Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

Window and OpenGL context

Before you can start drawing things, you need to initialize OpenGL. This is done by creating an OpenGL context, which is essentially a state machine that stores all data related to the rendering of your application. When your application closes, the OpenGL context is destroyed and everything is cleaned up.

The problem is that creating a window and an OpenGL context is not part of the OpenGL specification. That means it is done differently on every platform out there! Developing applications using OpenGL is all about being portable, so this is the last thing we need. Luckily there are libraries out there that abstract this process, so that you can maintain the same codebase for all supported platforms.

While the available libraries out there all have advantages and disadvantages, they do all have a certain program flow in common. You start by specifying the properties of the game window, such as the title and the size and the properties of the OpenGL context, like the anti-aliasing level. Your application will then initiate the event loop, which contains an important set of tasks that need to be completed over and over again until the window closes. These tasks usually handle window events like mouse clicks, updating the rendering state and then drawing.

This program flow would look something like this in pseudocode:

```
#include <libraryheaders>

int main()
{
    createWindow(title, width, height);
    createOpenGLContext(settings);

    while (windowOpen)
    {
        while (event = newEvent())
            handleEvent(event);

        updateScene();

        drawGraphics();
        presentGraphics();
    }

    return 0;
}
```

When rendering a frame, the results will be stored in an offscreen buffer known as the *back buffer* to make sure the user only sees the final result. The `presentGraphics()` call will copy the result from the back buffer to the visible window buffer, the *front buffer*. Every application that makes use of real-time graphics will have a program flow that comes down to this, whether it uses a library or native code.

By default, libraries will create an OpenGL context that supports the legacy functions. This is unfortunate, because we're not interested in those and they may become unavailable at some point in the future. The good news is that it is possible to inform the drivers that our application is ready for the future and does not depend on the old functions. The bad news is that at this moment only the GLFW library allows us to specify this. This little shortcoming doesn't have any negative consequences right now, so don't let it influence your choice of library too much, but the advantage of a so-called core profile context is that accidentally calling any of the old functions results in an invalid operation error to set you straight.

Supporting resizable windows with OpenGL introduces some complexities as resources need to be reloaded and buffers need to be recreated to fit the new window size. It's more convenient for the learning process to not bother with such details yet, so we'll only deal with fixed size (fullscreen) windows for now.

Setup

> Instead of reading this chapter, you can make use of the OpenGL quickstart boilerplate, which makes setting up an OpenGL project with all of the required libraries very easy. You'll just have to install SOIL separately.

The first thing to do when starting a new OpenGL project is to dynamically link with OpenGL.

- **Windows**: Add `opengl32.lib` to your linker input
- **Linux**: Include `-lGL` in your compiler options
- **OS X**: Add `-framework OpenGL` to your compiler options

> Make sure that you do **not** include `opengl32.dll` with your application. This file is already included with Windows and may differ per version, which will cause problems on other computers.

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

The rest of the steps depend on which library you choose to use for creating the window and context.

Libraries

There are many libraries around that can create a window and an accompanying OpenGL context for you. There is no best library out there, because everyone has different needs and ideals. I've chosen to discuss the process for the three most popular libraries here for completeness, but you can find more detailed guides on their respective websites. All code after this chapter will be independent of your choice of library here.

SFML

SFML is a cross-platform C++ multimedia library that provides access to graphics, input, audio, networking and the system. The downside of using this library is that it tries hard to be an all-in-one solution. You have little to no control over the creation of the OpenGL context, as it was designed to be used with its own set of drawing functions.

SDL

SDL is also a cross-platform multimedia library, but targeted at C. That makes it a bit rougher to use for C++ programmers, but it's an excellent alternative to SFML. It supports more exotic platforms and most importantly, offers more control over the creation of the OpenGL context than SFML.

GLFW

GLFW, as the name implies, is a C library specifically designed for use with OpenGL. Unlike SDL and SFML it only comes with the absolute necessities: window and context creation and input management. It offers the most control over the OpenGL context creation out of these three libraries.

Others

There are a few other options, like freeglut and OpenGLUT, but I personally think the aforementioned libraries are vastly superior in control, ease of use and on top of that more up-to-date.

SFML

The OpenGL context is created implicitly when opening a new window in SFML, so that's all you have to do. SFML also comes with a graphics package, but since we're going to use OpenGL directly, we don't need it.

Building

After you've downloaded the SFML binaries package or compiled it yourself, you'll find the needed files in the `lib` and `include` folders.

- Add the `lib` folder to your library path and link with `sfml-system` and `sfml-window`. With Visual Studio on Windows, link with the `sfml-system-s` and `sfml-window-s` files in `lib/vc2008` instead.
- Add the `include` folder to your include path.

> The SFML libraries have a simple naming convention for different configurations. If you want to dynamically link, simply remove the `-s` from the name, define `SFML_DYNAMIC` and copy the shared libraries. If you want to use the binaries with debug symbols, additionally append `-d` to the name.

To verify that you've done this correctly, try compiling and running the following code:

```
#include <SFML/System.hpp>

int main()
{
    sf::sleep(sf::seconds(1.f));
    return 0;
}
```

It should show a console application and exit after a second. If you run into any trouble, you can find more detailed information for Visual Studio, Code::Blocks and gcc in the tutorials on the SFML website.

