The ZooLib Cookbook

Michael David Crawford

The ZooLib Cookbook

Michael David Crawford

Published February 7, 2009

Table of Contents

Chapter 1. Introduction

ZooLib is a cross-platform application framework. What it allows you to do is to write a single set of C++ sources and compile for different operating systems and microprocessors to produce native executable applications with very little need for platform-specific client code.

This is of great benefit to a developer, as it allows you to support your application on a variety of platforms without a lot of extra work developing parallel codebases. It also allows you to spend the bulk of your time developing on whatever platform you enjoy the most while delivering for the platforms your users need, even if they're not the same.

ZooLib was written over a period of some years starting in the early 1990's by software consultant Andrew Green. In later years a great deal of work was contributed by Andy's clients Learning in Motion [http://www.learn.motion.com/] as well as the Ontario Institute for Studies in Education [http://www.oise.utoronto.ca/] (OISE). It was released as open source under the MIT License in November 2000.

ZooLib's nominal home on the web is www.zoolib.org [http://www.zoolib.org/] but for the time being it is hosted at zoolib.sourceforge.net [http://zoolib.sourceforge.net/].

ZooLib applications are multithreaded. This means there can be multiple sequences of program execution within one application. For the most part there is a thread for each window in a GUI application and a thread for the main application itself, but you can create as many threads as you like, within the limits imposed by the host operating system's resources.

It provides a graphical user interface toolkit that uses renderers to provide a look appropriate to the platform the ZooLib app is running on. For example, on the Macintosh it uses the Appearance Manager if it is available to enable a native Macintosh look.

ZooLib has a uniquely flexible and powerful method of laying out widgets in the user interface. Unfortunately, this very power makes learning to lay out user interfaces in your programs somewhat challenging, but once you come to understand the method (as implemented by the ZPaneLocator class) you will agree that it is indeed very useful.

ZooLib provides platform-independent TCP networking, and also has a database file format. The combination of these two allows one to write database servers with it. As evidence that ZooLib isn't just for GUI, Learning in Motion sells a client/server educational collaboration system called Knowledge Forum that uses a ZooLib database server running without a user interface to serve as many as several thousand ZooLib client programs in a school.

One advantage of ZooLib's database format is that the databases are wholly contained in single files. This allows databases to serve as end user documents. One could email a friend a database without knowing it is a database, but thinking of it as an ordinary document, to be opened for editing in a GUI ZooLib application.

ZooLib contains very helpful support for debugging. There are helpful functions and macros for checking assertions, causing debugger breaks and logging error messages, and many frequently used core components contain assertions that are enabled during debug builds. In addition, there is a debugging memory allocator that checks for such things as write overruns. My experience with developing ZooLib applications is that the programs I write in it are very reliable, because most bugs cannot get by for long before they trigger an assertion.

To enable efficient memory management, ZooLib contains the ZRef template, a reference counted smart pointer. You can use it in cases where you might otherwise have used the more familiar auto_ptr from the C++ standard library, but ZRef is much better than auto_ptr. First, because it uses reference counting, it is safe to use ZRef's in STL containers. That is not the case for auto_ptr. Secondly, ZRef is thread safe; it uses atomic arithmetic operations provided by the processor to increment and decrement the reference count, and so it will do the right thing even when two threads are accessing a ZRef simultaneously.

The need for atomic operations result in ZooLib's use of a small amount of assembly code. For efficiency, this is usually done as inline assembly, although the implementation allows for calling the atomic operations as subroutines.

This brings up the subject of ZooLib's portability. ZooLib has been ported to Mac OS 680x0 and PowerPC, Windows x86, BeOS x86, and Linux for x86 and PowerPC. However, bringing it to a new platform takes more than a simple recompile. Part of the work required to port ZooLib to a new microprocessor involves writing the atomic arithmetic required for ZRef's to work, and also one needs to write implementations of the parts of ZooLib that depend on the operating system or host GUI layer. ZooLib depends in only basic ways on the host it is running on, so it should be possible to fully port ZooLib to any reasonable GUI operating system in a few weeks.

