Simple OpenGL Framework

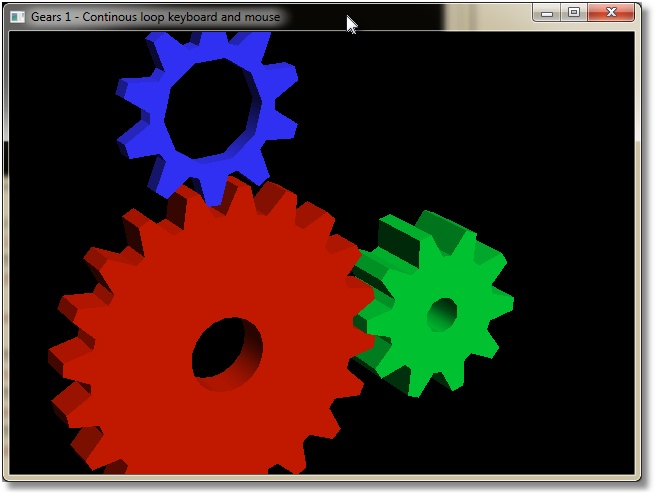
[**http://www.codeproject.com/Articles/371849/Simple-OpenGL-Framework**](http://www.codeproject.com/Articles/371849/Simple-OpenGL-Framework)

**[Member 8823543](http://www.codeproject.com/script/Membership/View.aspx?mid=8823543)**, 26 Apr 2012 [CPOL](http://www.codeproject.com/info/cpol10.aspx)

|  |  |
| --- | --- |
| http://codeproject.cachefly.net/script/Ratings/Images/stars-fill-md.png  http://codeproject.cachefly.net/script/Ratings/Images/stars-empty-md.png | 3.87 (8 votes) |
|  | |  |  |  | | --- | --- | --- | | Rate: | [vote 1vote 2vote 3vote 4vote 5](http://www.codeproject.com/Articles/371849/Simple-OpenGL-Framework) |  | |

GLW is a simple, compact, <i>drop-in</i> framework for developing simple OpenGL games and demos. Its focus is ease of use, platform abstraction and small footprint.

* [**Download source - 35.1 KB**](http://www.codeproject.com/Articles/371849/Simple-OpenGL-Framework/glw_src.zip)
* [**Download demo - 66.3 KB**](http://www.codeproject.com/Articles/371849/Simple-OpenGL-Framework/glw_demo.zip)

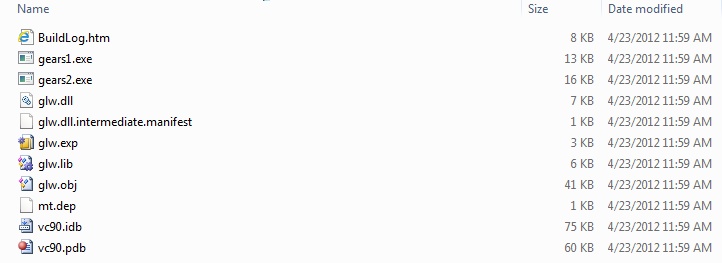


*Classical OpenGL Gears running in GLW framework.*

Welcome

Welcome to GLW – Open**GL** **W**indow!  As the title says, GLW is a very simple to use OpenGL framework. You can use it as a drop-in framework in your applications to open an OpenGL window. As  drop-in, I mean just copy source and header files into your own code and skip any library linking and loading. GLW consists of only one header and one source file so it is just to include gwl.h, and you are good to go.

Alternatively if you wish to use it as a shared library, you can compile it as a dll as well. Included is solution with sources and two projects. One uses static GLW (drop-in source) and another links to shared library. I have adapted standard GL Gears from GLUT 3.6 source as a demo for this article. Just to experiment with size, there are 3 different configurations of project, among which, one is extremely optimized for size (not meant for real use). It turns that in static build, GLW adds approximately around 3 kb to exe. Dll weights in around 6.5 kb (7 on disk):



*Compiled binaries compared in build folder*

Introduction

GLW abstracts away windowing and input in a friendlier, c-like, manner than what you see in standard Win32 API. Actually to use it you don’t have to dig into win32 at all to use it. If you have ever used GLUT, GLFW, or something similar, you will be at home with GLW at once. GLW is written as a very small and tiny subset of GLUT. However, it is GLUT-like, but it is not GLUT-clone, so don’t expect 1 : 1 correspondence to GLUT in GLW API.