Code

Start by including the window package and defining the entry point of your application.

```
#include <SFML/Window.hpp>

int main()
{
    return 0;
}
```

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

A window can be opened by creating a new instance of `sf::Window`. The basic constructor takes an `sf::VideoMode` structure, a title for the window and a window style. The `sf::VideoMode` structure specifies the width, height and optionally the pixel depth of the window. Finally, the requirement for a fixed size window is specified by overriding the default style of `Style::Resize|Style::Close`. It is also possible to create a fullscreen window by passing `Style::Fullscreen` as window style.

```
sf::ContextSettings settings;
settings.depthBits = 24;
settings.stencilBits = 8;
settings.antialiasingLevel = 2; // Optional

sf::Window window(sf::VideoMode(800, 600), "OpenGL", sf::Style::Close, settings);
```

The constructor can also take an `sf::ContextSettings` structure that allows you to specify the anti-aliasing level and the accuracy of the depth and stencil buffers. The latter two will be discussed later, so you don't have to worry about these yet. In the latest version of SFML, you do need to request these manually with the code above.

When running this, you'll notice that the application instantly closes after creating the window. Let's add the event loop to deal with that.

```
bool running = true;
while (running)
{
    sf::Event windowEvent;
    while (window.pollEvent(windowEvent))
    {

    }
}
```

When something happens to your window, an event is posted to the event queue. There is a wide variety of events, including window size changes, mouse movement and key presses. It's up to you to decide which events require additional action, but there is at least one that needs to be handled to make your application run well.

```
switch (windowEvent.type)
{
case sf::Event::Closed:
    running = false;
    break;
}
```

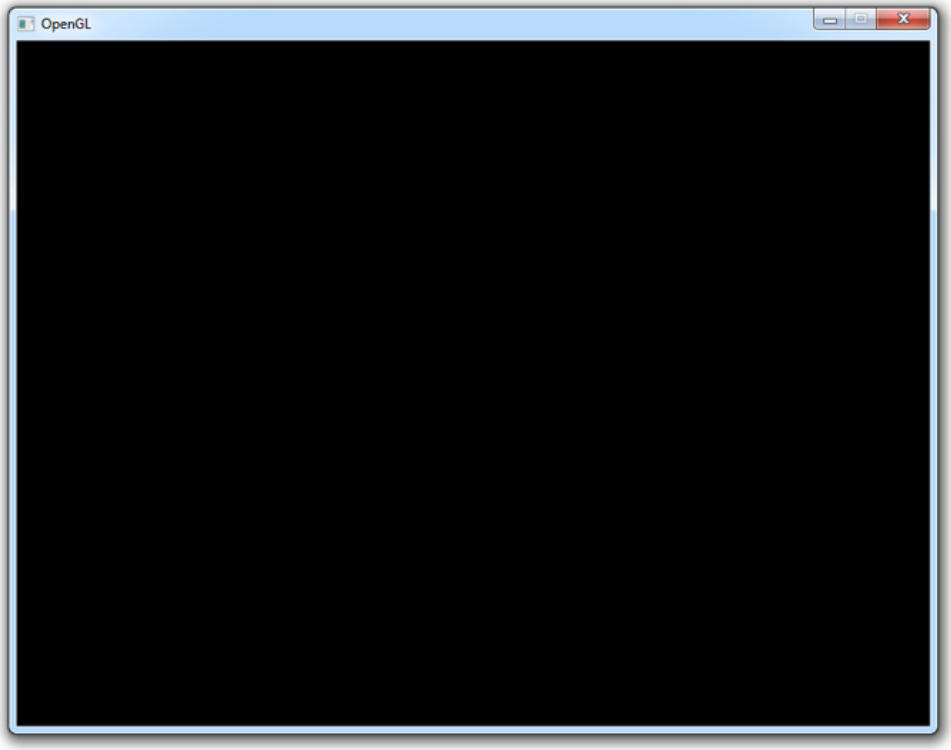
When the user attempts to close the window, the `Closed` event is fired and we act on that by exiting the application. Try removing that line and you'll see that it's impossible to close the window by normal means. If you prefer a fullscreen window, you should add the escape key as a means to close the window:

```
case sf::Event::KeyPressed:
    if (windowEvent.key.code == sf::Keyboard::Escape)
        running = false;
    break;
```

You have your window and the important events are acted upon, so you're now ready to put something on the screen. After drawing something, you can swap the back buffer and the front buffer with `window.display()`.

When you run your application, you should see something like this:

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference



Note that SFML allows you to have multiple windows. If you want to make use of this feature, make sure to call `window.setActive()` to activate a certain window for drawing operations.

Now that you have a window and a context, there's one more thing that needs to be done.

SDL

SDL comes with many different modules, but for creating a window with an accompanying OpenGL context we're only interested in the video module. It will take care of everything we need, so let's see how to use it.

Building

After you've downloaded the SDL binaries or compiled them yourself, you'll find the needed files in the `lib` and `include` folders.

- Add the `lib` folder to your library path and link with `SDL2` and `SDL2main`.
- SDL uses dynamic linking, so make sure that the shared library (`SDL2.dll`, `SDL2.so`) is with your executable.
- Add the `include` folder to your include path.