When working with Open Source software, it is important to read and understand the license for each package that you use. ZooLib uses the MIT License, the same license as is used for the X11 graphical user interface system. You will find ZooLib's license in a source code comment at the header of each source file:

```
/* -------------------------------------------------------------
Copyright (c) 2000 Andrew Green and Learning in Motion, Inc.
http://www.zoolib.org

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
COPYRIGHT HOLDER(S) BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHE
AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
------------------------------------------------------------- */
```

You can find the text of a number of Open Source software licenses as well as a discussion of what they mean on the Open Source Initiative's web page [http://www.opensource.org/]. You can also find an in-depth discussion of some of the different licenses at the Free Software Foundation's website at Licensing Free Software [http://www.gnu.org/philosophy/philosophy.html#LicensingFreeSoftware].

Most of the software that has been developed for ZooLib so far has been proprietary, closed source software. There is not yet a lot of sample code for new developers to learn from. At the time of its release, just two demo applications were made available for download, ZHelloWorld and ButtonMessage, both of them being very simple programs.

My plan for this book as I write it is to explain each sample program in detail, and later to write more sample programs, one for each important concept in ZooLib. At the time of this first writing we just have the two, but eventually there will be more and by reading this book you will gain a solid grasp of how to write a ZooLib application.

I have a special reason for writing this manual: I wanted to use ZooLib before it was available as open source, but when Andy was thinking of releasing it sometime soon. I encouraged him to release it to

a small number of developers who could use it in real products and give him feedback to enable him to get the final kinks worked out, and in addition I promised him that if he let me at the source code, I'd write a programmer's manual for it.

For various reasons its taken me a long time to get started, so documentation was unfortunately completely unavailable at the initial Open Source release of ZooLib. For that I accept complete responsibility and apologize to the many developers who have struggled to understand ZooLib without a manual.

I learned to work with ZooLib by actually writing an application in it, simultaneously for Mac OS and Windows. The application was a special purpose graphic editor called Instant Makeover. The "special purpose" was the application of simulated cosmetics to JPEG photos of the user's face, or the faces of famous women. It was to be given away for free over the Internet to promote the sale of real cosmetics, but unfortunately due to the catastrophic decline in Internet investment late in the year 2000 I was never able to complete the program (my client, BeautyRiot.com, eventually released the prototype of Instant Makeover, which was written in Macromedia Director).

It took me a few weeks to learn to work with ZooLib effectively, but once I did I found it uniquely powerful and easy to develop with. Instant Makeover was intended to run on low-end consumer computers, and so it was important that my program run fast. It was my experience that ZooLib was extremely efficient. I was able to do real-time resizing and moving of alpha blended images on a 150 MHz PowerPC 604 Macintosh and find the program responsive. In addition, Instant Makeover had a complex layout in its main window, and screen updates always happened very quickly. ZooLib is not guilty of the poor performance that some other cross-platform solutions are sometimes accused of.

Chapter 2. ZHelloWorld

Our first ZooLib application must of course be the proverbial "Hello, World". With ZooLib we display a window with those words, and menu items that show how to retrieve the text from a resource file, and display a BMP graphic.

ZMain: the Application Entry Point

Every ZooLib application's client code starts up in ZMain(). ZMain is used rather than main() because different operating systems have different conventions for the parameters to be passed to main, and on Windows, WinMain is used rather than main(). Most ZooLib applications will have a ZMain that is much like ZHelloWorld's:

```
int ZMain(int argc, char** argv)
{
    try
    {
        ZHelloWorld_App theApp;
        theApp.Run();
    }
    catch (...)
    {}

    return 1;
}
```

The first thing we do is create an instance of our application object, here ZHelloWorld_App. It is a subclass of ZApp. We call the ZApp's Run() method, and when Run() finishes, we exit ZMain, and the program terminates.

There are ordinarily several threads in a ZooLib application. The application object has its own thread, and each window has a thread. The window threads come and go as windows are created and closed, but the application object's thread keeps running the whole time.

ZHelloWorld_App: the Application Object

Let's look at the class declaration for ZHelloWorld_App:

```
class ZHelloWorld_App : public ZApp
 {
public:
 ZHelloWorld_App();
 virtual ~ZHelloWorld_App();

// From ZWindowSupervisor via ZApp
 virtual void WindowSupervisorInstallMenus(ZMenuInstall& inMenuInstall);

// From ZApp
 virtual void RunStarted();
 };
```

In our case, ZHelloWorld_App has few responsibilities - it initializes the user interface widget factories, installs the menus, and responds to the RunStarted message.