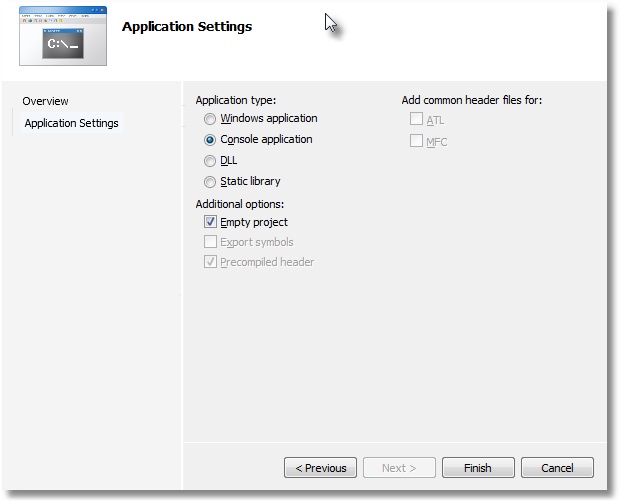
Background

Long time ago I have written myself a library I called GLW that was much more of a GLUT clone. I was always interested about how such frameworks are done, so I put together my own. I alse had handling for several keyboards, mice and gamepads/gamejoysticks at once (with raw\_input). I have used it in my own projects, but it was too messy in some detail to be put it in front of public eyes. Also I never got to Linux part Smile | :) . Since then I have moved to other stuff in my life and game programming at all have been on the shelf for few years for my part. Anyway few days ago I felt for doing something for the fun, so naturally playing with OpenGL comes first on my list. So I picked up my old library, and realized I don’t need really most of the stuff from it for simple tests and demos, so I have cleaned out unnecessary stuff, and left is very minimal but still usable framework to write simple demos or games. My goal is not to make smallest possible in terms of kilobytes; but small in terms of usage and learning curve.

Certainly it is not very difficult to program Win32, but at same time it is unnecessary ugly and involved. Also you are tying your code into Windows platform and making it hard to port to some other platform if you ever want to. I believe that it is always better to abstract those (ugly) platform details away, and create yourself a clean API you are working against. Even if it is for personal use.  If you need a serious framework, as mentioned there are GLUT and its clones, there are Qt, SDL and numerous other libraries and frameworks. Almost any 3d engine or toolkit will come with platform independent and more programmer-friendly windowing API then what one have to work when coding plain Win32 or MFC. If you need a simple to use and learn framework with basic functionality than GLW might be for you. GLW is meant to be used in small, personal demos and projects, if you wish to throw together some small test or demo fast.

Usage

If you are using Visual Studio (I use VS2008 Express), create Win32 empty console project (exe). In your main source file, include glw.h and you are good to go. Be sure you also copy over glw.c, otherwise your linker won’t be happy.



*Visual Studio 2008 Project Settings for an GLW application.*

As shown above, make an empty console app if you are starting a new project in Visual Studio. I suggest you drop in GLW source directly in your code, since you will anyway use almost all of its stuff in any program that opens a window. If you anyway wish to link against dll you have to define preprocessor directive USEGLWDLL.

Rendering

To open a window for rendering just call glwMainLoop() from your main:

Hide   Copy Code

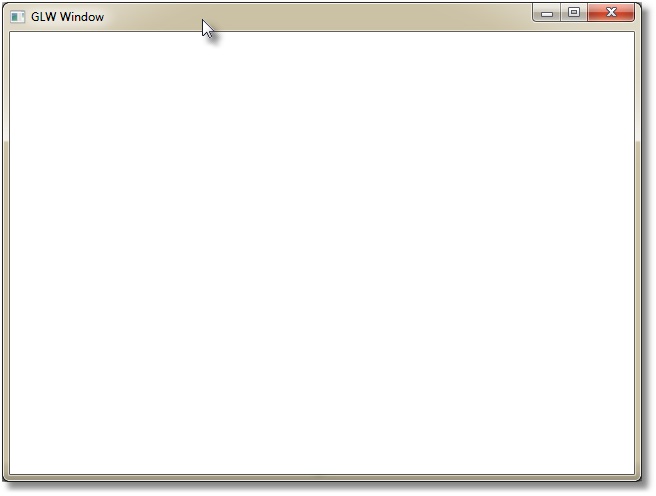
#include "glw.h"

int main(int argc, char\*\* argv){

return glwMainLoop();

}

The result is a simple window on the screen:



*Default window from GLW*

If you wish to remove console window that starts in the background you may add to your source (or change linker settings in project properties):

Hide   Copy Code

#pragma comment(linker, "/subsystem:\"windows\" /entry:\"mainCRTStartup\"")

Don’t disable console windows in gears1 or gears2 tests; it is used for printing some information. To dismiss window and exit application press escape.

