doing this, as it relies on the implementation details.

E.1.2.2 Format of Floating Point Types

Since almost all platforms use IEEE format floating point types to represent floats and doubles, we have used this format (in little endian order) to save them to disk.

A downside of this decision is that it is unclear how to store long doubles. The 80bit format that is native to Intel platforms would not appear to be sufficiently general. The best alternative would be to switch to variable length encoded floating point values. It might be easier just to design a 128 bit format. Until then I/O of long doubles is not supported. Please contact the designers if you need to add it.

E.1.3 Changing the IO format

The `vsl` system stores a IO schema version number at the start of the stream. This will enable a clean backwards compatible upgrade of the IO schema. It will not however make the rewrite of `vsl` trivial. Contact the designers if you wish to change the schema.

E.1.4 File Magic Numbers

If you use Unix, the following can be inserted in your /etc/magic file so that the `file` command will recognise `vsl` files.

```
#------------------------------------------------------------------------
# VXL: file(1) magic for VXL binary IO data files
#
# from Ian Scott <scottim@sf.net>
#
# VXL is a collection of C++ libraries for Computer Vision.
# See the vsl chapter in the VXL Book for more info
# http://www.isbe.man.ac.uk/public_vxl_doc/books/vxl/book.html
# http:/vxl.sf.net

2       lelong  0x472b2c4e      VXL binary file,
>0      leshort >0      schema version no %d
```

E.1.5 Serialisation

A common approach to performing serialisation, is to store the serial number in the shared object itself. This is the approach used by the Microsoft Foundation Classes, and by DEX (the IUE's IO and serialisation scheme.)

There are two disadvantages to this. The first is that you have to modify the object being serialised.

The second is that this scheme can get confused if a set of objects is being written to two streams. When do you clear the already saved flag for each object? What happens if the output to the two streams is being interleaved? Will a shared object get saved to one stream and not the other?

A different approach is used by `vsl`. Here the serialisation record is kept by the stream object. Pointers are used to uniquely identify each shared object within each computer, and a serial number is generated to be saved on the stream. Since there is now a record for each pair (stream, shared object) there will not be any clashes when saving to multiple streams. It also avoids having to modify the shared object.

E.1.6 Loading by base class pointer: Design Overview

When using polymorphism, there are frequently times when one needs to save and restore an object just using a base class pointer to it. `vsl` provides facilities to do this.

There are two cases to consider

- IO is provided by the class itself
- IO is provided by external 'clip-on' classes.

The former is the preferred method.

The latter is provided to allow binary IO for third-party libraries which cannot be modified. However, for technical reasons (see design notes below) it is also used to provide binary IO for the polymorphic hierarchies in the VXL core libraries.

E.1.6.1 IO provided within the class

```
#include <vsl/vsl_binary_loader.h>

class vxl_my_class : public vxl_baseclass
{
// ...
```

```
  virtual void b_write(vsl_b_ostream&) const;
  virtual void b_read(vsl_b_ostream&);
  virtual vcl_string is_a() const;
  virtual bool is_class(vcl_string const&) const;
  virtual vxl_baseclass* clone() const;
};

//: Provide examples of each type of polymorphic object that might appear in the stream.
void vxl_configure_loaders()
{
  vsl_add_to_binary_loader(vxl_my_class());
}

void demonstrate_save(vsl_b_ostream& os)
{
  vxl_baseclass *base_ptr = new vxl_my_class;

  // Write the object to the stream, together with
  // an identifier indicating what type it is.
  vsl_b_write(os,base_ptr);

  // Tidy up
  delete base_ptr;
}


void demonstrate_load(vsl_b_istream& is)
{
  vxl_baseclass *base_ptr

  vsl_b_read(is,base_ptr);

  // Show the object
  vcl_cout<<"Loaded object: "<<base_ptr<<vcl_endl;

  // Tidy up
  delete base_ptr;
}
```

E.1.6.2 How loading by baseclass pointer works (in-class case)

When an object is saved by baseclass pointer, using `vsl_b_write(os,const baseclass*)`, the name of the class is written first, then the object itself.

When one comes to load the object, using `vsl_b_read(is,baseclass* &)`, the following occurs:

1. The singleton loader object (`vsl_binary_loader<baseclass>`) is invoked.
2. The loader reads the name of the class from the stream.
3. The loader compares this name with a list of possible objects, which have been supplied by earlier calls to `vsl_add_to_loader(derived_class())`.
4. If the loader finds a match, it creates a clone of the named class object.
5. This clone loads the data from the stream.
6. The loader then sets the baseclass pointer to point to this new object.

So you now have a shiny pointer to the object. Note: The caller is then responsible for the object that the loader created.

The above methods also work if the pointer that was saved was NULL. The loader will detect this and set the baseclass pointer to zero.

