GXBase Introduction

James Ward

Created: 22nd August 2003.

Updated: 31st October 2003.

This is a brief introduction to the GXBase library, and explains how to create a simple application that does some OpenGL drawing.

Including GXBase

In order to use GXBase, you must *include* the GXBase header files so that your program will compile, and *link* against the GXBase library files, so that the linker can build the library into your application. In fact, the second step (linking) is done automatically, if you have included the GXBase header correctly.

All the GXBase classes use their own *namespace*, called simply **gxbase**. This helps to prevent name collisions between the GXBase classes and any other libraries that you might use. Normally, you can tell the compiler that you wish to use this namespace immediately after including the GXBase headers.

Therefore, the recommended way to include GXBase is as follows:

```
#include "GXBase.h"
using namespace gxbase;

// your code goes here
```

Create an Application Class

The first step in creating an application is to derive a new application class from the **App** base class, and to create a single instance of that class:

1. Derive **MyApp** class from **App**.
2. Create a single instance of **MyApp**.

The following example shows two files **MyApp.h** and **MyApp.cpp** that together implement an 'empty' application that will run and then exit.

MyApp.h contains:

```
#include "GXBase.h"
using namespace gxbase;

// Derive our custom application class from the base class
class MyApp :public App {
};
```

MyApp.cpp contains:

```
#include "MyApp.h"

// Create a single instance of our application, so it can run
static MyApp theApp;
```

Note: you do not need to supply a main() function; this is provided by the framework.

The application class offers a number of *event handlers* in the form of virtual functions. These are used to notify your application when particular events occur. To use them, you simply override the virtual function with your own implementation. Your function will then be called automatically when that event occurs.

For example, the **App** class provides an event handler called **OnCreate** that will be called when the application is first created. This is useful to initialise your application, for example to load configuration files, or to process command line arguments. The following example simply prints a 'hello world' message

when the application first starts.

MyApp.h

```
#include "GXBase.h"
using namespace gxbase;

class MyApp :public App {
public:
      void OnCreate() {
            MsgPrintf( "hello world\n" );
      }
};
```

Create a Window Class

Having created an application class, the next step is to create one or more windows that can be used to display OpenGL drawing. This is very easily achieved by deriving your own custom window class from the **GLWindow** base class, and implementing one or more of the event handlers to perform your custom drawing. Having done this, you can simply add your custom window(s) to the application class as member variables.

The following steps are required to create a new window:

1. Derive **MyWnd** from **GLWindow** class.
2. Insert **MyWnd** into **MyApp** as a member variable.
3. Implement custom drawing code for the various event handlers on **MyWnd**.

This more easily explained with a simple example, so the following code demonstrates how to create a simple application called **MyApp** that includes a custom window **MyWnd** that does some simple OpenGL drawing.

MyApp.h contains:

```
#include "GXBase.h"
using namespace gxbase;

// Derive our custom window class from the base class, it
// just overrrides the OnDisplay event to draw a cross
class MyWnd :public GLWindow {
public:
      void OnDisplay() {
            glClear(GL_COLOR_BUFFER_BIT);

            // draw a cross
            glBegin(GL_LINES);
                  glVertex2f(-1,-1);
                  glVertex2f(+1,+1);
                  glVertex2f(-1,+1);
                  glVertex2f(+1,-1);
            glEnd();

            SwapBuffers();
      }
};

// Derive our custom application class from the base class
class MyApp :public App {
public:
      MyWnd win;       // the window is a child of our app
};
```

MyApp.cpp contains:

```
#include "MyApp.h"

// Create a single instance of our application, so it can run
static MyApp theApp;
```

The **MyWnd** class is based on the **GLWindow** class, and overrides the **OnDisplay** event handler in order to do some OpenGL drawing. This example just clears the window, draws a cross, and swaps the back buffer to the front to make the drawing visible (by default, windows are created with double-buffering enabled).