Well that is not much of stuff, just an empty window; but it has OpenGL up and initiated with reasonable default 3d projection (first person camera).

Drawing is done in  glwUpdateFunc  callback (GLUTs glutRenderFunc). Draw a simple triangle:

Hide   Copy Code

static void draw(){

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glColor3f(0,1,0);

glBegin(GL\_TRIANGLES);

glVertex3f( 0, 1,0);

glVertex3f(-1,-1,0);

glVertex3f( 1,-1,0);

glEnd();

}

To set up that function as glw rendering callback use glwSetUpdateFunc(draw):

Hide   Copy Code

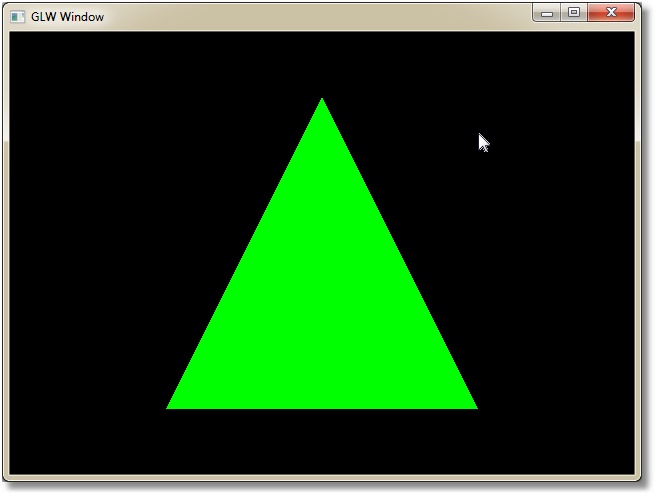
int main(int argc, char\*\* argv){

glwSetUpdateFunc(draw);

return glwMainLoop();

}

Just for the fun, here is the output:



*GLW Window with a triangle in it*

Generally for any callback that GLW uses, there is a setter in form of: glwSetCallbacknameFunc( glwCallbackFuncf ). All callback funtions have name in form of glwCallbacknameFunc( … ).

There are not many callbacks defined; glw only does a window and input, so naturally there are only callbacks to handle those aspects.

By the way; I am using “old ways” to send data to OpenGL, just for the purpose of demonstration for this demo. GLW works fine with shaders, vbos and all other nice modern OpenGL stuff you may wish to use.

Seeing GLW window from a perspective

As you see GLW have reasonable perspective setup so that you may start writing your drawing code immediately. However if you wish to setup perspective for yourself, you have to define

Hide   Copy Code

glwResizeFunc(int width, int height)

  and set it up with

Hide   Copy Code

glwSetResizeFunc

callback. Width and height are off course with and height of window used. For example you might do something like:

Hide   Copy Code

void reshape(int w, int h){

float a;

if (h==0)

h=1;

a = (GLfloat) h / (GLfloat) w;

glViewport(0, 0, (GLint) w, (GLint) h);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

glFrustum(-1.0, 1.0, -a, a, 5.0, 500.0);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

glTranslatef(0.0, 0.0, -10.0);

}

and set it up in *main*:

Hide   Copy Code

int main(int argc, char\*\* argv){

glwSetResizeFunc(reshape);

glwSetUpdateFunc(draw);

return glwMainLoop();

}

If you run same app now, you will see no change Smile | :) . It is just because I have pasted here the resize function glw uses as default for 3d perspective anyway. Now you know what defaults are if you don’t plan on setting up your own perspective.