E.1.6.3 IO provided by external 'clip-on' classes

```
class vxl_my_class : public vxl_baseclass
{
// ...
  // No IO here
};

//: Provide IO for vxl_my_class
class vxl_my_class : public vxl_baseclassIO
{
 public:
  //: Constructor
  vxl_my_classIO();

  //: Destructor
  virtual ~vxl_my_classIO();

  //: Create new object of type vxl_my_class on heap
  virtual vxl_baseclass* new_object() const;

  //: Write derived class to os using baseclass reference
  virtual void b_write_by_base(vsl_b_ostream& os, const vxl_baseclass& base) const;

  //: Write derived class to os using baseclass reference
  virtual void b_read_by_base(vsl_b_istream& is, vxl_baseclass& base) const;

  //: Copy this object onto the heap and return a pointer
  virtual vxl_baseclassIO* clone() const;

  //: Return name of class for which this object provides IO
  virtual vcl_string target_classname() const;

  //: Return true if b is of class target_classname()
  //  Typically this will just be "return b.is_class(target_classname())"
  //  However, third party libraries may use a different system
  virtual bool is_io_for(const vxl_baseclass& b) const;
};


//: Provide IO class for each type of polymorphic object that might appear in the stre
void vxl_configure_loaders()
{
  vsl_add_to_binary_loader(vxl_my_class_io());
}

// The actual IO calls are then identical to those described above for the
// case of the IO provided within the class
```

See below for details of the implementation of the clip-on IO libraries.

E.1.6.4 How loading by baseclass pointer works (external IO case)

Essentially this is similar to the within-class-IO case, in that a string is written to the stream to indicate which class is saved, and when reading the stream this is used to indicate what to generate on the heap. However, the details of the implementation are more complicated.

When an object is saved by baseclass pointer, using `vsl_b_write(os,const baseclass*)`, a function in `vsl_clipon_binary_loader<baseclass,baseclass_io>` is invoked. It contains a list of IO objects, each of which is queried (using the `io.is_io_for(const baseclass&)` function to see if it can deal with the given class. If so a name, `io.target_classname()`, is saved to the stream, then the IO object is used to save the target object.

When one comes to load the object, using `vsl_b_read(is,baseclass* &)`, the following occurs:

1. The singleton loader object (`vsl_clipon_binary_loader<baseclass,baseclass_io>`) is invoked.
2. The loader reads the name of the class from the stream.
3. The loader compares this name with a list of possible objects, which have been supplied by earlier calls to `vsl_add_to_loader(derived_class())`.
4. If the loader finds a match, it uses the IO object to create a new object on the heap
5. The io object then loads the data from the stream into the object on the heap
6. The loader then sets the baseclass pointer to point to this new object.

So you now have a shiny pointer to the object. Note: The caller is then responsible for the object that the loader created.

The above methods also work if the pointer that was saved was NULL. The loader will detect this and set the baseclass pointer to zero.

E.1.7 Future Work

1. Currently, a few error checks in the IO still result in an error message and `std::abort()`. Since IO code is error prone for reasons often beyond the control of the programmer, this behaviour should be replaced with a standard error message a setting of the stream's fail flag, and return.
2. Design a format for `long double` and add it `to vsl_binary_io.h`.
3. Many of the function and file names do not conform to VXL standards - that each function and class should be named after the file-stem that declares and defines it.

E.1.8 Credits

Your principle designers for this chapter have been Tim Cootes and Ian Scott. We based the first draft on the design of the Binary IO system in RADIAL, which was originally designed in Manchester by Dave Bailes and Dave Cooper.

Finally, the hard work of programming `vsl` and the vxl/io libraries was performed by the C++ user's group at the Dept. of Imaging Science, University of Manchester; Danny Allen, Tim Cootes, Nick Costen, Ian Scott, Christine Beeston, David Cristinacce, Franck Bettinger, Louise Butcher, and John Kang.

E.2 Adding Binary I/O to a New Class

The golden rule for trouble free binary IO is

> Always write the input and output code in tandem - the output should precisely mirror the input.

First you need to evaluate whether the class needs binary I/O. For example, the `vnl_matrix_ref` can just be left to inherit from `vnl_matrix` because it does not have any

additional member variables. Often iterators tend to be transient things which should not be saved.

Where one does wish to save state, there are four cases to consider:

1. The class is a non-polymorphic (plain) class with no virtual functions
2. The class is a base class with virtual functions
3. The class is a derived class with virtual functions
4. The class is derived from a plain class but is not strictly polymorphic, and has no virtual functions

The last must be considered because various classes in vxl are derived from plain classes. The base classes in these cases are kept as plain classes to improve memory efficiency.

E.2.1 Non-polymorphic (Plain) Classes

For plain classes (those with no virtual functions), and those derived from plain classes, you need to add the following functions to the class definition in the .h file.

```
//: Binary save self to stream.
void b_write(vsl_b_ostream &os) const;

//: Binary load self from stream.
void b_read(vsl_b_istream &is);

//: Return IO version number;
short version() const;

//: Print an ascii summary to the stream
void print_summary(vcl_ostream &os) const;

//: Return a platform independent string identifying the class
vcl_string is_a() const;

//: Return true if the argument matches the string identifying the class or any pare
bool is_class(vcl_string const&) const;
```

You will also need to add the following global scope helper function declarations to the .h file

```
//: Binary save vnl_my_class to stream.
void vsl_b_write(vsl_b_ostream &os, const vnl_my_class & b);

//: Binary load vnl_my_class from stream.
void vsl_b_read(vsl_b_istream &is, vnl_my_class & b);
```