To create the window and make it appear, we need to create an instance of our custom window. The simplest way to achieve this is to add **MyWnd** as a member variable of our application class. If we wanted to create a second window, identical to the first, we would only need to add a second member variable to the **MyApp** class as shown below:

```
class MyApp :public App {
public:
	MyWnd win1, win2;
};
```

You can have as many different types of window as you choose, and you can share code between them by using the normal inheritance rules.

Looking at the example above, you might be wondering how the OpenGL viewport is made to track the window size? In fact the default implementation of the **GLWindow::OnResize** event handler does this for you automatically. In general, the framework tries to apply some reasonable default behaviour for most cases, but does allow you to override those functions to provide your own implementation when required. The same is true of the **OnDisplay** method for example, which will automatically clear and swap the buffers if you don't provide any drawing code.

Window creation order

The windows are not created at time of construction, but are instead created *immediately after* the **App::OnCreate** has finished executing. This allows window settings such as size and position to be adjusted before the window is created, by placing your initialisation code either in the constructor of your custom window class, or in the **MyApp::OnCreate** function.

For example, you can change window settings in the constructor, so that all windows of that class will be created with the same settings:

```
class MyWnd :public GLWindow {
public:
	MyWnd() {
		// all MyWnd objects will now use this size:
		SetSize(256,256);
	}
};
```

Alternatively, you could initialise your windows in the application **OnCreate** event:

```
void MyApp::OnCreate() {
	w.SetSize(640,480);
	w.SetPosition(200,50);
}
```

There are a few cases where you might want to create the window early; for example so that you have access to a valid OpenGL context so that you can check to see if a particular OpenGL extension is supported.

This can be achieved by simply calling the **GLWindow::Show** function to ensure that the window is created. An example is shown below:

```
void MyApp::OnCreate() {
      w.Show();
      if ( !w.HasExtension("GL_EXT_vertex_array") )
            MsgPrintf("Vertex array extension not found\n");
}
```

In general though, code that requires an OpenGL context at time of creation is best placed in the **GLWindow::OnCreate** function.

Memory allocation

The window classes are typically small enough to safely allocate as members of the application class, which is normally declared statically. However, if you prefer to allocate on the heap (using new and delete), then you can allocate in the **App::OnCreate** event and release in the **App::OnDelete** event.

Simple Animation

In the examples shown so far, the window contents are only redrawn when required (for example, when the window is uncovered or resized). If you want to create animation, you will normally need to refresh the window more frequently. This can be achieved by calling the **GLWindow::Redraw** method, which requests that the window contents are redrawn. This actually schedules an event to redraw the window, and the **GLWindow::OnDisplay** event handler will then be called automatically at the next available opportunity.

The most efficient way to handle window refreshes is to call **GLWindow::Redraw** whenever something in your display changes that would require a redraw. For example, when an object is moved with the mouse. Another alternative is to simply use the **GLWindow::OnIdle** event to continuously request redraws.

An example of a simple rotating cross is shown below. This uses the simplest approach of requesting a redraw from the **OnIdle** event as described above.

```
class AnimWnd :public GLWindow {
public:
      void OnDisplay() {
            glClear(GL_COLOR_BUFFER_BIT);
            glPushMatrix();
                  // rotation angle increases with time
                  glRotated( 45.0 * App::GetTime(), 0,0,1);

                  // draw a cross
                  glBegin(GL_LINES);
                        glVertex2f(-1,-1);
                        glVertex2f(+1,+1);
                        glVertex2f(-1,+1);
                        glVertex2f(+1,-1);
                  glEnd();
            glPopMatrix();
            SwapBuffers();
      }

      void OnIdle() {
            Redraw();  // request to redraw the window
      }
};
```

The above example also demonstrates how the **App::GetTime** function can be used for simple animation that is independent of the speed of the computer. Using this method, the cross will always rotate at the same speed (45 degrees per second) on any computer, although the animation will of course appear smoother on fast machines.

