Date: 4/12/2015

Sparky Game Engine – Notes

Author: @nyguerrillagirl

> "Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning."
> – Rich Cook

V1: Sparky Engine

These notes capture the development of the Sparky Game Engine. The game engine will be developed as part of a new video series hosted at the following urls:

- https://www.youtube.com/channel/UCQ-W1KE9EYfdxhL6S4twUNw - you tube videos

Figure 1 - TheCherno youtube home page

- http://www.twitch.tv/thecherno - live videos of the game being developed
- https://github.com/TheCherno - GitHub location of source code for this project

The game engine will be built in 45-days in preparation for the upcoming Ludem Dare competition. This will allow game developers to concentrate on the game mechanics and not the common details that the typical game engine handles.

What is Ludem Dare?

Developers from around the world get together several times a year and spend a weekend building a game from scratch using the game theme presented to all the developers. The game theme is proposed and selected from the game development community. The goal is for the individual or a team to create a game in 72 hours. You can use any tools or libraries or start with any base-code you have. You can even use art work and music from 3rd party sources. You are encouraged to create games that can be played right in the browser. The source code created is made freely available to community. This practice encourages sharing and growth as others can learn new techniques and programming styles by reviewing code (of the winners and losers).

I will post the latest notes on github at https://github.com/nyguerrillagirl/Sparky-Engine. The repository will contain:

- The latest copy of these notes
- A file named codeCommitSessions listing the commit codes corresponding to the code developed for the end of a video coding session. For example if you want to obtain the code as it was developed up to video session #5 then the commit number (e.g. d42371e6a7) will represent

the code completed up to that session. So checking out that commit[1] will get you to that location in these notes and on the video.

- The code related to this project

I highly recommend that you follow TheCherno on the various locations listed above for more information and to view the videos relates to these notes and to donate to TheCherno.

You can follow me at @nyguerrillagirl on twitter or visit my web site www.brainycode.com . Do not hesitate to provide corrections and suggestions to my e-mail address at nyguerrillagirl at brainycode dot com.

What is a Game Engine?

A game engine is a code package, framework or application tool set that implement the common tasks that all games perform – rendering the screen, physics, input, and collision detection, etc. so that the game developer can concentrate on game specific details like art work, and specific game logic (e.g. player has laser eye weapon).

A typical game engine provides an API or SDK (collection of libraries)[2] that the game developer will use as a starting point to build their own game.

You are probably familiar with popular and well-known game engines such as Unity (http://unity3d.com/) or the Unreal Engine (https://www.unrealengine.com/). These engines are rather complex and have a high learning curve. There are simpler game engines such as GameMaker:Studio (http://www.yoyogames.com/studio) and Construt 2 (https://www.scirra.com/construct2) that make it quite easy to build 2D games.

Using a game engine (most are free) will provide an insight into what you want your own game engine to do, so I advise that you check out one or more game engines in order to get an idea of the typical functions that are abstracted away into APIs and how a game developer would interface with a game engine in order to build their actual game.

Why build your own game engine?

The benefits of developing your own game engine is the appreciation you gain on what a decent game engine is doing and the technical complexity in implementing one. In addition, it will make is easier to learn and work with more complex game engines.

What Programming Language to use?

The first question being debated is what programming language to use. The two top choices in order to achieve maximum portability is Java or C++.

[1] See Appendix A for a quick tutorial on using GitHub.
[2] http://www.gamecareerguide.com/features/529/what_is_a_game_.php

C or C++ has been the workhorse or primary language of choice for most in-house game engines and/or games. The advantages of using the programming language C++ are:

- Compiles into native machine code so the game engine will be fast
- Supported on all platforms
- Free IDEs and compilers are available (that run on all platforms)
- C++ is well-known by most developers and students
- C++ has many supporting libraries and frameworks

The downside of using C++ is that we probably will not get as many features as we would like completed for our game engine in the designated timeframe. The reason most game engine developers select C++ is because C++ has been the dominant language choice for many years (yes - use it because everyone else has for years!). The leading game consoles development tools use C++, many middleware packages are written in C++ and when you think about it if you want support, assistance or want to expand you work of art it will be easier to find C++ experts.

The major downside of using C++ is the plain ugliness[3] of the code since it has object-oriented concepts layered on top of C so there is much reuse of same symbols which makes it rather difficult to read.