To verify that you're ready, try compiling and running the following snippet of code:

```
#include <SDL.h>

int main(int argc, char *argv[])
{
    SDL_Init(SDL_INIT_EVERYTHING);

    SDL_Delay(1000);

    SDL_Quit();
    return 0;
}
```

It should show a console application and exit after a second. If you run into any trouble, you can find more detailed information for all kinds of platforms and compilers in the tutorials on the web.

Code

Start by defining the entry point of your application and include the headers for SDL.

```
#include <SDL.h>
#include <SDL_opengl.h>

int main(int argc, char *argv[])
{
    return 0;
}
```

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

To use SDL in an application, you need to tell SDL which modules you need and when to unload them. You can do this with two lines of code.

```
SDL_Init(SDL_INIT_VIDEO);
...
SDL_Quit();
return 0;
```

The `SDL_Init` function takes a bitfield with the modules to load. The video module includes everything you need to create a window and an OpenGL context.

Before doing anything else, first tell SDL that you want a forward compatible OpenGL 3.2 context:

```
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_PROFILE_CORE);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 2);
SDL_GL_SetAttribute(SDL_GL_STENCIL_SIZE, 8);
```

You also need to tell SDL to create a stencil buffer, which will be relevant for a later chapter. After that, create a window using the `SDL_CreateWindow` function.

```
SDL_Window* window = SDL_CreateWindow("OpenGL", 100, 100, 800, 600, SDL_WINDOW_OPENGL);
```

The first argument specifies the title of the window, the next two are the X and Y position and the two after those are the width and height. If the position doesn't matter, you can specify `SDL_WINDOWPOS_UNDEFINED` or `SDL_WINDOWPOS_CENTERED` for the second and third argument. The final parameter specifies window properties like:

- *SDL_WINDOW_OPENGL* - Create a window ready for OpenGL.
- *SDL_WINDOW_RESIZABLE* - Create a resizable window.
- **Optional** *SDL_WINDOW_FULLSCREEN* - Create a fullscreen window.

After you've created the window, you can create the OpenGL context:

```
SDL_GLContext context = SDL_GL_CreateContext(window);
...
SDL_GL_DeleteContext(context);
```

The context should be destroyed right before calling `SDL_Quit()` to clean up the resources.

Then comes the most important part of the program, the event loop:

```
SDL_Event windowEvent;
while (true)
{
    if (SDL_PollEvent(&windowEvent))
    {
        if (windowEvent.type == SDL_QUIT) break;
    }

    SDL_GL_SwapWindow(window);
}
```

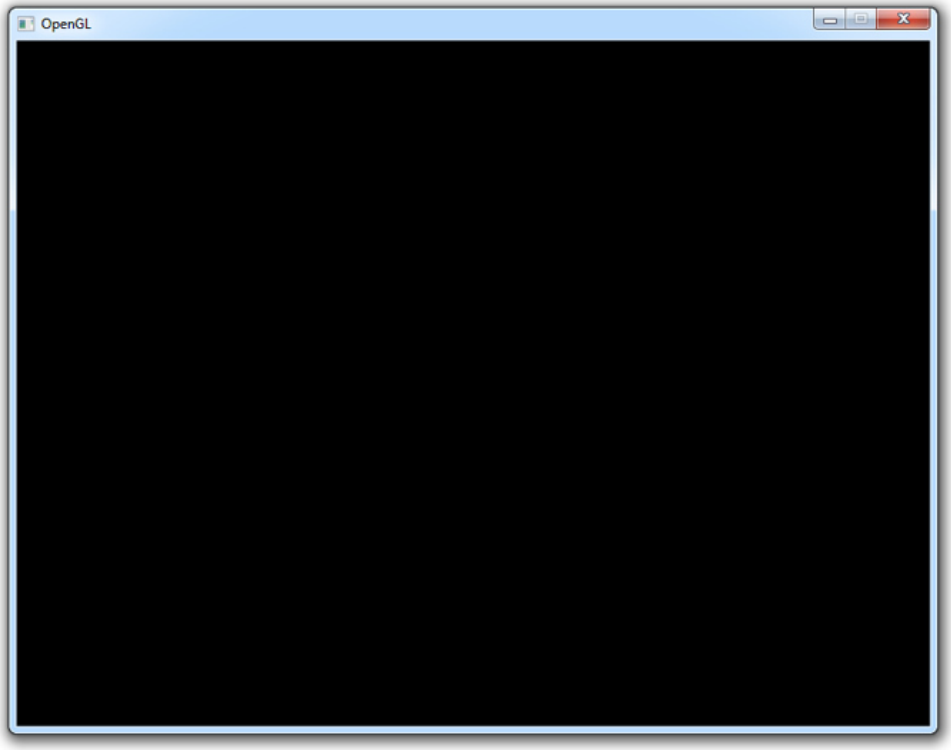
The `SDL_PollEvent` function will check if there are any new events that have to be handled. An event can be anything from a mouse click to the user moving the window. Right now, the only event you need to respond to is the user pressing the little X button in the corner of the window. By breaking from the main loop, `SDL_Quit` is called and the window and graphics surface are destroyed. `SDL_GL_SwapWindow` here takes care of swapping the front and back buffer after new things have been drawn by your application.