Mouse and Keyboard Events

You will often want to control your application with the mouse and keyboard. There are several events provided by the **GLWindow** class that make this possible:

```
OnMouseMove(int x, int y)
OnMouseButton(MouseButton button, bool down)
OnKeyboard(int key, bool down)
```

To use these, you simply override them in your custom window class, and they will automatically be called when a mouse or keyboard event occurs.

Using OnMouseMove

The **OnMouseMove** event is sent whenever the mouse pointer is moved inside the window. The parameters specify the current (x,y) position of the pointer in pixels, relative to the bottom left hand corner of the window.

The example below shows how OpenGL drawing can easily be made to track mouse movement in real-time.

```
// custom window that tracks mouse movement
class TrackWnd :public GLWindow {
private:
      float u,v;       // used to store mouse coordinates
public:
      TrackWnd() :u(0),v(0) {}

      void OnDisplay() {
            glClear(GL_COLOR_BUFFER_BIT);

            glBegin(GL_LINES);
                  glVertex2f(-1,-1); glVertex2f(u,v);
                  glVertex2f(-1,+1); glVertex2f(u,v);
                  glVertex2f(+1,-1); glVertex2f(u,v);
                  glVertex2f(+1,+1); glVertex2f(u,v);
            glEnd();

            SwapBuffers();
      }

      void OnMouseMove(int x, int y) {
            // scale mouse into range -1 to +1 on each axis
            // 'if' tests are used to avoid division by zero
            if (Width())
                  u = 2.0f * (float)x / (float)Width()  - 1.0f;
            if (Height())
                  v = 2.0f * (float)y / (float)Height() - 1.0f;

            // redraw when mouse moves
            Redraw();
      }
};
```

In this example, notice that the mouse coordinates (x,y) are sent to the **OnMouseMove** function as integer pixel coordinates, and have to be scaled to match the default window coordinate system. Note that the **GLWindow::Width** and **GLWindow::Height** methods can be used to find the current size of the window in pixels. Similarly, the **GLWindow::Aspect** method will return the aspect ratio (width/height) for the current window, as a floating point value (and is safe when Height=0). This is suitable for use with OpenGL functions such as **gluPerspective**.

Using OnMouseButton

The **OnMouseButton** event is sent whenever a mouse button is pressed or released. If the button is held for a long period, the mouse pressed event may be sent several times (auto repeat). The event has two parameters: a **button** identifier (which indicates which button was pressed/released), and a flag called **down**, which indicates if the button is down (true) or up (false).

The example below demonstrates how to use **OnMouseButton**, changing the drawing colour to either Red, Green or Blue, depending upon which mouse button was pressed.

```
void MyWnd::OnMouseButton(MouseButton button, bool down) {
      if (down) return;
      switch (button) {
      case MBLeft:
            glColor3f(1,0,0);     // red
            break;
      case MBMiddle:
            glColor3f(0,1,0);     // green
            break;
      case MBRight:
            glColor3f(0,0,1);     // blue
            break;
      }
      Redraw();
}
```

Notice the first line that just returns immediately if the button is *pressed* down. This ensures that the code in the switch statement is called only when a particular button is *released*. This is because the OnMouseButton event handler will be called twice for each button; once when the button is first pressed down, and again when that button is released.

Using OnKeyboard

The OnKeyboard event is sent when a key is pressed or released. The event has two parameters: an integer **key** identifier (which gives the system keyboard code), and a flag called **down**, which indicates if the key is down (true) or up (false).

The following example changes the drawing colour to either Red, Green or Blue when the R, G or B keys are pressed, then redraws the display if required:

```
void MyWnd::OnKeyboard(int key, bool down) {
      if (!down) return;
      bool needRedraw = true;
      switch( tolower(key) ) {
      case 'r':
            glColor3f(1,0,0);     // red
            break;
      case 'g':
            glColor3f(0,1,0);     // green
            break;
      case 'b':
            glColor3f(0,0,1);     // blue
            break;
      default:
            needRedraw=false;     // some other key..
      }
      if (needRedraw) Redraw();
}
```