Java is up for consideration because

- it is easier to build a cross-platform game engine with it
- software development and debugging is faster
- has freely available tools and frameworks

Java comes with a host of libraries and tools (all free!) that make it easier to build and test programs. It will be cross-platform with no extra work or flags required in our code. The downside is the fact that if you are trying to prepare to get a job with a company that develops professional console games chances are that they DO NOT USE Java[4].

What platforms should it work on?

The main platform that the game engine will be designed for is Windows. A secondary goal will be to have it run on Mac and Linux environments.

[3] This is rather subjective since beauty is the eye of the beholder (and therefore so is ugliness). I think it is fair to state that the more experience one has with C++ the less ugly it seems.
[4] This may not be true if they are building games for Android devices

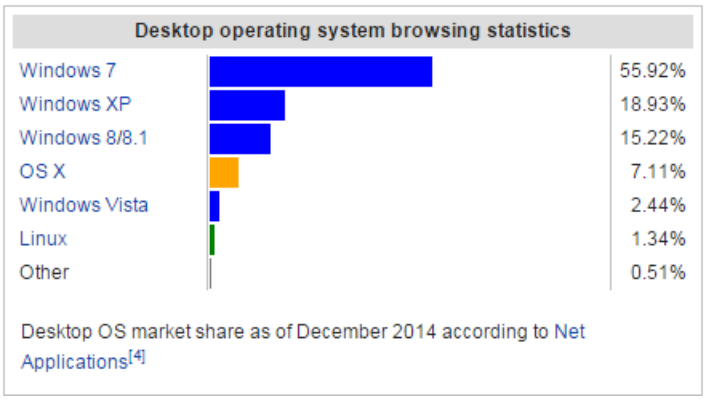


Figure 2 - Operating systems on desktops

As you can see Windows 7 has the largest share of users on desktop environments. The game engine will be built on a Windows OS. I myself will be using Windows 7.

As the project progresses I plan on using the same compiler and IDE that is available on Mac and Linux platforms in order to ensure everything works on as many platforms as possible. I plan on setting up my Mac and Linux machines with the same version of all the tools and provide any notes on differences in one of the appendices.

TBD: Add reference to actual appendix.

What are the goals?

The goal is to build a 3D game engine. We will get to our goal by first focusing on building a 2D game engine and expanding it to support 3D.

Figure 3 - 2D game look-and-feel

The classic and most popular type of 2D game is the 2D platform scroller (as shown above). The game engine will support the basic elements of building a 2D game:

- Creating and laying out game objects (players and enemies)

- Collision detection
- Input handing
- Scrolling
- Level transition
- Sound
- Heads up display

Many game engines (e.g. Unity) allow you to build either a 2D or 3D game.

Figure 4 - 3D Game, Off-Road Velociraptor Safari

It probably will be stretch but the ultimate goal is to have our game engine support the building of a 2D or 3D game.

What technologies?

The graphics for our game engine will be rendered using OpenGL. If you are new to OpenGL please check out Appendix C for an introductory overview. The book will be written under the assumption that Windows programming and OpenGL are new topics.

OpenGL stands for Open Graphics Library. It is an API (set of libraries) for defining 2D and 3D graphics images.

What languages can developers use to build their games?

We would like not to limit game designers or developers to using the same programming language we selected for the game engine. If we select C++ as the programming language for the game engine it will still be possible for game developers to use Java, C#, C++ or other languages.

In addition, it is quite possible to make our game engine extendable by providing support for LUA or JavaScript as an internal scripting language.

Open Source – Available on GitHub

All material related to this project (even these notes) will be open source and available on GitHub. In addition, we will stick to tools and libraries that are open source and freely available.

What about images and art work?

In order to test and exercise the portions of the game engine we will need to build and "see" things working. This will require game assets such as sprites and texture models. I will stick to freely available assets but may on occasion demonstrate a feature using art assets that we have purchased. I find the website https://www.gamedevmarket.net/ a good place to obtain decent and well-priced art assets.