NOTE: YOU SHOULD ALSO ADD APPROPRIATE TEST PROGRAMS : See below

E.2.2 Base Classes

For base classes (those classes at the base of a polymorphic hierarchy, with virtual functions), you need to add the following functions to the class definition in the .h file.

```
//: Binary save self to stream.
virtual void b_write(vsl_b_ostream &os) const;

//: Binary load self from stream.
virtual void b_read(vsl_b_istream &is);

//: Return IO version number;
```

```
short version() const;

//: Print an ascii summary to the stream
virtual void print_summary(vcl_ostream &os) const;

//: Return a platform independent string identifying the class
virtual vcl_string is_a() const=0;

//: Return true if the argument matches the string identifying the class
virtual bool is_class(vcl_string const&) const;

//: Create a copy of the object on the heap.
// The caller is responsible for deletion
virtual vanyl_my_base_class* clone() const=0;
```

You will also need to add the following global scope helper function declarations to the .h file

```
//: Binary save vnl_my_class to stream.
void vsl_b_write(vsl_b_ostream &os, const vnl_my_class & b);

//: Binary load vnl_my_class from stream.
void vsl_b_read(vsl_b_istream &is, vnl_my_class & b);

//: Allows derived class to be loaded by base-class pointer
//  A loader object exists which is invoked by calls
//  of the form "vsl_b_read(os,base_ptr)".  This loads derived class
//  objects from the disk, places them on the heap and
//  returns a base class pointer.
//  In order to work the loader object requires
//  an instance of each derived class that might be
//  found.  This function gives the model class to
//  the appropriate loader.
void vsl_add_to_binary_loader(const vanyl_my_base_class& b);

//: Stream output operator for class pointer
void vsl_print_summary(vcl_ostream& os,const vanyl_my_base_class* b);
```

NOTE: YOU SHOULD ALSO ADD APPROPRIATE TEST PROGRAMS : See below

E.2.3 Derived Classes

For derived classes in a polymorphic hierarchy, which include virtual functions, you need to add the following functions to the class definition in the .h file.

```
//: Binary save self to stream.
virtual void b_write(vsl_b_ostream &os) const;

//: Binary load self from stream.
virtual void b_read(vsl_b_istream &is);

//: Return IO version number;
short version() const;

//: Print an ascii summary to the stream
virtual void print_summary(vcl_ostream &os) const;
```

```
//: Return a platform independent string identifying the class
virtual vcl_string is_a() const;

//: Return true if the argument matches the string identifying the class or any pare
bool is_class(vcl_string const&) const;

//: Create a copy of the object on the heap.
// The caller is responsible for deletion
virtual vanyl_my_base_class* clone() const;
```

In this case you should not need any global helper functions, as they are included in the base class.
NOTE: YOU SHOULD ALSO ADD APPROPRIATE TEST PROGRAMS : See below

E.2.4 The Implementation Code to Add

The following is a template for the standard code for the implementation of each of the methods and functions declared above.

E.2.4.1 Version Numbering

This identifies the I/O version numbering. It needs to be incremented when the class's binary I/O changes.

There is no reason why this version number can't be used for other purposes. So long as the `b_read()` method is updated to deal with it.

It is common and perfectly acceptable to modify a classes I/O during very early development without changing the version number.

```
//: Return IO version number
short vanyl_my_class::version() const
{
  return 1;
}
```

E.2.4.2 Saving Binary State to a Stream

Classes should first save their version number. Derived classes should then call the base class `b_write()`, if the base class has any data members. Finally write the classes data members.

If data is duplicated within the class, (e.g. some workspace data) you will have to decide whether to save and reload the data, or else not save it and regenerate it on loading.

The standard form for the method is

```
//: Binary save self to stream.
void vanyl_my_class::b_write(vsl_b_ostream &os) const
{
  vsl_b_write(os, version());
  vanyl_my_base_class::b_write(os); // vanyl_my_base_class is parent of vanyl_my_class
  vsl_b_write(os, this->my_value);
  ...
}
```

E.2.4.3 Restoring Binary State from a Stream

This function is less complicated than it looks. The complexity of the switch statement is there solely to give as much backwards compatibility as you are willing to program.

```
//=====================================
//: Binary load self from stream.
void vanyl_my_class::b_read(vsl_b_istream &is)
{
  if (!is) return;

  short ver;
  vsl_b_read(is, ver);
  switch(ver)
  {
   case 1:
    vanyl_my_base_class::b_read(is); // vanyl_my_base_class is parent of vanyl_my_class
    vsl_b_read(is, this->my_value);
    ...
    break;

   default:
    vcl_cerr << "I/O ERROR: vanyl_my_class::b_read(vsl_b_istream&)\n"
             << "           Unknown version number "<< ver << '\n';
    is.is().clear(vcl_ios::badbit); // Set an unrecoverable IO error on stream
    return;
  }
}
```

E.2.4.4 Print a Human Readable Summary to a Stream

Classes should print a short summary of their contents. Devolve printing of complex members and parent classes to that class, only if it is sensible.

Do not give too much information. For instance if your class is a container print the number of elements , and then no more than perhaps the first 5 elements, with ellipsis to indicate if there are more.

To aid clear printing the following rule has been found to be helpful.

> If a summary can be printed on a single line, then do not output a linefeed. If a summary contains linefeeds in the middle, then finish the summary with a linefeed.