ZooLib needs to have its user interface factories initialized and terminated when the application starts and quits. The natural place to do this is in the application class' constructor and destructor:

```
ZHelloWorld_App::ZHelloWorld_App()
 {
 ZUIUtil::sInitializeUIFactories();
 }

ZHelloWorld_App::~ZHelloWorld_App()
 {
 ZUIUtil::sTearDownUIFactories();
 }
```

ZooLib draws its UI widgets indirectly, through the use of ZUIFactories that manage renderers for the different standard platform appearances. There are renderer sets for each of the platforms supported, so a program running on Windows looks like a native Windows program, for example. For the Mac OS, two ZUIFactories are available, one which provides the classic Platinum look, and another which passes of responsibility to the Mac OS Appearance Manager, so it will adopt the appearance of the particular system it is running on as well as the theme selected by the user.

When you create a UI widget like a button, you ask the ZUIFactory to make one for you. ZUIUtil::sInitializeUIFactories creates the UI factory for you according to the platform.

There is a switchable UIFactory that calls through to a real one that may be changed at runtime. You can also provide your own factory if you want to completely control the appearance of your application, for example to provide a theme. It is more common to customize the appearance of certain items in your UI while letting the default factory handle the rest.

ZApp is a subclass of ZMessageLooper, which is a class that can receive and dispatch ZMessages. You use ZMessages to communicate between threads. When a ZMessageLooper has started up, it sends itself the message "zoolib:RunStarted" ZMessage. ZApp responds to receiving this message by calling its RunStarted virtual member function. ZHelloWorld_App overrides RunStarted, so its version is called instead.

Now looking at RunStarted:

```
void ZHelloWorld_App::RunStarted()
 {
 ZWindow* theWindow = new ZHelloWorld_Window(this);
 theWindow->Center();
 theWindow->BringFront();
 theWindow->GetLock().Release();
 }
```

We see that RunStarted creates a window. It does this by using new to dynamically allocate a ZHelloWorld_Window, passing its this pointer as a parameter. It centers the window on the screen, moves it to the front, and then releases its lock.

Since ZooLib is a multithreaded framework, we need to be concerned about locking objects that might be accessed simultaneously by different threads. Windows run in their own threads, so we need to lock the window for a different thread to modify its state. ZHelloWorld_Window is a subclass of ZWindow; ZWindows are created initially locked. We call GetLock() to get a reference to the ZMutexBase that is the window's lock, then call its Release method to unlock it.

ZWindows are created initially invisible. While they are still locked, you can do what you need to prepare the window to be first shown. Once it is unlocked, the window will draw and begin receiving messages.

Do not delete a window's pointer. If the user or operating system closes the window on the screen, ZooLib will automatically delete the pointer to the window itself. It is possible to programmatically close a window; I will get into this later.

Installing the Menus

ZHelloWorld_App is also responsible for installing menus. Menus may be managed in one central location, as in the ZApp object, or each window can manage its menus on its own, or they can manage the menus in combination. ZApp can manage menus because it is also a subclass of ZWindowSupervisor.

Note that while menus are managed by or on behalf of windows, the location of the menus depends on the platform-specific user interface. On BeOS and Microsoft Windows, each window has its own menu bar. On Mac OS, there is a single global menu bar at the top of the screen. If a window chooses to have a unique menu bar, then activating that window will change the menu at the top of the screen.

At the time of this writing, X11 ZooLib applications are not able to have menus at all. This is because the code to implement them has not been written yet; hopefully it will be available soon. When X11 menus are implemented, the menu bars will be placed in each window.

Now let's look at how menus are installed by a ZApp:

```
void ZHelloWorld_App::WindowSupervisorInstallMenus(ZMenuInstall& inMenuInstall)
 {
 ZApp::WindowSupervisorInstallMenus(inMenuInstall);

 if (ZRef<ZMenu> appleMenu = inMenuInstall.GetAppleMenu())
  {
  appleMenu->RemoveAll();
  appleMenu->Append(new ZMenuItem(mcAbout, "About " + this->GetAppName() + "...
  }

 ZRef<ZMenu> fileMenu = new ZMenu;
 inMenuInstall.Append("&File", fileMenu);
  fileMenu->Append(mcClose, "&Close", 'W');
  fileMenu->AppendSeparator();
  fileMenu->Append(mcQuit, "&Quit", 'Q');
 }
```

First we call the base class to allow it to do any menu installation it needs.