There is default 2d setup as well. To switch between 3d and 2d mode, use glwSet2DMode() and glwSet3DMode(). To draw same triangle, but in 2d, try this:

Hide   Copy Code

void draw(){

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glColor3f(0,1,0);

glBegin(GL\_TRIANGLES);

glVertex2f(320,0);

glVertex2f(0,480);

glVertex2f(640,480);

glEnd();

}

int main(int argc, char\*\* argv){

glwSet2DMode();

glwSetUpdateFunc(draw);

return glwMainLoop();

}

And same triangle is up and running again.

Getting window size

If you resize the window, triangle will stay at same spot and same size. The reason is that viewport of OpenGL is configured to have origin (0,0) at top left and max at lower right corner of the window, but triangle is configured to draw in fixed coordinates and does not account for change in window size. To update triangle when window size changes, you need window width and height. You get those with

Hide   Copy Code

glwGetWindowSize(int\* w, int\* h);

It will put width and height of the window into parameters you give it:

Hide   Copy Code

void draw(){

int w, h;

glClearColor(1,1,1,0);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glwGetWindowSize(&w,&h);

glColor3f(0,1,0);

glBegin(GL\_TRIANGLES);

glVertex2f((float)w\*0.5f,0);

glVertex2i(0,h);

glVertex2i(w,h);

glEnd();}

}

Now the triangle resizes with the window (with white background this time).

There is also corresponding glwSetWindowSize(int width, int height) to set width and height of the window. However it is only meaningful in context of creation. It means you can use it before you enter main loop to request initial size of the window. You cannot use it later on to change size of the window (use your mouse). In same spirit there are no positioning functions. I just don’t found those very useful for a dirty and simple do-some-quick-drawing-and-close style work. I wanted to keep this simple, not to make it a Swiss-knife of everything.

Animation with GLW

Let’s go back to our 3d triangle and add rotation about y-axis:

Hide   Copy Code

void draw(){

glClearColor(1,1,1,0);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glRotatef(1,0,1,0);

glColor3f(0,1,0);

glBegin(GL\_TRIANGLES);

glVertex3f(0,1,0);

glVertex3f(-1,-1,0);

glVertex3f(1,-1,0);

glEnd();

}

If you run this you would expect normally to see the triangle rotate around Y-axis, with constant speed, but it ain’t happening. By default, glw is event-driven framework. It means it will do nothing until there is an event from the OS. I reason that for simple apps it is not always you wish to do animation. If all you want to do is render a frame with OpenGL content, there is no need to hog CPU with constant redraw. To enable constant redraw, tell glw to use polling (asynchronious) loop instead of event-driven (blocking) one. Switch to polling loop is done with:

Hide   Copy Code

glwSetUpdateMode(GLW\_CONTINOUS\_LOOP);

To switch back to event-driven loop, use

Hide   Copy Code

glwSetUpdateMode(GLW\_EVENT\_LOOP);

To see it in action use main function below:

Hide   Copy Code

int main(int argc, char\*\* argv){

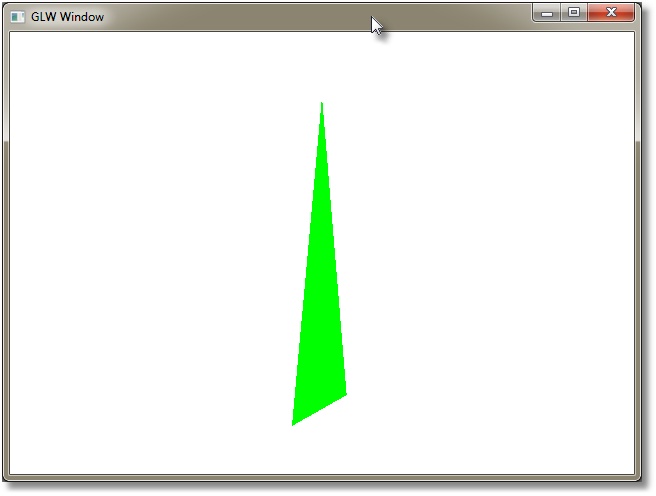
glwSetUpdateMode(GLW\_CONTINOUS\_LOOP);

glwSetUpdateFunc(draw);

return glwMainLoop();

}

Now you should see a green triangle constatntly rotating around Y-axis:



*Triangle rotating around Y-axis.*

Rendering on demand

To trigger rendering while in event mode use:

Hide   Copy Code

glwRedisplay()

As you might guess it is glw equivalent of glutPostRedisplay() (for those used to GLUT).  If you run polling (continuous) loop, you don’t have to touch this one, but if you run event-driven loop, you will need it to tell glw when to render. GLW renders on standard system triggers such as resize, but it won’t render on key press or mouse movement. Generally you don’t wish to render on every mouse movement, but sometimes you will use mouse to do stuff like rotations or picking and moving objects around in the scene. After say rotation, you would call redisplay callback to trigger new render of the scene. It is impossible to know in advance how mouse is going to be used, a 3d modeler uses mouse in different way than a 3d shooter; that choice is best left to application.