If you have a fullscreen window, it would be preferable to use the escape key as a means to close the window.

```
if (windowEvent.type == SDL_KEYUP &&
    windowEvent.key.keysym.sym == SDLK_ESCAPE) break;
```

When you run your application now, you should see something like this:

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference



Now that you have a window and a context, there's one more thing that needs to be done.

GLFW

GLFW is tailored specifically for using OpenGL, so it is by far the easiest to use for our purpose.

Building

After you've downloaded the GLFW binaries package from the website or compiled the library yourself, you'll find the headers in the `include` folder and the libraries for your compiler in one of the `lib` folders.

- Add the appropriate `lib` folder to your library path and link with `GLFW`.
- Add the `include` folder to your include path.

> You can also dynamically link with GLFW if you want to. Simply link with `GLFWDLL` and include the shared library with your executable.

Here is a simple snippet of code to check your build configuration:

```
#include <GLFW/glfw3.h>
#include <thread>

int main()
{
    glfwInit();
    std::this_thread::sleep_for(std::chrono::seconds(1));
    glfwTerminate();
}
```

It should show a console application and exit after a second. If you run into any trouble, just ask in the comments and you'll receive help.

Code

Start by simply including the GLFW header and define the entry point of the application.

```
#include <GLFW/glfw3.h>

int main()
{
    return 0;
}
```

To use GLFW, it needs to be initialised when the program starts and you need to give it a chance to clean up when your program closes. The `glfwInit` and `glfwTerminate` functions are geared towards that purpose.

Links

OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference


```
glfwInit();
...
glfwTerminate();
```

The next thing to do is creating and configuring the window. Before calling `glfwCreateWindow`, we first set some options.

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);

GLFWwindow* window = glfwCreateWindow(800, 600, "OpenGL", nullptr, nullptr); // Windowed
GLFWwindow* window =
    glfwCreateWindow(800, 600, "OpenGL", glfwGetPrimaryMonitor(), nullptr); // Fullscreen
```

You'll immediately notice the first three lines of code that are only relevant for this library. It is specified that we require the OpenGL context to support OpenGL 3.2 at the least. The `GLFW_OPENGL_PROFILE` option specifies that we want a context that only supports the new core functionality.

The first two parameters of glfwCreateWindow specify the width and height of the drawing surface and the third parameter specifies the window title. The fourth parameter should be set to `NULL` for windowed mode and `glfwGetPrimaryMonitor()` for fullscreen mode. The last parameter allows you to specify an existing OpenGL context to share resources like textures with. The `glfwWindowHint` function is used to specify additional requirements for a window.

After creating the window, the OpenGL context has to be made active:

```
glfwMakeContextCurrent(window);
```

Next comes the event loop, which in the case of GLFW works a little differently than the other libraries. GLFW uses a so-called *closed* event loop, which means you only have to handle events when you need to. That means your event loop will look really simple:

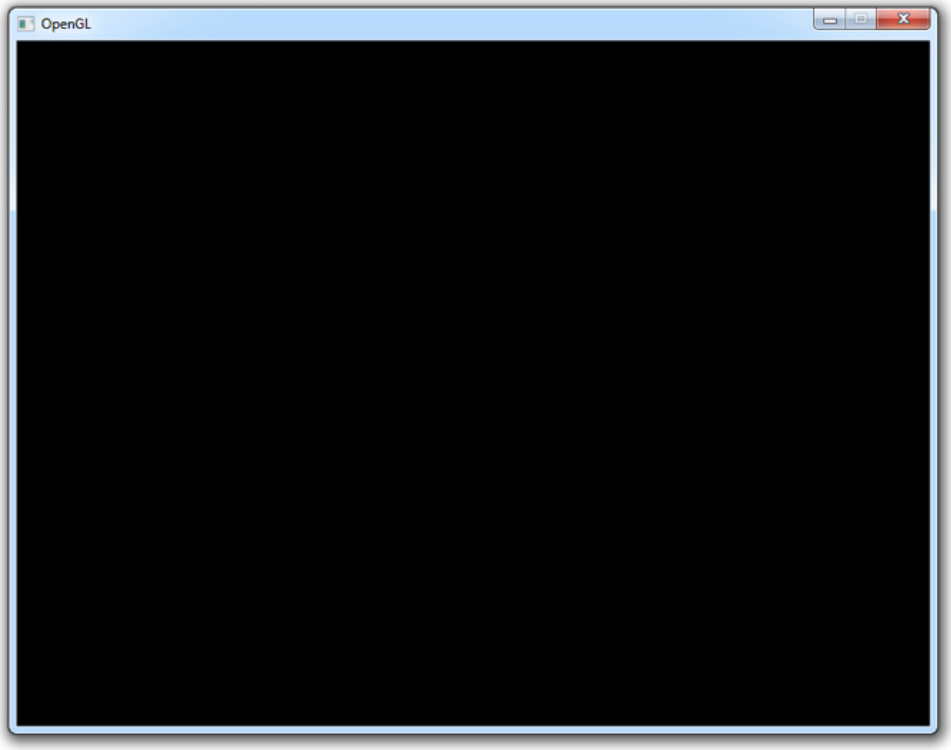
```
while(!glfwWindowShouldClose(window))
{
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

The only required functions in the loop are `glfwSwapBuffers` to swap the back buffer and front buffer after you've finished drawing and `glfwPollEvents` to retrieve window events. If you are making a fullscreen application, you should handle the escape key to easily return to the desktop.

```
if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
    glfwSetWindowShouldClose(window, GL_TRUE);
```

If you want to learn more about handling input, you can refer to the documentation.