Then, we check if an Apple menu is available. This is the menu on the top-left corner of the screen running under the Mac OS, whose title is an apple symbol. If we're running on a Macintosh, the ZRef<ZMenu> returned by GetAppleMenu will contain a pointer to the Apple menu's ZMenu; otherwise it will contain null. If the menu exists, we clear it, and append a ZMenuItem pointer with the "About ZHelloWorld..." item in it.

The individual menu items are represented by ZMenuItems. The menus are represented by ZMenus. You can either create a ZMenuItem and call ZMenu::Append to put it in the menu, or you can call the overloaded ZMenu::Append with the text of the item, the menu command selector, and the accellerator key. ZMenu::AppendSeparator will append a horizontal line to the menu, useful for separating groups of menu items from each other.

The ZMenuInstall object receives the ZMenus and takes care to put them in the menu bar for you. Once you have a ZRef<ZMenu> you can call ZMenuInstall::Append to put the menu into the menu bar. Note that you can still install menu items in the ZMenu after it has been placed in the ZMenuInstall.

ZRef: the Thread-Safe Reference Counted Smart Pointer

In our menu code we have our first mention of a very important utility template in ZooLib: the ZRef. The ZRef template is a thread-safe reference counted smart pointer. ZooLib uses it for a purpose similar to auto_ptr from the standard library, or boost::shared_ptr from the Boost C++ library [http://www.boost.org/] - automated management of allocated memory. All three templates share the behaviour that they automatically delete the pointer when you are done with it, and they provide for exception safety.

If you allocate a pointer with new, hold it in a variable (whether a local variable or a member variable), and manually call delete when you are done, your code will work correctly under normal circumstances. But it is not exception safe. If you hold a pointer in a local variable, and an exception is thrown before you get to the delete, you will leak memory, and potentially other important resources held by the memory, like open files, network connections or database locks. Only whole objects are automatically destroyed when they go out of scope - pointers and references are not touched.

If you hold a pointer in a member variable and delete it in the class' destructor, it will leak memory if an exception is thrown during the constructor - incompletely constructed objects are not destroyed if an exception is thrown during their construction. This has to be so because the object may be in an inconsistent state that may have unpredictable behavior if the destructor is called.

The automatic memory management and exception safety both are enabled by holding the pointers in objects that are possessed "by value" - as whole objects, either on the stack or as member variables. Whole, "by value" objects are guaranteed to be destroyed if they go out of scope, whether by leaving some code inside a pair of curly braces, or if the object that holds them is destroyed, or an exception is thrown. Each kind of smart pointer will delete the pointer it holds in its destructor, if it judges that you are done with it - the different types have different policies for this.

Why not just use auto_ptr? auto_ptr does not allow the sharing of resources. If you copy or assign an auto_ptr, the ownership of the resource passes to the recipient. This is an odd behaviour, but is necessary for simple management of a pointer to work. auto_ptr is also incompatible with the Standard Template Library containers; a std::vector< auto_ptr< Foo > > simply will not work.

To enable resource sharing, and sensible behavior upon copying and assignment, you need reference counting. boost::shared_ptr does reference counting, why not use it? ZooLib needs to provide its own template because ZooLib is a multithreaded framework. boost::shared_ptr only provides for single-threaded operation. The increment and decrement of the reference count must be performed with atomic memory operations: two threads may be acquiring or releasing the same shared object simultaneously. Such operations are not directly available in C++; sometimes they are provided as library functions by the host OS, but often the set of atomic operations is not sufficient for everything ZRef does. For that reason, the atomic functions needed by ZooLib are provided by assembly code, usually inline assembly, in the files ZAtomic.h and ZAtomic.cpp

When you construct a ZRef, you pass it a pointer to an object that has just been allocated with new. The ZRef holds the pointer as a data member. Classes that can be held with ZRefs must inherit from ZRefCounted. The ZRefCounted base class holds the reference count, so ZRef can tell the ZRefCounted object to initialize its reference count to 1.

If a ZRef object is assigned or copied, the reference count is increased by one. If a ZRef is destroyed, the reference count of the ZRefCounted it points to is decreased first. If the reference count reaches zero, then the ZRefCounted pointer is deleted.

Syntactically, you can mostly treat the ZRef like an ordinary pointer. It provides overloads for operator-> and operator* that access the pointer. You can also store NULL in a ZRef, it will act like a nil pointer.