This function does nothing when you are not in event-driven mode so it’s ok to have it in code that works in both modes.

To test updating modes in gears app, press ‘E’ to put GLW in event-mode or ‘C’ to go back to continuous rendering.

To recap, these are all windowing functions available (see header file for more details):

Hide   Copy Code

glwMainLoop

glwSetResizeFunc

glwSetUpdateMode

glwSetUpdateFunc

glwSet2DMode

glwSet3DMode

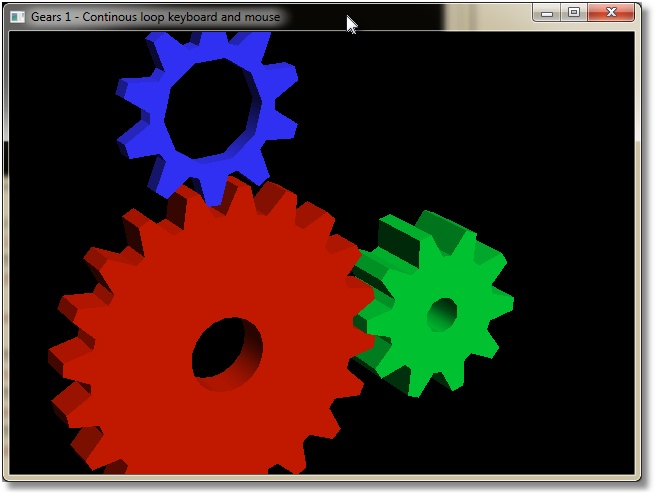
glwSetWindowSize

glwGetWindowSize

By default you just need to setup your render callback and call in main loop.

GL Gears example in demo

Well I could go on with just a green triangle, but it is more fun to see something more complex.  As mentioned I have adaptated Gears app as found in GLUT 3.6 code. Both gears1 and gears2 projects found in solution, are 100% identical with each other. Only reason I have two projects, was to test size of executable when linked with dll versus static linking with source code. So you may peek in either one, but you don’t have to look at both.



*OpenGL gears running in a GLW window*

Input

I have tried to do input as simple as I could too. I use input just occasionally to manipulate objects in the scene, and it reflects in capabilities of GLW. Keyboard handling is limited to physical keys. It means you can listen to what keys are pressed, like Enter, PageUp, PageDown, A, B, C and so on. But you can’t get things like ‘@,$,a-z’ (ascii characters). I just didn’t found it very usefull to move stuff around in a 3d scene with say ‘@’. If you really wish to be able to write advanced text with glw, you will have to add code to handle WM\_CHAR yourself, I am afraid. So for instance glw is \*not\* the app you would do your text editor on (but maybe a shooter?).

Mouse is handled quite well; you can listen on left, middle and right buttons, mouse motion and wheel motion.

Keyboard

Signature for keyboard callback is

Hide   Copy Code

void glwKeyboardFunc(short key, short event)

Key will tell you what key is pressed down, while event will tell you if it was pressed or released.

Usually you would ignore key up, and turn all keyboard handling into “keypress” kind-a model. But if you wish to do some more advanced stuff like doing something while you hold down a key, possibility is there.

I am using shorts instead of ints in hope it will be little less pushing and popping on stack (2 shorts are 4 bytes, 2 ints are 8 bytes). Well I guess it does not save much, but I always try to get away with as little as possible.