This approach gives flexibility in the output format whilst preserving readability and predictability for programmers.

The standard form for the method is

```
//: Output a human readable summary to the stream
void vanyl_my_class::print_summary(vcl_ostream &os) const
{
  os<<"Important Value: "<<this->my_value<<" .. ";
  ...
  // optionally os << vcl_endl;
}
```

E.2.4.5 Platform Independent Class Identification

Unfortunately neither the `type_info` class nor the `type_info::name` field given by RTTI are platform independent. To allow loading of a class by base class pointer, the `is_a()` method is used to identify exactly which object to create. It can also be used for other purposes.

If the class is not part of an inheritance hierarchy or will not ever be loaded by base class pointer, the `is_a()` method is not necessary and can be left out.

```
//: Return a platform independent string identifying the class
vcl_string vanyl_my_class::is_a() const
{
  return vcl_string("vanyl_my_class");
}
```

The `is_class()` methods are not used by the vsl system, but are useful when you don't have RTTI.

```
//: Return true if the argument matches the string identifying the class or any parent
bool vanyl_my_class::is_class(vcl_string const& s) const
{
  return s==vanyl_my_class::is_class || vanyl_parent_class::is_class(s);
}
```

E.2.4.6 Polymorphic Copy Creation onto the Heap

The base class loader scheme needs to be able to create a new object on the heap from a base˙class pointer to an old one. If the class is not part of an inheritance hierarchy or will not ever be loaded by base class pointer, the `clone()` method is not necessary and can be left out.

The code below assumes that either a working copy constructor has been defined, or else that compiler generated default copy constructor is suitable (e.g. the class has no pointer or reference member variables.)

```
//: Create a copy of the object on the heap.
// The caller is responsible for deletion
vanyl_my_base_class* vanyl_my_class::clone() const
{
  return new vanyl_my_class(*this);
}
```

E.2.4.7 Helper Functions.

```
//: Binary save vnl_vector to stream.
void vsl_b_write(vsl_b_ostream &os, const vanyl_my_class & v)
{
  v.b_write(os);
}

//: Binary load vnl_vector from stream.
void vsl_b_read(vsl_b_istream &is, vanyl_my_class & v)
{
  v.b_read(is);
}
```

It is acceptable to inline the previous two functions, in which case a half-decent compiler will optimise them completely away.

```
//: Output a human readable summary to the stream
void vsl_print_summary(vcl_ostream& os,const vanyl_my_class& v)
{
  os << v.is_a() << ": ";
  vsl_indent_inc(os);
  v.print_summary(os);
  vsl_indent_dec(os);
}
```

E.2.4.8 Base Classes

The implementation code for functions to be added to base classes is as follows

```
//=============================================
//: Allows derived class to be loaded by base-class pointer.
//  A loader object exists which is invoked by calls
//  of the form "vsl_b_read(os,base_ptr);".  This loads derived class
//  objects from the stream, places them on the heap and
//  returns a base class pointer.
//  In order to work the loader object requires
//  an instance of each derived class that might be
//  found.  This function gives the model class to
//  the appropriate loader.
void vsl_add_to_binary_loader(const vanyl_my_base_class& b)
{
  vsl_binary_loader<vanyl_my_base_class>::instance().append(b);
}

//=============================================
//: Stream summary output for base class pointer
void vsl_print_summary(vcl_ostream& os,const vanyl_my_base_class* b)
{
  if (b)
    return vsl_print_summary(*b);
  else
    return os << "No vanyl_my_base_class defined.\n";
}
```

If the base class is abstract, then you should not to provide implementations for the following methods. Instead declare them as pure virtual methods in the .h file.

```
  //: Create a copy on the heap and return base class pointer
  virtual my_base_class* clone() const = 0;

  //: Print class to os
  virtual void print_summary(vcl_ostream& os) const = 0;

  //: Save class to binary file stream
  virtual void b_write(vsl_b_ostream& bfs) const = 0;

  //: Load class from binary file stream
  virtual void b_read(vsl_b_istream& bfs) = 0;
```

E.2.5 Serialisation - Saving Objects with Shared Ownership

There is problem when you want to save an object whose ownership is shared by several other objects. The most common way this happens is if several objects (A1, A2, ..., An) contain pointers to a single other object B. If all the objects save B, when the stream is reloaded, there wile be n copies of B in memory, instead of the single shared one there was before saving.

If one object can be designated the *owner* (for example if one object, say A1, is responsible for deleting B,) then A1 is responsible for saving and loading object B. This prevents multiple copies of B being created during loading, but all the other A objects do not have their pointer set up correctly.

The standard way of dealing with these problems involves giving each shared object a unique *serial number*, and the process of performing IO on shared objects is called *serialisation*.

If you are using shared objects though existing classes such as `vbl_smart_ptr`, or `vil_image`, then you do not need to do anything, since these classes handle the serialisation.

If you are managing shared ownership in any of your classes, you will need to write the serialising code yourself. There is no language support for serialisation in C++ (unlike Java), however the `vsl` library does provide some support.