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference



You should now have a window or a full screen surface with an OpenGL context. Before you can start drawing stuff however, there's one more thing that needs to be done.

One more thing

Unfortunately, we can't just call the functions we need yet. This is because it's the duty of the graphics card vendor to implement OpenGL functionality in their drivers based on what the graphics card supports. You wouldn't want your program to only be compatible with a single driver version and graphics card, so we'll have to do something clever.

Your program needs to check which functions are available at runtime and link with them dynamically. This is done by finding the addresses of the functions, assigning them to function pointers and calling them. That looks something like this:

> Don't try to run this code, it's just for demonstration purposes.

```
// Specify prototype of function
typedef void (*GENBUFFERS) (GLsizei, GLuint*);

// Load address of function and assign it to a function pointer
GENBUFFERS glGenBuffers = (GENBUFFERS)wglGetProcAddress("glGenBuffers");
// or Linux:
GENBUFFERS glGenBuffers = (GENBUFFERS)glXGetProcAddress((const GLubyte *) "glGenBuffers");
// or OSX:
GENBUFFERS glGenBuffers = (GENBUFFERS)NSGLGetProcAddress("glGenBuffers");

// Call function as normal
GLuint buffer;
glGenBuffers(1, &buffer);
```

Let me begin by asserting that it is perfectly normal to be scared by this snippet of code. You may not be familiar with the concept of function pointers yet, but at least try to roughly understand what is happening here. You can imagine that going through this process of defining prototypes and finding addresses of functions is very tedious and in the end nothing more than a complete waste of time.

The good news is that there are libraries that have solved this problem for us. The most popular and best maintained library right now is *GLEW* and there's no reason for that to change anytime soon. Nevertheless, the alternative library *GLEE* works almost completely the same save for the initialization and cleanup code.

If you haven't built GLEW yet, do so now. We'll now add GLEW to your project.

- Start by linking your project with the static GLEW library in the `lib` folder. This is either `glew32s.lib` or `GLEW` depending on your platform.
- Add the `include` folder to your include path.

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

Now just include the header in your program, but make sure that it is included before the OpenGL headers or the library you used to create your window.

```
#define GLEW_STATIC
#include <GL/glew.h>
```

Don't forget to define `GLEW_STATIC` either using this preprocessor directive or by adding the `-DGLEW_STATIC` directive to your compiler command-line parameters or project settings.

> If you prefer to dynamically link with GLEW, leave out the define and link with `glew32.lib` instead of `glew32s.lib` on Windows. Don't forget to include `glew32.dll` or `libGLEW.so` with your executable!

Now all that's left is calling `glewInit()` after the creation of your window and OpenGL context. The `glewExperimental` line is necessary to force GLEW to use a modern OpenGL method for checking if a function is available.

```
glewExperimental = GL_TRUE;
glewInit();
```

Make sure that you've set up your project correctly by calling the `glGenBuffers` function, which was loaded by GLEW for you!

```
GLuint vertexBuffer;
glGenBuffers(1, &vertexBuffer);

printf("%u\n", vertexBuffer);
```

Your program should compile and run without issues and display the number `1` in your console. If you need more help with using GLEW, you can refer to the website or ask in the comments.

Now that we're past all of the configuration and initialization work, I'd advise you to make a copy of your current project so that you won't have to write all of the boilerplate code again when starting a new project.

Now, let's get to drawing things!

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

Joseph Surley ➚ elfenlaid • 3 months ago
OH MY GOD

I had gotten rid of the GLFW_OPENGL_FORWARD_COMPAT hint by accident and glfwCreateWindow was failing every time. THANK YOU
Reply • Share ›

Julien ➚ elfenlaid • 2 years ago
thanks a lot
Reply • Share ›

Martin • 4 years ago
It took me some time to figure out that I have to call glewInit after creating a window and not the other way :).
10 Reply • Share ›

DrDaz ➚ Martin • 2 years ago
Indeed. You have to initialise GLEW after you have given OpenGL a context. For example, with the GLFW library the glewInit() function should be written after the glfwMakeContextCurrent() function, other wise the GLuint vertexBuffer in the code snippet above is uninitialised when you pass it to the screen for printing - you'll get an access violation at runtime.
Reply • Share ›

Ruben Verdoes • 4 years ago
With SDL and GLEW, compiling for x64 on Windows 7 with Visual Studio 11, I had to use GLEW dynamically linked and I had to #define NO_SDL_GLEXT.