In gears I do keyboard like this:

Hide   Copy Code

void key(short k, short event){

if(event == GLW\_KEYUP) *// act only on keydown*

return;

switch (k) {

case 'A': showfps = ~showfps; break;

case 'X': view\_rotz += 5.0; break;

case 'Z': view\_rotz -= 5.0; break;

case 'E': glwSetUpdateMode(GLW\_EVENT\_LOOP); break;

case 'C': glwSetUpdateMode(GLW\_CONTINOUS\_LOOP); break;

case 'F': glwToggleFullscreen(); break;

case 'V':

vsync = !vsync;

glwSetVSync(vsync);

break;

case 'H':

view\_movx = 0;

view\_movy = 0;

view\_movz = 0;

break;

case 27: */\* Escape \*/*

case 'Q':

glwExit();

break;

}

}

Keys are abstracted to be platform-independent; they are usually called like keys are normally called, but all in uppercase and with prefix GLW\_, for example:

Hide   Copy Code

GLW\_DELETE

GLW\_INSERT

GLW\_ESCAPE

GLW\_SHIFT

GLW\_ALT

GLW\_CTRL

( ... )

Actually what is “normal” depends from person to person, so you will have to take a peek in header for full list of key names, but you got the idea.

Key-up event is called GLW\_KEYUP, and key-down event is called GLW\_KEYDOWN. There is almost no key processing in GLW; keys are pretty much mapped 1:1 to their counterparts in win32 (VK\_). Alt and “extended keys” handling in windows is a bit miss moaner that needed some special care, but aside from that, keys are just passed through to the application. Well fastest processing is one you don’t have to do Smile | :) .

That’s about it; you can’t do stuff like case ‘q’ or case ‘a’ - it won’t work. ASCII A-Z (as used in example) and 0-9 works only because OS assigns same values as ASCII to corresponding virtual keys.

Mouse buttons

Mouse is a little bit more versatile than keyboard. For starter, left, right and middle mouse buttons are handled just like keyboard. Signature for the callback is:

Hide   Copy Code

void glwMouseFunc(short x, short y, short event, short modifiers)

X and Y holds naturally coordinate for the cursor. Coordinates are relative to client-space of the window which means area of the window where OpenGL draws, but not frames, borders or other decorations around that area.

Usually one is rather interested in deltas (amount of movement) between last read and current one. For that reason it does not matter if they are in screen or window coordinates. If you for some reason need screen space coordinates, it is trivial to provide conversion; either calculate it yourself or by calling ScreenToClient and ClientToScreen  from Win32 api. Both solutions require you to modify glw itself (code is found in sglw.c).

Another twist is that windows calculate coordinates relative to upper left corner, while GLW uses lower left corner as origin, so if you need to translate position of the mouse in window, to GLW you need to take that in account, also be aware that cursor coordinates are in 2d space of window, while OpenGL thinks in origin of you 3D scene (world).

*Event* means same as for keyboard, either a button was up or down. To find which button is pressed you have to look into modifiers. Possibilities are:

Hide   Copy Code

GLW\_LBUTTON

GLW\_RBUTTON

GLW\_MBUTTON

Beside those keys it might be of interest to know if user is holding down a shift, ctrl or alt key. You can get that information from *modifiers* parameter. Modifiers are packed as bit flags, so to test which one is pressed you do something like

Hide   Copy Code

if(modifiers & GLW\_KEY)

/do something here /

There is an example in code for gears that will print out various presses and modifiers to stdout:

Hide   Copy Code

void mpdbg(short x, short y, short e, short m){

if(e == GLW\_KEYDOWN)

puts("Event: mouse key down");

if(e == GLW\_KEYUP)

puts("Event: mouse key up");

printf("Modifiers: ");

if(m & GLW\_ALT)

printf("alt ");

if(m & GLW\_CTRL)

printf("ctrl ");

if(m & GLW\_SHIFT)

printf("shift ");

printf("\nButtons: ");

if(m & GLW\_LBUTTON)

printf("left ");

if(m & GLW\_RBUTTON)

printf("right ");

if(m & GLW\_MBUTTON)

printf("middle ");

puts("");

}

Observe that *event* in this function is only related to left, right and middle mouse buttons, not to modifier keys (control, alt and shift).

Mouse motion

Mouse motion is called when mouse moves inside the window. The signature is somewhat simpler than previous one:

Hide   Copy Code

void glwMouseMoveFunc(short x,short y, short modifiers);

There are only x and y coordinates, and list of modifiers. The list is same as in previous example, but there is no event (key up or down). If a button is pressed, then glwMouseFunc is called instead, not this one.