The basic algorithm performed by all of the owner objects during saving is

- If it is a null pointer, save a unique serial number.
- else if object B has not been saved before
  - generate a unique serial number for object B
  - save the serial number
  - save object B
- else if we have already saved B
  - find unique serial number for object B
  - save the serial number
- end

The basic algorithm performed by all of the owner objects during loading is

- load unique serial number
- if it is the null-pointer id, set my pointer to 0
- else if this is the first time we have seen this serial number
  - load object B
  - set my pointer to point to B
- else if we have seen the serial number before
  - find location in memory of object B with given serial number
  - set my pointer to point to B
- end

You can use the serialisation record in `vsl_b_istream` and `vsl_b_ostream` to record the serial numbers, and whether you have already saved object B.

The best way to explain the detail is in this simplified and cut-down version of '`vxl/vbl/io/vbl_io_smart_ptr.txx`' In this case all the smart˙ptr<T> objects are the As, and the objects they point to are the Bs.

```
#include <vsl/vsl_binary_io.h>
#include <vbl/vbl_smart_ptr.h>

//: Binary save smart_ptr and serialised *smart_ptr to stream.
template<class T>
void vsl_b_write(vsl_b_ostream & os, const vbl_smart_ptr<T> &p)
{
  // write version number
  const short io_version_no = 1;
  vsl_b_write(os, io_version_no);

  if (p.ptr() == 0)  // Deal with Null pointers first.
  {
    vsl_b_write(os, 0ul); // Use 0 to indicate a null pointer.
                          // True serialisation IDs are always 1 or more.
    return;
  }
```

```
  // Get a serial_number for object being pointed to
  unsigned long id = os.get_serial_number(p.ptr());

  // Find out if this is the first time the object being
  // pointed to is being saved
  if (id == 0)
  {
    // Store a record of the address of B and get a serial number
    id = os.add_serialisation_record(p.ptr());

    vsl_b_write(os, id);     // Save the serial number

    // If you get a compiler error in the next line, it could be because your type T
    // has no vsl_b_write(vsl_b_ostream &,const T*)  defined on it.
    // See the documentation in the .h file to see how to add it.
    vsl_b_write(os, p.ptr());  // Only save the actual object if
                               // it hasn't been saved before to this stream
  }
  else
  {
    vsl_b_write(os, id);         // Save the serial number
  }
}

//=======================================================================
//: Binary load self from stream.
template<class T>
void vsl_b_read(vsl_b_istream &is, vbl_smart_ptr<T> &p)
{
  short ver;
  vsl_b_read(is, ver);
  switch(ver)
  {
   case 1: {
    unsigned long id; // Unique serial number identifying object
    vsl_b_read(is, id);

    if (id == 0) // Deal with Null pointers first.
    {
      p = 0;
      return;
    }

    T * pointer = (T *) is.get_serialisation_pointer(id);
    if (pointer == 0) // Not loaded before
    {
      // If you get a compiler error in the next line, it could be because your type T
      // has no vsl_b_read(vsl_b_ostream&,T*&)  defined on it.
      // See the documentation in the .h file to see how to add it.
      vsl_b_read(is, pointer);                 // load object B
      is.add_serialisation_record(id, pointer); // remember location of B
    }
```

```
      p.set_ptr(pointer); // This operator method will set the internal
                          // pointer in vbl_smart_ptr.
      break; }
     default:
      vcl_cerr << "vbl_smart_ptr::b_read() Unknown version number "
               << ver << vcl_endl;
      vcl_abort();
    }
  }
```

You do not need to write any special code for the owned object, except to provide member functions for loading and saving the object by pointer. These will already exist if you have written polymorphic IO for the class of object B. If not, the following examples might help.

```
// Save with base class pointers
void vsl_b_read(vsl_b_istream& is, class_B * &p)
{
  delete p;
  bool not_null_ptr;
  vsl_b_read(is, not_null_ptr);
  if (not_null_ptr)
  {
    p = new class_B;
    vsl_b_read(is, *p);
  }
  else
    p = 0;
}

template<class T>
void vsl_b_write(vsl_b_ostream& os, const class_B *p)
{
  if (p==0)
  {
    vsl_b_write(os, false); /* Indicate null pointer stored */
  }
  else
  {
    vsl_b_write(os,true); /* Indicate non-null pointer stored */
    vsl_b_write(os,*p);
  }
}
```

The upshot of this is that serialisation can be tricky, but not necessarily difficult. You are advised to manage shored ownership though smart pointers which will do all the serialisation (and memory management) for you.

E.3 Adding Binary I/O to Level-1 Libraries

There are several reasons for doing slightly different things for classes that are members of a level-1 library.

1. No level-one library can depend on any library except vcl.
2. We do not wish to 'break' any existing code
3. There is a desire to keep the level 1 libraries small. IO is available if required.

Binary IO for the level-1 libraries (`vbl`,`vil`,`vgl`,`vnl`) is implemented using 'clip-on' libraries, the code for which lives in 'io' subdirectories of each library. Thus the code for

the IO for vnl_vector<T> lives in 'vnl/io/vnl_io_vector.h/.txx'. Binary IO for vcl containers is provided in the library vsl.

The 'clip-on' libraries provide a set of external functions and classes which allow binary save and restore of the level 1 classes through their public member functions.