ZHelloWorld_Window: the Window Objects

The Window Class Declaration

The objects that represent our windows are of class ZHelloWorld_Window. This is a subclass of ZWindow and of ZPaneLocator:

```
class ZHelloWorld_Window : public ZWindow,
       public ZPaneLocator
 {
public:
 ZHelloWorld_Window(ZApp* inApp);
 ~ZHelloWorld_Window();

// From ZEventHr via ZWindow
 virtual void DoInstallMenus(ZMenuInstall* inMenuInstall);
 virtual void DoSetupMenus(ZMenuSetup* inMenuSetup);
 virtual bool DoMenuMessage(const ZMessage& inMenuMessage);

// From ZPaneLocator
 virtual bool GetPaneLocation(ZSubPane* inPane, ZPoint& outLocation);

protected:
 ZWindowPane* fWindowPane;
 ZUICaptionPane* fHelloPane;
 };
```

Looking in ZWindow.h, we see that ZWindow is a subclass of ZOSWindowOwner, ZMessageLooper, ZMessageReceiver and ZFakeWindow.

ZOSWindowOwner links ZWindow to the real windows supplied by the operating system's GUI layer. You will find the implementations of the different OS windows in each of the subdirectories of zoolib/platform: ZOSWindow_Mac, ZOSWindow_Windows and so on.

ZMessageLoopers may have messages posted to them and are responsible for dispatching them to the ZMessageReceivers, which receive and handle them. Windows handle messages not only to respond to menu commands, but also to GUI events like keypresses, mouse clicks, activations, notifications that the window needs to draw, resizing and so on. You can also define your own messages to allow different threads to communicate among each other or to themselves.

ZFakeWindow is a subclass of ZEventHr, which is defined to respond to most GUI events. ZHelloWorld_Window overrides several of ZEventHr's methods to provide menu handling, similar to the menu handling provided by the application object.

Our First Look at ZPaneLocator

ZHelloWorld_Window is a subclass of ZPaneLocator. The ZPaneLocator class is a central concept in the management of ZooLib graphical user interfaces, and it is very powerful and flexible, but it seems to be difficult for most beginners to learn to work with. I had a hard time with it myself, but I found it very worthwhile to learn how to use it well. I will discuss it in some detail in this book, returning to it several times.

ZPaneLocators serve several functions, the first of them being the layout of widgets in windows. ZPaneLocators have a number of other duties that I will get to later.

In most GUI frameworks, the location and size of each widget are stored as member variables in the widget. This is even the case for non-object oriented toolkits, such as the Mac OS Control Manager, where a Mac OS Button stores its own location in a data structure.

This works well for the most part, but is difficult to work with when the layout of the window is complex and must be flexible. If we want the widgets to rearrange themselves as the window is resized, or to automatically adjust for the width of label text that may be translated into different languages, it is hard for the individual widgets to know how to adjust.

It is particularly inflexible if the windows are designed with a graphical layout tool that saves the widget coordinates in files, such as Mac or Windows resource files.

In ZooLib, individual GUI widgets are not responsible for knowing their own sizes or locations. Instead, they hold a pointer to their ZPaneLocator, and any inquiries about dimensions are passed on to the locator. Typically a ZPaneLocator manages a number of different widgets and can carry out the calculations needed to keep them arranged relative to each other.

ZHelloWorld_Window is a simple ZPaneLocator, it only serves to provide the location of its subpanes:

```
bool ZHelloWorld_Window::GetPaneLocation(ZSubPane* inPane, ZPoint& outLocation)
 {
 if (inPane == fHelloPane)
  {
  ZPoint theSize = inPane->GetSize();
  ZPoint superSize = inPane->GetSuperPane()->GetInternalSize();
  outLocation = (superSize - theSize) / 2;
  return true;
  }
 return ZPaneLocator::GetPaneLocation(inPane, outLocation);
 }
```

GetPaneLocation is passed a pointer to the ZSubPane whose location is needed, and a reference to the ZPoint where the location is to be stored. The code here tests if the subpane is the one whose pointer is stored in the member variable fHelloPane, if it is, it calls the subpane's GetSize() method to find its size, and the pane's superpane's GetInternalSize() method to find the size of the pane it is inside of. Then it divides the vector difference of these by two to get the value to place in outLocation. This has the effect of centering fHelloPane in its superpane.

GetPaneLocation returns true if it handled the call by supplying the location, otherwise it passes on the call by returning the result of ZPaneLocator::GetPaneLocation. With this, it is possible to chain PaneLocators that handle different responsibilities.