Here is what I do in gears to enable simple translation and rotation with mouse:

Hide   Copy Code

void onmousemove(short x, short y, short m){

int dx, dy;

if(oldx == 0) oldx = x;

if(oldy == 0) oldy = y;

dx = (x-oldx), dy = (y-oldy);

if(m & GLW\_ALT){

if(m & GLW\_LBUTTON){

view\_rotx += dx;

view\_roty += dy;

}

else if(m & GLW\_RBUTTON){

view\_movx += dx\*0.01f;

view\_movy -= dy\*0.01f;

}else if(m & GLW\_MBUTTON){

view\_movz += (dx+dy)\*0.5f\*0.1f;

}

glwRedisplay();

}

oldx = x; oldy = y;

}

I use Alt key as a modifier. Hold down alt key and press left mouse button then move mouse to rotate the camera (or model, whichever you prefer). Hold down right button and alt key and move mouse to translate in x and y, which will create effect somewhat similar to panning the scene. Finally if you press middle mouse button and alt, and move around mouse, you are translating in z coordinate which gives effect of zooming in and out with camera.

Don’t hang–up on that code, it is there just to illustrate how to read buttons from the mouse not to show how to implement a camera in OpenGL.

Mouse wheel

Mouse wheel function is called when you turn mouse wheel. Unlike x and y coordinate for the cursor, the wheel is not sending coordinates. It sends ticks. So when you turn wheel a bit, you get one tick and a windowing event. When you turn it a bit more you get another tick and so on. Continuous moving of wheel will also not produce continuous coordinates, but a discrete value on every event. Tick has size and name. Its name is WHEEL\_DELTA, a term you can google on to learn more about how wheel events are handled.

Size is either +120 or -120, depending on the direction of movement (+ from you and – towards you). So you will never get something like 240, 360 and so on. Instead of 360; you get 3 updates a 120 each. It means that we really are interested just in sign of the movement, not amount. For that reason, our z will always be either +1, or -1.  Here is an example of usage from gears:

Hide   Copy Code

void onmousewheel(short x, short y, short z, short m){

float amount = 10\*(float)z;

if(m & GLW\_CTRL)

view\_rotz += amount;

if(m & GLW\_SHIFT)

view\_roty += amount;

if(m & GLW\_ALT)

view\_rotx += amount;

glwRedisplay();

}

I use wheel to rotate the view, but this time I constrain rotation to x, y and z axis depending on modifier key pressed.

Initialization and cleanup

Usually you will need to load textures and maybe some other resources for which you need OpenGL context. Well until main loop has started there is no context. Instead you may register a function to be called once the OpenGL is created and ready for use. Function signature is

Hide   Copy Code

void glwInit()

and setter is

Hide   Copy Code

void glwSetInitFunc(glwInitFunc cb)

Don’t confuse it with initialization of glw; glw need nothing to get initialized! Init function is convenient place where you will put your own initialization tasks, at least those that require GL context. In init you would prepare texture, buffers, shaders, modify state and so on.

Analogous to initialization you need to perform cleanup when you are done. For that purpose there is glwExitFunc set with glwSetExitFunc. Just like init is safe place to initiate gl stuff, so is exit safe place to clean it up. Exit function is called while OpenGL context is still active. Doing any cleanup after main loop has exited will most likely result in a crash, since GL context will be destroyed at that point.

Finally there is

Hide   Copy Code

glwExit()

to terminate GLW app. It is an internal function to replace standard *exit()*

It should not be confused with glwExitFunc. *glwExit* is GLWs API function that in turn will call your callback glwExitFunc to clean up resources allocated by the applicaiton.

It also brings issue of terminating an glw app: don’t call exit(0), call glwExit to make sure OpenGL resources are properly released. For same reason, don’t rely on atexit() to clean up any of OpenGL. Instead register your callback with glwSetExitFunc() and GLW will call it when appropriate.