Essentially this means writing the following functions for each class:

```
//: Binary save vnl_my_class to stream.
void vsl_b_write(vsl_b_ostream &os, const vnl_my_class & c);

//: Binary load vnl_my_class from stream.
void vsl_b_read(vsl_b_istream &is, vnl_my_class & c);

//: Stream output operator for class pointer
void vsl_print_summary(vcl_ostream& os, const vnl_my_class& c);
```

Each one is written using only the public access functions of the class.

For instance, vgl_point_2d has binary IO as follows:

```
// This is core/vgl/io/vgl_io_point_2d.h
#ifndef vgl_io_point_2d_h_
#define vgl_io_point_2d_h_
//:
// \file
// \author Tim Cootes

#include <vgl/vgl_point_2d.h>
#include <vsl/vsl_binary_io.h>

//: Binary save vgl_point_2d to stream.
template <class T>
void vsl_b_write(vsl_b_ostream &os, const vgl_point_2d<T>& p);

//: Binary load vgl_point_2d from stream.
template <class T>
void vsl_b_read(vsl_b_istream &is, vgl_point_2d<T>& p);

//: Print human readable summary of a vgl_point_2d object to a stream
template <class T>
void vsl_print_summary(vcl_ostream& os,const vgl_point_2d<T>& p);

#endif // #ifndef vgl_io_point_2d_h_
```

The implementation is

```
// This is core/vgl/io/vgl_io_point_2d.txx

#include "vgl_io_point_2d.h"
#include <vsl/vsl_binary_io.h>

//=======================================================================
//: Binary save vgl_point_2d to stream.
template<class T>
void vsl_b_write(vsl_b_ostream &os, const vgl_point_2d<T>& p)
{
  vsl_b_write(os, p.version());
  vsl_b_write(os, v.x());
  vsl_b_write(os, v.y());
}
```

```
//=======================================================================
//: Binary load vgl_point_2d from stream.
template<class T>
void vsl_b_read(vsl_b_istream &is, vgl_point_2d<T>& p)
{
  if (!is) return;

  short w;
  vsl_b_read(is, w);
  switch(w)
  {
   case 1:
    vsl_b_read(is, p.x());
    vsl_b_read(is, p.y());
    break;

   default:
    vcl_cerr << "I/O ERROR: vsl_b_read(vsl_b_istream&, vgl_point_2d<T>&)\n"
             << "           Unknown version number "<< w << '\n';
    is.is().clear(vcl_ios::badbit); // Set an unrecoverable IO error on stream
    return;
  }
}


//=======================================================================
//: Output a human readable summary of a vgl_point_2d object to the stream
template<class T>
void vsl_print_summary(vcl_ostream &os, const vgl_point_2d<T>& p)
{
  os<<'(';
  vsl_print_summary(p.x());
  os<<','
  vsl_print_summary(p.y());
  os<<')';
}

#define VGL_IO_POINT_2D_INSTANTIATE(T) \
template void vsl_print_summary(vcl_ostream &, const vgl_point_2d<T >&); \
template void vsl_b_read(vsl_b_istream &, vgl_point_2d<T >&); \
template void vsl_b_write(vsl_b_ostream &, const vgl_point_2d<T >&)
```

E.3.1 'Clip-On' IO for Polymorphic Hierarchies

Here are some examples:

E.3.1.1 Clip-on IO BaseClass Header

Here is an example of a '`BaseClassIO.h`'

```
// This is BaseClass.h
#ifndef BaseClassIO_h_
#define BaseClassIO_h_
```

```
#include <vsl/vsl_binary_io.h>

// Predeclare classes
class BaseClass;

//:
// \file
// \author Tim Cootes

//: Base for objects which provide IO for classes derived from BaseClass
class BaseClassIO
{
 public:
  //: Constructor
  BaseClassIO();

  //: Destructor
  virtual ~BaseClassIO();

  //: Create new object of type BaseClass on heap
  virtual BaseClass* new_object() const;

  //: Write derived class to os using baseclass reference
  virtual void b_write_by_base(vsl_b_ostream& os, const BaseClass& base) const;

  //: Write derived class to os using baseclass reference
  virtual void b_read_by_base(vsl_b_istream& is, BaseClass& base) const;

  //: Copy this object onto the heap and return a pointer
  virtual BaseClassIO* clone() const;

  //: Return name of class for which this object provides IO
  virtual vcl_string target_classname() const;

  //: Return true if b is of class target_classname()
  //  Typically this will just be "return b.is_class(target_classname())"
  //  However, third party libraries may use a different system
  virtual bool is_io_for(const BaseClass& b) const;
};

//: Add example object to list of those that can be loaded
//  The vsl_binary_loader must see an example of each derived class
//  before it knows how to deal with them.
//  A clone is taken of b
void vsl_add_to_binary_loader(const BaseClassIO& b);

//: Binary save to stream by baseclass pointer
void vsl_b_write(vsl_b_ostream &os, const BaseClass * b);

//: Binary read from stream by baseclass pointer
void vsl_b_read(vsl_b_istream &is, BaseClass* &b);

//: Binary save vgl_my_class to stream.
void vsl_b_write(vsl_b_ostream &os, const BaseClass & b);
```

```
//: Binary load vgl_my_class from stream.
void vsl_b_read(vsl_b_istream &is, BaseClass & b);

//: Print human readable summary of object to a stream
void vsl_print_summary(vcl_ostream& os,const BaseClass & b);
```