Now what does ZSubPane::GetSize() do? We can have a look at the source code in ZPane.cpp:

```
ZPoint ZSubPane::GetSize()
 {
 ZPoint theSize;
 if (fPaneLocator && fPaneLocator->GetPaneSize(this, theSize))
  return theSize;
 if (fSuperPane)
  return fSuperPane->GetInternalSize();
 return ZPoint::sZero;
 }
```

If the pane has a non-nil ZPaneLocator pointer, then it called GetPaneSize to ask the pane locator for the size. If it has no locator, then it asks for the internal size of its superpane, so if there is no pane locator, it fills up its whole superpane. If it has no superpane it defaults to (0,0). Thus we see that by default, ZSubPanes do not know about their sizes on their own. The code for ZSubPane::GetLocation() is similar.

It is possible to override GetSize() and provide a size directly if you want to do so. That makes the most sense for widgets that will always be the same size. Alternatively, you can do this in a superpane that wants to set its size to just surround all its subpanes.

Constructing and Destructing the Window

The constructor for the ZHelloWorld_Window calls its base class constructor, passing it the ZApp pointer (used as a ZWindowSupervisor pointer) and a pointer to a ZOSWindow it has just created. It also constructs its other base class, ZPaneLocator, by passing nil as the next locator in the chain, to indicate that there are no others. Then it creates the window's content:

```
ZHelloWorld_Window::ZHelloWorld_Window(ZApp* inApp)
: ZWindow(inApp, sCreateOSWindow(inApp)),
 ZPaneLocator(nil)
 {
 this->SetTitle("Hello World Window");
 this->SetBackInks(ZUIAttributeFactory::sGet()->GetInk_WindowBackground_Dialog(
 fWindowPane = new ZWindowPane(this, nil);
 fHelloPane = ZUIFactory::sGet()->Make_CaptionPane(fWindowPane, this, "Hello Wo
 }
```

Creating the ZOSWindow

You will need to provide a function like sCreateOSWindow for each window variation you wish to create. If it is a member of your window's class it should be declared static as it is used during the constructor initializer list, when the window object is not completely constructed yet (in general you should never pass "this" as a parameter to functions called from initializer lists, not even implicitly by calling your own non-static member functions. It is permissible to pass "this" to base class member functions, as any base classes have already been constructed)

sCreateOSWindow is responsible for creating the real window on the screen that is managed by the operating system:

```
static ZOSWindow* sCreateOSWindow(ZApp* inApp)
 {

 ZOSWindow::CreationAttributes attr;

 attr.fFrame = ZRect(0, 0, 200, 80);
 attr.fLook = ZOSWindow::lookDocument;
 attr.fLayer = ZOSWindow::layerDocument;
 attr.fResizable = true;
 attr.fHasSizeBox = true;
 attr.fHasCloseBox = true;
 attr.fHasZoomBox = true;
 attr.fHasMenuBar = true;
 attr.fHasTitleIcon = false;

 return inApp->CreateOSWindow( attr );

 }
```

First you initialize a ZOSWindow::CreationAttributes structure with your options for the size, appearance and behaviour of the window. Note the fLook and fLayer options; the look and feel of the window are specified separately. Using ZOSWindow::lookDocument and ZOSWindow::layerDocument creates an ordinary kind of window. You can also create windows with appropriate appearances for modal dialogs, movable modal dialogs, tool palettes and so on. From ZOSWindow.h:

```
enum Look { lookDocument, lookPalette, lookModal, lookMovableModal,
```

```
lookAlert, lookMovableAlert, lookThinBorder, lookMenu, lookHelp };
```

ZooLib allows windows to be managed in different ways, to provide normal window behaviour, or modal dialogs (windows that must be dealt with by the user before work can continue), windows that float above the rest, and "sinkers" or windows that stay at the bottom of the heirarchy. The selections available are again found in ZOSWindow.h:

```
enum Layer { layerDummy, layerSinker, layerDocument, layerFloater, layerDialog,
           layerMenu, layerHelp,
  layerBottomMost = layerDummy, layerTopMost = layerHelp };
```

Creating the Window Contents

Now we examine the body of ZHelloWorld_Window's constructor:

```
this->SetTitle("Hello World Window");
this->SetBackInks(ZUIAttributeFactory::sGet()->GetInk_WindowBackground_Dialog(
fWindowPane = new ZWindowPane(this, nil);
fHelloPane = ZUIFactory::sGet()->Make_CaptionPane(fWindowPane, this, "Hello Wo
```

First we set the window's title.