Also, to fix the "entry point must be defined" linker error, I had to set Project Properties -> Linker -> System -> Subsystem to Windows. This does mean you no longer get to see a console, but at least it will compile and run.
5 Reply • Share ›

OpenGL hates me • 4 years ago
I'm getting ACCESS_DENIED errors when calling the glGenBuffers function when compiling for glew x32 on windows 7 x64 with visual studio 10. Tried dynamic linking and also static linking. Haven't tried glew x64 yet though.

I haven't found a solution to it though.
8 Reply • Share ›

Anonymous ➚ OpenGL hates me • 4 years ago
Had the same problem. Solution is to call the Buffer code after opening the window ("glfwOpenWindow").
4 Reply • Share ›

ososhyong ➚ OpenGL hates me • 3 years ago
Same, except with SDL on VS 2012. I also had to add the glide2x.dll library and now, it gives me the following errors:
gd error (glide): Genport I/O initialization failure.
gd error (glide): grSstSelect: non-existent SST.

Anyone else having problems with glGenBuffers(1, &vbo); using SDL/GLEW?

Also, wow, didn't realize this tutorial was that old. Saw it linked in a Hacker News thread.
1 Reply • Share ›

ososhyong ➚ ososhyong • 3 years ago
Solved my problem, included the opengl32.dll in the executable's folder. I'm guessing it was conflicting with the SDL2 GL context.
1 Reply • Share ›

It's so hard to quickly commen ➚ ososhyong • 2 years ago
Fez!
1 Reply • Share ›

Stan ➚ OpenGL hates me • 3 years ago
Getting the same
Reply • Share ›

anirudhrata ➚ OpenGL hates me • 4 years ago
Try giving administrator rights to the executable.

Introduction
Context creation
Drawing polygons
Textures
Transformations
Depth and stencils
Framebuffers
Geometry shaders
Transform Feedback

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

• Reply • Share ›

Ron K • 4 years ago

i keep on getting error LNK2019: unresolved external symbol when i try to run the test code for GLFW...any advice? ive linked everything put the libraries and headers and dll in the right place...im using visual c++2010

4 • Reply • Share ›

Derry Holt ➚ Ron K • 3 years ago

Can you paste the full error?

If it has an underscore before glfw:

error LNK2019: unresolved external symbol _glfwInit referenced in function _main

This could be because you're using x64 libs whilst building for x86 (Win32).

Select the drop down box that reads Win32->Configuration Manager->Active Solution Platform->New...

From here I chose ARM for the first box and copied settings from Win32. Compiled just fine after this.

• Reply • Share ›

Thomas Lee ➚ Derry Holt • 2 years ago

So I followed this and got another error saying:
error MSB8022: Compiling Desktop applications for the ARM platform is not supported.
Any way you can help me out?
Thanks in advance!

http://gyazo.com/91cac44699e99...

1 • Reply • Share ›

Tim ➚ Ron K • 3 years ago

Hey I'm receiving the same problem...do you know how to resolve it?

• Reply • Share ›

DrDaz ➚ Tim • 2 years ago

Just in case other people are getting the same Linker error with MSVC (LNK2019 in debug, or LNK2001 in release) ensure you have included opengl32.lib in the Linker > Input tab of the property sheets. This is mentioned in the Tutorial above but people tend to skim read - which is exactly what I did.

• Reply • Share ›

Derry Holt ➚ Tim • 3 years ago

Read my reply to Ron K, see if that helps? :)

• Reply • Share ›

Alexander Overvoorde Mod ➚ Tim • 3 years ago

Could you send me your project in a zip? alexander@while.io

• Reply • Share ›

Guest • 4 years ago

Hey, thanks for making this guide. I am having an issue setting up GLFW on my mac os x however. I am running a fully updated mac os x 10.8, so I am assuming that I have the most up to date mac-compatible OpenGL version and I have compiled the GLFW libraries using homebrew (OS X package manager).

When I compile the code provided for testing GLFW with

| g++ a.cpp -lglfw -framework OpenGL -framework Cocoa -framework IOkit

, I get no errors, but when I run the program, no console application is created. I've even tried increasing the the time glfwSleep time to verify that it is not there. Any ideas?

3 • Reply • Share ›

silverhammermba • 4 years ago

I tried this out on Arch Linux using GLFW and ran into a problem where the window would instantly close after opening but give no error. After a bit of Googling I figured out that my graphics card doesn't support OpenGL 3.2 and glfwOpenWindow was quietly failing.
I think it would help if you pointed out that the return values of glfwInit, glfwOpenWindow, and glewInit can all be checked to see if they were successful.
It would also be nice if there were a concrete way to determine the highest version of OpenGL your graphics card can support, rather than just trying to find it listed somewhere online.