E.3.1.2 Clip-on IO BaseClass Implementation

For a BaseClass, one creates '`BaseClassIO.cxx`' as follows

```
// This is BaseClassIO.cxx
#include "BaseClassIO.h"
//:
// \file
// \author Tim Cootes

#include <BaseClass.h>
#include <vsl/vsl_clipon_binary_loader.txx>

//: Constructor
BaseClassIO()::BaseClassIO()
{
}

//: Destructor
BaseClassIO()::~BaseClassIO()
{
}

//: Create new object of type BaseClass on heap
BaseClass* BaseClassIO()::new_object() const
{
  return new BaseClass;
}

//: Write derived class to os using baseclass reference
void BaseClassIO()::b_write_by_base(vsl_b_ostream& os, const BaseClass& base) const
{
  vsl_b_write(os,base);
}

//: Write derived class to os using baseclass reference
void BaseClassIO()::b_read_by_base(vsl_b_istream& is, BaseClass& base) const
{
  vsl_b_read(is,base);
}


//: Copy this object onto the heap and return a pointer
BaseClassIO* BaseClassIO()::clone() const
{
  return new BaseClassIO(*this);
}
```

```
//: Return name of class for which this object provides IO
vcl_string BaseClassIO()::target_classname() const
{
  return string("BaseClass");
}

//: Return true if b is of class target_classname()
bool BaseClassIO()::is_io_for(const BaseClass& b) const
{
  return b.is_a()==target_classname();
}

//=======================================================================
//: Binary write to stream.
void vsl_b_write(vsl_b_ostream &os, const BaseClass & p)
{
  const short io_version_no = 1;
  vsl_b_write(os, io_version_no);
  vsl_b_write(os, p.x());
  vsl_b_write(os, p.y());
  // ... Insert rest of xxxxxx code here
}

//=======================================================================
//: Binary load from stream.
void vsl_b_read(vsl_b_istream &is, BaseClass & p)
{
  if (!is) return;

  short v;
  vsl_b_read(is, v);
  switch(v)
  {
   case 1:
    vsl_b_read(is, p.x());
    vsl_b_read(is, p.y());
    // ... Insert rest of xxxxxx code here
    break;

   default:
    vcl_cerr << "I/O ERROR: vsl_b_read(vsl_b_istream&, BaseClass&)\n"
             << "           Unknown version number "<< v << '\n';
    is.is().clear(vcl_ios::badbit); // Set an unrecoverable IO error on stream
    return;
  }
}

//=======================================================================
//: Output a human readable summary to the stream
void vsl_print_summary(vcl_ostream& os,const BaseClass & p)
{
  os<<'('<<p.x()<<','<<p.y()<<')';
  // ... Insert rest of xxxxxx code here
}
```

```
//: Add example object to list of those that can be loaded
//  The vsl_binary_loader must see an example of each derived class
//  before it knows how to deal with them.
//  A clone is taken of b
void vsl_add_to_binary_loader(const BaseClassIO& b)
{
  vsl_clipon_binary_loader<BaseClass,BaseClassIO>::instance().add(b);
}


//: Binary save to stream by baseclass pointer
void vsl_b_write(vsl_b_ostream &os, const BaseClass * b)
{
  vsl_clipon_binary_loader<BaseClass,BaseClassIO>::instance().write_object(os,b);
}

//: Binary read from stream by baseclass pointer
void vsl_b_read(vsl_b_istream &is, BaseClass* &b)
{
  vsl_clipon_binary_loader<BaseClass,BaseClassIO>::instance().read_object(is,b);
}

// Explicitly instantiate loader
VSL_CLIPON_BINARY_LOADER_INSTANTIATE(BaseClass, BaseClassIO);
```

E.3.1.3 Clip-on IO Derived Class Header

For a DerivedClass, one creates '`DerivedClassIO.h`' as follows

```
// This is DerivedClassIO.h
#ifndef DerivedClassIO_h_
#define DerivedClassIO_h_
//:
// \file
// \author Tim Cootes

#include <BaseClassIO.h>

class DerivedClass;

//: Provide IO for DerivedClass
class DerivedClass : public BaseClassIO
{
 public:
  //: Constructor
  DerivedClassIO();

  //: Destructor
  virtual ~DerivedClassIO();

  //: Create new object of type DerivedClass on heap
  virtual BaseClass* new_object() const;

  //: Write derived class to os using baseclass reference
```

```
  virtual void b_write_by_base(vsl_b_ostream& os, const BaseClass& base) const;

  //: Write derived class to os using baseclass reference
  virtual void b_read_by_base(vsl_b_istream& is, BaseClass& base) const;

  //: Copy this object onto the heap and return a pointer
  virtual BaseClassIO* clone() const;

  //: Return name of class for which this object provides IO
  virtual vcl_string target_classname() const;

  //: Return true if b is of class target_classname()
  //  Typically this will just be "return b.is_class(target_classname())"
  //  However, third party libraries may use a different system
  virtual bool is_io_for(const BaseClass& b) const;
};

//: Binary save vgl_my_class to stream.
void vsl_b_write(vsl_b_ostream &os, const DerivedClass & b);

//: Binary load vgl_my_class from stream.
void vsl_b_read(vsl_b_istream &is, DerivedClass & b);

//: Print human readable summary of object to a stream
void vsl_print_summary(vcl_ostream& os, const DerivedClass & b);
```