Then we set the window's background inks. These are the colors that will be drawn if nothing else is - that is, the window will be erased with these colors when an update starts, before the contents are drawn. There are two inks that are provided to SetBackInks; the first is used when the window is active, and the second is used when the window is in the background.

Here is an example of using a factory, in this case the ZUIAttributeFactory, to enable a standard appearance for the application. It calls GetInk_WindowBackground_Dialog to get the normal ink for dialog windows according to the platform standard. For the deactivated ink, we construct a fixed yellow color ink; thus the window will turn yellow when it is no longer in the front.

When a window is constructed, it has no panes in it. The panes are where the actual drawing takes place, and they are the ultimate recipients of UI events like mouse clicks and keystrokes. First we must construct a special pane that takes up the whole window, by allocating a ZWindowPane. We store a pointer to it in fWindowPane.

Finally we get to the real meat of our application, shouting "Hello World!":

```
fHelloPane = ZUIFactory::sGet()->Make_CaptionPane(fWindowPane, this, "Hello Wor
```

We obtain a pointer to the ZUIFactory through its static method sGet(). Then we call Make_CaptionPane to allocate a new ZUICaptionPane that says "Hello World!". The interface to Make_CaptionPane is:

```
virtual ZUICaptionPane* Make_CaptionPane(ZSuperPane* inSuperPane, ZPaneLocator*
```

All ZSubPanes will need to have pointers to their ZSuperPane and ZPaneLocator provided. It is permissible to pass nil for each of these; a nil superpane indicates the subpane is not attached to any window yet, and a nil ZPaneLocator indicates the defaults are to be used for such things as pane size and location.

Once a ZUICaptionPane is placed in a window, it takes care of keeping itself updated. No more work is required to spread our greetings to the world. However, if you want to implement your own subclass of ZSubPane to provide custom drawing, you should do the actual rendering in an override to ZSubPane's DoDraw method.

Destroying the Window

You never explicitly delete a ZWindow pointer. Ordinarily ZWindows are deleted by ZooLib in response to the user clicking the window's close box. But you can provide a destructor for your window. ZHelloWorld_Window does not do anything in its destructor:

```
ZHelloWorld_Window::~ZHelloWorld_Window()
 {}
```

A lot happens behind the scenes in the base class destructors though. ZooLib will delete all the subpanes, the menu bar and menus, and dispose of the operating system window.

You can close a window yourself, and cause its eventual deletion, by calling ZWindow::CloseAndDispose().

On all the systems besides the Mac OS, normal behaviour is for the application to quit after the last window is closed. On the Mac, the application normally stays running with only the top menu bar remaining. ZooLib does not keep count of your windows for you, so it is possible to close all of the windows and have the application object still running. This is the case with ZHelloWorld as the source is provided, and it is a bug. If you close all the windows without selecting "Quit" from the File menu first, the process will be left running with no user interface. You will have to kill the process with the Windows task manager or the kill command on BeOS or Unix.

One simple way to deal with this is for your application object to keep a count of open windows. When a new window is created, it sends a message to the ZApp informing of its birth. It also sends a notification from its destructor. When the count of windows reaches zero, you call PostRequestQuitMessage from the ZApp object. ZooLib will take care of shutting down your application object, and then its Run method will return to ZMain, where you will then return and ZooLib will terminate the program.

Creating the Window's Menus

The window provides a menu in addition to those provided by the application object:

```
void ZHelloWorld_Window::DoInstallMenus(ZMenuInstall* inMenuInstall)
 {
 ZWindow::DoInstallMenus(inMenuInstall);

 ZRef<ZMenu> helloMenu = new ZMenu;

 inMenuInstall->Append("&Hello", helloMenu);
  helloMenu->Append(mcHello_Again, "&Hello World Again!", 'H');
  helloMenu->AppendSeparator();
  helloMenu->Append(mcHello_Pixmap, "My New Niece");
  helloMenu->Append(mcHello_TextResource, "Text Resource");
  helloMenu->Append(mcHello_TextHardCoded, "Text (Hard coded)");
 }
```

ZHelloWorld::DoInstallMenus creates a menu titled "Hello" that has several items in it, one for creating a new Hello World window, as well as showing a picture of Andy's newborn niece Amy from a BMP graphic stored in a resource file, retrieving the message from a resource file, and inserting hardcoded text into the message

Enabling the Menu Items

Before ZooLib responds to a mouse click on the menu bar, it calls DoSetupMenus to allow your code to enable or disable menu items according to the current state of the application or document.