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

3 • Reply • Share ›

Rafaël Kooi ↗ silverhammermba • 3 years ago
The best way would be to create a compatibility window and then get a pointer to the glGetString function. Then you can print your maximum GL version and GLSL with "cout << "OpenGL version: " << glGetString(GL_VERSION) << ", GLSL version: " << glGetString(GL_SHADING_LANGUAGE_VERSION) << endl;".
This should always print the highest available version in a compatibility profile.
2 • Reply • Share ›

Matthew Fowler ↗ silverhammermba • 4 years ago
Agreed. I think it'd also be nice to see a full source code at the end for consumption. The small blocks of code are great, but without full context - it can be sort of confusing to a newbie like myself.
2 • Reply • Share ›

EvanR ↗ silverhammermba • 2 years ago
GLFW initialization stuff might fail and give no reason why. Unless you install an error callback before doing anything. Even then the error call doesn't seem to execute until you call glfwPollEvents. So glfwPollEvents if any initialization commands return an odd status code. Then you can see an error message.
• Reply • Share ›

goose • 2 years ago
Some things to mention about SDL2:
You can do your rendering in a separate thread by doing the calls to OpenGL and SDL_GL_Create/DeleteContext & SDL_GL_SwapWindow in that thread as well.
You can use SDL_GL_ProcAddress to grab the function pointers without having to worry about compatibility between platforms, which is useful if you don't want to rely on GLEW.
2 • Reply • Share ›

Guest • 2 years ago
I literally spent 3 hours trying to get GLEW to work, it just won't, and I give up at this point. I set up SDL on Code::Blocks on my 64bit Windows 7 PC and it worked like a charm. Then came GLEW... I downloaded the prebuilt Windows files, included and linked the files properly and it didn't work. some "undefined reference to glewExperimental or wglGetProcAddress@4" errors (it was always something like that, but after a while i lost track which setup caused which errors) So I tried the dynamic library. It didn't work. So I tried the 64 bit variant, just to be sure. Did not work. So i googled. I read alot about compiling with MinGW, so i though, that sounds about right, let's try that. So I went through that whole annyoing process of installing MinGW and MSYS to build the source files. After a while I got that to work and I got myself some freshly compiled libs and headers. Only these libs ended on .a, but I still tried it. It didn't work. Nothing fucking worked and at this point I don't even have the energy to try to get it to work. I'll have a look at the official and unofficial OpenGL SDKs and if they don't work, fml. I'll go back to playing games.
2 • Reply • Share ›

dario90 ↗ Guest • 2 years ago
hi, I had the same problem. In CodeBlocks go to Settings - Compiler - Linker settings and add the files glew32.lib and glew32s.lib. Use the absolute path just in case.
• Reply • Share ›

GOWRI SANKAR ↗ dario90 • a year ago
No, CodeBlocks uses MinGW which means .a files are needed and not .lib files. Guest mentioned he took the route of compiling the GLEW source and producing the .a files with MinGW, but he mentioned it didn't work either. I wonder what happened exactly with that.
• Reply • Share ›

Tim Shen • 3 years ago
Hi, I keep getting the compiling error:
Error 1 error C3861: 'wglGetProcAddress': identifier not found

On VS 2013.
I've followed all the steps, that is I've linked the glew32.lib, glew32s.lib and opengl32.lib files by placing it in my VS/lib folder and for good measure I also included it in the linker. I have the glew32.dll as well as opengl32.dll in the same folder as the dll as well as system32.
I have the proper #include and #define in my code...I can't figure out what's causing this error, any help would be greatly appreciated!
2 • Reply • Share ›

AbdElHameed ↗ Tim Shen • 3 years ago
How did you fix it???

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

• Reply • Share ›

Tim ➚ AbdElHameed • 3 years ago
Hey,
I'm sorry I can't exactly remember the solution I had to fixing this, it's been a while...
I believe I fixed it by experimenting with different ways of linking the .dll's to the project. Do a google search on it.
I think there are about three ways of linking. One is by using the linking options on visual studio. (which didn't work for me). And then another is linking through windows32 folder (which i believe also didn't work for me). And finally you can link them by actually including the .dll files in the visual studio folder path. (which I think solved the issue for me)
Again, it's been a while since I did this so I'm not exactly certain if this is how I solved it. Do some research on google and hopefully you'll get it working!
1 • Reply • Share ›

AbdElHameed ➚ Tim • 3 years ago
Thanks!
• Reply • Share ›

Tim Shen ➚ Tim Shen • 3 years ago
Ok so I solved that, but now when I run my code I receive an access violation when calling glGenBuffers. I initialize glfw, set window settings and then create a window before initializing glew like the above code. Not sure where I'm going wrong.
• Reply • Share ›

Alexander Overvoorde Mod ➚ Tim Shen • 3 years ago
Don't include opengl32.dll with your application.
1 • Reply • Share ›

Tim Čas • 3 years ago
The text about only GLFW having context version selection is incorrect – SDL2 has that:
http://www.opengl.org/wiki/Tut...
2 • Reply • Share ›

Brett • a year ago
http://stackoverflow.com/quest...
1 • Reply • Share ›

Jason Ewton • a year ago
For anyone getting errors for wglGetProcAddress or any of it's related counterparts, don't type in this code snippet. The author is simply trying to show as an example what glewInit(); does for us behind the scenes.

Author, do you think you could add a note for this? It took me about an hour to figure it out and there are at least 3 other people below that had issues with it :)
1 • Reply • Share ›

Alexander Overvoorde Mod ➚ Jason Ewton • a year ago
You're right, I've added a notice to clarify that part.
2 • Reply • Share ›

Rich Evans • 2 years ago
I followed this though on Linux (Ubuntu 14.04) and found a fair few bits of Linux specific config are missing from this guide to get it working, at least using glfw for context creation.