E.3.1.4 Clip-on IO Derived Class Implementation

For a DerivedClass, one creates '`DerivedClassIO.cxx`' as follows

```
// This is DerivedClassIO.cxx

//:
// \file
// \author Tim Cootes

#include <DerivedClass.h>
#include <DerivedClassIO.h>

//: Constructor
DerivedClassIO()::DerivedClassIO()
{
}

//: Destructor
DerivedClassIO()::~DerivedClassIO()
{
}

//: Create new object of type DerivedClass on heap
BaseClass* DerivedClassIO()::new_object() const
{
  return new DerivedClass;
}
```

```
//: Write derived class to os using baseclass reference
void DerivedClassIO()::b_write_by_base(vsl_b_ostream& os, const BaseClass& base) const
{
  vsl_b_write(os,base);
}

//: Write derived class to os using baseclass reference
void DerivedClassIO()::b_read_by_base(vsl_b_istream& is, BaseClass& base) const
{
  vsl_b_read(is,base);
}

//: Copy this object onto the heap and return a pointer
BaseClassIO* DerivedClassIO()::clone() const
{
  return new DerivedClassIO(*this);
}

//: Return name of class for which this object provides IO
vcl_string DerivedClassIO()::target_classname() const
{
  return vcl_string("DerivedClass");
}

//: Return true if b is of class target_classname()
bool DerivedClassIO()::is_io_for(const BaseClass& b) const
{
  return b.is_a()==target_classname();
}

//=======================================================================================
//: Binary write to stream.
void vsl_b_write(vsl_b_ostream &os, const DerivedClass & p)
{
  const short io_version_no = 1;
  vsl_b_write(os, io_version_no);
  vsl_b_write(os, p.x());
  vsl_b_write(os, p.y());
  //... Insert rest of xxxxxx code here
}

//=======================================================================================
//: Binary load from stream.
void vsl_b_read(vsl_b_istream &is, DerivedClass & p)
{
  if (!is) return;

  short v;
  vsl_b_read(is, v);
  switch(v)
  {
   case 1:
    vsl_b_read(is, p.x());
    vsl_b_read(is, p.y());
```

```
    //... Insert rest of xxxxxx code here
    break;

   default:
    vcl_cerr << "I/O ERROR: vsl_b_read(vsl_b_istream&, DerivedClass&)\n"
             << "           Unknown version number "<< v << '\n';
    is.is().clear(vcl_ios::badbit); // Set an unrecoverable IO error on stream
    return;
  }
}
```

There. Nothing to it really.

E.4 Test Programs

It is important to write test programs for all the IO. Below is a summary of how it is done for classes in vgl. Extrapolate as appropriate.

The test programs for vgl live in the subdirectory core/vgl/tests.

To add a test for a class you

1. Write a test function in a file : '`test_classname_io.cxx`'
2. Add a call to that function in the file '`test_driver.cxx`'

For instance, to test the IO in class `vgl_point_2d<T>`, we write '`test_point_2d_io.cxx`':

```
#include <vcl_iostream.h>

#include <vgl/vgl_point_2d.h>
#include <vgl/io/vgl_io_point_2d.h>

#include <testlib/testlib_test.h>

void test_point_2d_io()
{
  vcl_cout << "*******************************\n"
           << "Testing vgl_point_2d<double> io\n"
           << "*******************************\n";

  vgl_point_2d<double> p_out(1.2,3.4), p_in;

  vsl_b_ofstream bfs_out("vgl_point_2d_test_double_io.bvl.tmp");
  TEST("Created vgl_point_2d_test_double_io.bvl.tmp for writing", (!bfs_out), false);
  vsl_b_write(bfs_out, p_out);
  bfs_out.close();

  vsl_b_ifstream bfs_in("vgl_point_2d_test_double_io.bvl.tmp");
  TEST("Opened vgl_point_2d_test_double_io.bvl.tmp for reading", (!bfs_in), false);
  vsl_b_read(bfs_in, p_in);
  TEST("Finished reading file successfully", (!bfs_in), false);
  bfs_in.close();

  TEST("p_out == p_in", p_out, p_in);

  vsl_print_summary(vcl_cout, p_out);
  vcl_cout << vcl_endl;
```

```
}

TESTMAIN(test_point_2d_io);
```

The macros `TEST` and `TESTMAIN` are defined in the file '`testlib/testlib_test.h`'
We must then add a call to this test to '`test_driver.cxx`':

```
#include <testlib/testlib_register.h>

DECLARE( test_point_2d_io );
DECLARE( test_point_3d_io );
// More tests ....

void
register_tests()
{
  REGISTER( test_point_2d_io );
  REGISTER( test_point_3d_io );
  // More tests ....
}

DEFINE_MAIN;
```

When one runs make (under Unix) all the programs are compiled and the tests automatically run. A message will be output giving a summary of how many were successful/unsuccessful.

E.4.1 Summary

To summarise, when adding a test program to v?l you need to

1. Create a test function in '`tests/test_classname_io.cxx`'
2. Add a call to the function in '`tests/test_driver.cxx`'

Concept Index

(Index is nonexistent)