It is important to provide the DoSetupMenus implementation because the menu items are disabled by default:

```
void ZHelloWorld_Window::DoSetupMenus(ZMenuSetup* inMenuSetup)
 {
 ZWindow::DoSetupMenus(inMenuSetup);
 inMenuSetup->EnableItem(mcClose);
 inMenuSetup->EnableItem(mcHello_Again);
 inMenuSetup->EnableItem(mcHello_Pixmap);
 inMenuSetup->EnableItem(mcHello_TextResource);
 inMenuSetup->EnableItem(mcHello_TextHardCoded);
 }
```

DoSetupMenus is passed a pointer to a ZMenuSetup object. Call its EnableItem member function with the menu command constant to enable an item.

Note

I need to check with Andy about this, I see a comment in ZMenu.h that indicates that EnableItem is deprecated.

Handling Menu Messages

Menu messages may be handled by a window or by its ZWindowSupervisor, the application object in our case. This allows common functions to be handled in a central location by the application, but allows menu commands that are particular to a document to be handled by the window that holds the document.

```
bool ZHelloWorld_Window::DoMenuMessage(const ZMessage& inMenuMessage)
 {
 switch (inMenuMessage.GetInt32("menuCommand"))
  {
  case mcClose:
   {
   this->CloseAndDispose();
   break;
   }
  case mcHello_Again:
   {
   ZWindow* theWindow = new ZHelloWorld_Window(ZApp::sGet());
   theWindow->Center();
   theWindow->BringFront();
   theWindow->GetLock().Release();
   return true;
   }
  case mcHello_Pixmap:
   {
   ZDCPixmap thePixmap = ZUIUtil::sLoadPixmapFromBMPResource(kRSRC_BMP_Amy);
   fHelloPane->SetCaption(new ZUICaption_Pix(thePixmap), true);
   break;
   }
  case mcHello_TextResource:
   {
   string theText = ZString::sFromStrResource(kRSRC_STR_HelloWorld);
   ZRef<ZUIFont> theUIFont = ZUIAttributeFactory::sGet()->GetFont_SystemLarge()
```

```
   ZRef<ZUICaption> theUICaption = new ZUICaption_Text(theText, theUIFont, 0);
   fHelloPane->SetCaption(theUICaption, true);
   break;
   }
  case mcHello_TextHardCoded:
   {
   string theText = ZString::sFromStrResource(kRSRC_STR_HelloWorld);
   ZDCFont theDCFont = ZDCFont::sApp9;
   theDCFont.SetStyle(theDCFont.GetStyle() | ZDCFont::underline);
   ZRef<ZUIFont> theUIFont = new ZUIFont_Fixed(theDCFont);
   ZRef<ZUICaption> theUICaption = new ZUICaption_Text("Hello World! (hard code
   fHelloPane->SetCaption(theUICaption, true);
   break;
   }
  }
 return ZWindow::DoMenuMessage(inMenuMessage);
 }
```

Another important concept within ZooLib is the ZMessage. ZMessages allow formatted packets of data to be communicated between threads or within a thread. ZMessages store data of different types that are accessed by name and type. In the case of a menu command, the data stored is a 32-bit integer, and its name is "menuCommand". There will be much to say about ZMessages later on.

Here we switch according to the command after retrieving its value:

```
 switch (inMenuMessage.GetInt32("menuCommand"))
```

The different "mc" constants are defined at the top of the file as integer values following mcUser:

```
#include "ZMenuDef.h"
//...

#define mcHello_Again mcUser + 1
#define mcHello_Pixmap mcUser + 2
#define mcHello_TextResource mcUser + 3
#define mcHello_TextHardCoded mcUser + 4
```

If you look in ZMenuDef.h, you will see that it defines a number of standard UI commands, like mcAbout, which is the command to display an "About Box". It also defines mcUser. The command numbers less than mcUser are reserved for definition by ZooLib, although usually intended to be implemented by your own code. The values above mcUser are for your use as you please.

We'll go into the details of what each of the menu commands do later. But for now notice the last line of the function, which passes off any unknown menu commands to the window:

```
 return ZWindow::DoMenuMessage(inMenuMessage);
```