Modern Opengl

As known OpenGL board has changed a lot how OpengGL API works. I don’t really agree with all changes they introduce, mostly those about deprecation, but who am I to judge? Smile | :) I have thrown in support for initialization of modern context (3.xx and later). If you believe new ways are better, you might do something like this:

Hide   Copy Code

int main(int argc, char\*\* argv) {

int attribs[] =

{

WGL\_CONTEXT\_MAJOR\_VERSION\_ARB, 3,

WGL\_CONTEXT\_MINOR\_VERSION\_ARB, 2,

WGL\_CONTEXT\_FLAGS\_ARB,

WGL\_CONTEXT\_FORWARD\_COMPATIBLE\_BIT\_ARB,

WGL\_CONTEXT\_PROFILE\_MASK\_ARB,

*//WGL\_CONTEXT\_CORE\_PROFILE\_BIT\_ARB,*

WGL\_CONTEXT\_COMPATIBILITY\_PROFILE\_BIT\_ARB,

0

};

glwSetGLAttribs(attribs);

glwSetInitFunc(init);

glwSetExitFunc(clean);

glwSetKeyboardFunc(key);

glwSetUpdateFunc(render);

return glwMainLoop();

}

Pass in your attribs array as argument to glwSetGLAttribs and GLW will do its best to honor your request.

As a note, if I ask for GL version from driver I will always get latest at least on my card (Nvidia 560ti), no matter what attributes I pass to OpenGL. I get latest even with “old ways”, so why would I bother? Well just because it works that way on my card does not mean it will work same on some other driver.

Miscelaneous

That was pretty much the whole framework Smile | :) , but there are few extras I haven’t touched yet.

GLW window can be set in and out of full screen mode with call to

Hide   Copy Code

glwToggleFullscreen()

In gears app you can test it if you press ’F’.

Note it does not trigger real video-mode change. It just puts window on top and hides all window decorations. Yeah – I know, it is old GLUT ‘game-mode’, but I am quite fine with it. Switch is fast and almost for free in terms of code. To add real video-mode handling would require much more code and would be a bloat for something as simple as GLW.

There is issue with full screen. Sometimes, but just sometimes, when an app is switching into full screen for the very first time, the screen may go black (window is not updated). I have seen it only very few times, and cannot reproduce it on-demand actually, so I can’t hunt down the cause. Switch out and in again fixes problem; maybe some other update of the screen would solve it as well, but as said I am unable to reproduce it so it is difficult for me to say.

Last function is

Hide   Copy Code

glwSetVsync()

It takes values of 1 or 0 to enable or disable vertical sync for the graphics card. By default graphics card is setup to render in synchronization with monitors vertical refresh rate. It render only as many frames as monitor can display. And really there is no need for more, unless you wish to benchmark of course. I am not sure what one actually measure in gears demo; I guess we have to send a lot of polygons to graphics card before it starts to measure performance of the card. Also those numbers we get out from gears demo are probably just telling how fast system can switch its buffers. Anyway, if you wish to see tearing on the screen, press ‘V’ in gears demo.

As a funny note – did you know you could set graphics card to half of the monitor refresh rate? I didn’t, but I found out by mistake. Pass in ~1 to vsync, and the fps will go half the rate (at least does for me, Gf560 ti,24’’ flatscreen at 60hz refresh rate). Well I know that vsync means that graphics card can only refresh in multiples of monitor refresh rate, but I didn’t know I can turn it off or on in those multiples manually.

Internals

I am not going to write much about internals; it is so small and simple framework, that discussion would probably turn into introductive Win32 tutorial Smile | :) . However for those that are used to windows programming, there are few notes to mention.

For the first GLW gets rid of WinMain. You don’t need WinMain in windows application; it is just a convention, and it is myth that this convention is useful. Really Smile | :) ; winmain just makes your code less portable and adds additional idiom to remember, while not giving you any special value. You can obtain all arguments passed to WinMain through Win32 API if you need them, and you can configure your linker to produce windowed application anyway. Maybe there is something deeper about WinMain I miss, in that case please inform me.

Main loop is done little differently than normal. Usually frameworks implement one or another kind of loop (either GetMessage or PeekMessage); but I have coded it in a way that makes it possible to switch between those.

Points of Interest

Included with this framework is a high-resolution timer code that I have found and adapted from an [article](http://software.intel.com/en-us/articles/measure-code-sections-using-the-enhanced-timer/) on Intel's site. It is found in etimer.h and etimer.c. I believe that code was free, but I include link to both original article and to [licence](http://software.intel.com/en-us/articles/code-samples-license-2/?target=http://www.intel.com/cd/ids/developer/asmo-na/eng/282380.htm" \t "_blank) in etimer.h. It is also a *drop-in* code you can use independently of this library.

History

2012-04-23 Article and source published

2012-04-25 Fixed a bug with initial window size in glw.c.