Some of these issues may be ubuntu/g++ version specific (gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1)) but nowhere did I see the guide mention linking to X11 or link ordering. To get it working I compiled with:

`g++ -Wall -g -std=c++11 main.cpp -o opengl_test -lglfw3 -lGLU -lGL -lGLEW -lX11 -lXxf86vm -lXrandr -lpthread -lXi -lXinerama -lXcursor`

If you miss any of the link options, or list them in the wrong order you'll get errors about undefined symbols. Note, I tested on arch linux and there the link order doesn't seem to matter... Go figure :)

HTH
1 • Reply • Share ›

John ➚ Rich Evans • a year ago
Yep, the above is totally true. Seems like GLFW has updated its list of dependencies, though, so the current bleeding edge libraries require that you link with:
-lglfw3 -lGL -lGLU -lpthread -lm -lX11 -lXcursor -lXxf86vm -lXrandr -lXinerama -lXi -lXmu -ldl

Links

OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

-lglfw3 -lGL -lGLU -pthread -lm -lXrr -lXcursor -lXxf86vm -lXrandr -lXinerama -lXi -lXmu -ldl

I uploaded an example makefile that uses these parameters to https://github.com/Uni-Sol/cg-...

Reply • Share ›

Krist Silvershade ➚ Rich Evans • 2 years ago

Thank you so much for posting this. This would have taken me forever to figure out what was wrong. Google-searches elsewhere kept pointing me to re-compiling glfw, which I knew wasn't the issue.

Reply • Share ›

hello_world • 2 years ago

This is such an amazing tutorial - succinct and well-written; it is exactly what I was looking for. Thank you!

1 Reply • Share ›

Matt • 3 years ago

I recommend to use this code instead for loading GLEW:

```
glewExperimental=TRUE;
GLenum err=glewInit();
if(err!=GLEW_OK)
{
//Problem: glewInit failed, something is seriously wrong.
std::cout << "glewInit failed, aborting. Code " << err << ". " << std::endl;
}
```

If there is a failure, it will give you an error code you can look up in glew.h.

Also, if you have lots of trouble with libraries and linking, you can try manually including the source glew.c and glew.h.

1 Reply • Share ›

vombatus • 4 years ago

Hooray new version of GLEW is out! (1.8.0; 17 July '12) Will you use it?

1 Reply • Share ›

Syaiful Maulana Abbidin • a month ago

Thanks, I have tried using SFML method, & it more easy.

Reply • Share ›

Zewen Huang • 2 months ago

Hi there
Sorry for asking a silliy question,but I am using Mac 10.11.6. I clone the repository from https://github.com/Polytonic/G.... However when I was trying to type
cmake -G "Xcode" ..
The following error occurs,Can anyone help me?

CMake Error at /usr/local/Cellar/cmake/3.6.2/share/cmake/Modules/CMakeDetermineSystem.cmake:177 (file):
file failed to open for writing (No such file or directory):

/Users/zewenhuang/Desktop/UW/CS488/Practise/Glitter/Build/CMakeFiles/CMakeOutput.log
Call Stack (most recent call first):
CMakeLists.txt:2 (project)

CMake Error: Could not open file for write in copy operation
/Users/zewenhuang/Desktop/UW/CS488/Practise/Glitter/Build/CMakeFiles/3.6.2/CMakeSystem.cmake.tmp
CMake Error: : System Error: No such file or directory
CMake Error at /usr/local/Cellar/cmake/3.6.2/share/cmake/Modules/CMakeDetermineSystem.cmake:189
(configure_file):
configure_file Problem configuring file

see more

Reply • Share ›

Alexander Overvoorde Mod ➚ Zewen Huang • 2 months ago

I recommend posting your problem as an issue in the Glitter repository.

Reply • Share ›

Zewen Huang ➚ Alexander Overvoorde • 2 months ago

Sorry I didnt know I can post to there

Reply • Share ›

Links

OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

GDW13 • 7 months ago

Thanks for the great site!

I'm using Glitter to get started quickly:
"Instead of reading this chapter, you can make use of the OpenGL quickstart boilerplate, which makes setting up an OpenGL project with all of the required libraries very easy. You'll just have to install SOIL separately."

I found this in the Glitter documentation:
"If you started the tutorials by installing SDL, GLEW, or SOIL, stop. The libraries bundled with Glitter supersede or are functional replacements for these libraries."

Do I have to install SOIL separately or does that note need to be removed?

Reply • Share ›

Load more comments

ALSO ON OPEN.GL

Drawing polygons
2 comments • 2 years ago•
Alejandro Segovia — VAOs are not optional if you're working with a Core OpenGL Context, otherwise you'll get an Invalid Operation error when you draw your …

Transform Feedback
17 comments • 3 years ago•
Wayasam — I would love to see a lightning tutorial. Most lightning tutorials are deprecated, using old OpenGL.

Context creation
1 comment • 2 years ago•
Martin — I'd like to point out, that using SDL, I needed to add GL/glx.h header to properly compile the file, otherwise the compilator couldn't find …

Geometry shaders
12 comments • 3 years ago•
Simon Jackson — I found this link (http://www.gamedev.net/page/re...) to be a much better tutorial for instancing