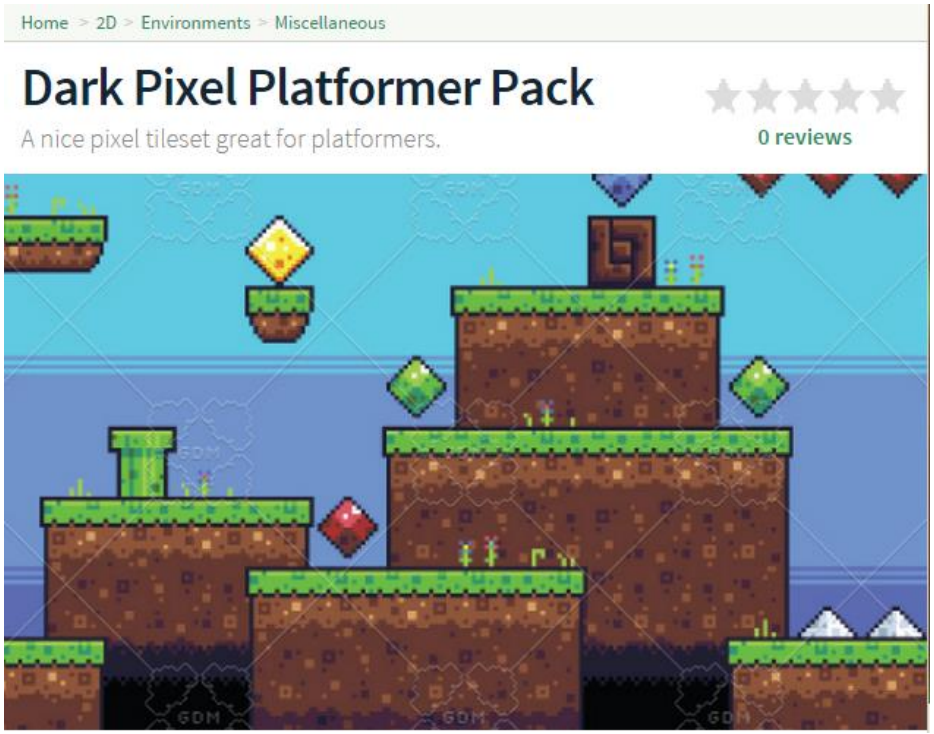


Figure 5 - My purchased art assets

I cannot make any purchased art assets available but I do encourage you to consider making a purchase and supporting the game art development community.

Prerequisites

This set of videos and notes assume you know how to program in C++. You should know the selected language well enough to feel comfortable with the creation of simple programs and how to build C++ objects.

V2: Setting up the Development Environment

The development environment used on the videos is Microsoft Visual Studio Community 2013. I will not follow the videos along using the same IDE but will document and detail in these notes using an IDE that is available on the Mac and Linux platforms – Code::Blocks. See Appendix B for details on how to set up on other environments.

Visual Studio Community 2013

1. Search the web for the download page
2. Download the initial installer executable (vs_community.exe)
3. Follow the instructions[5]

Cross-Platform IDE – Code::Blocks

We will use Code::Blocks to develop the Sparky Game Engine.

1. Go to http://www.codeblocks.org/
2. Click the Downloads link and download and start the installation of the latest version of Code::Blocks

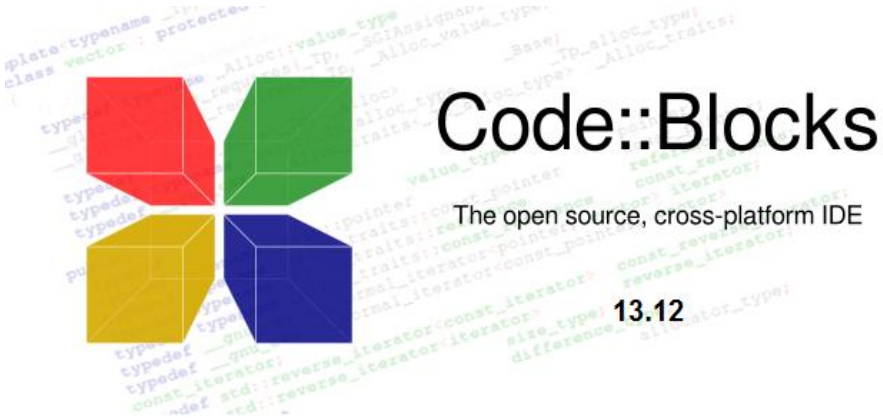
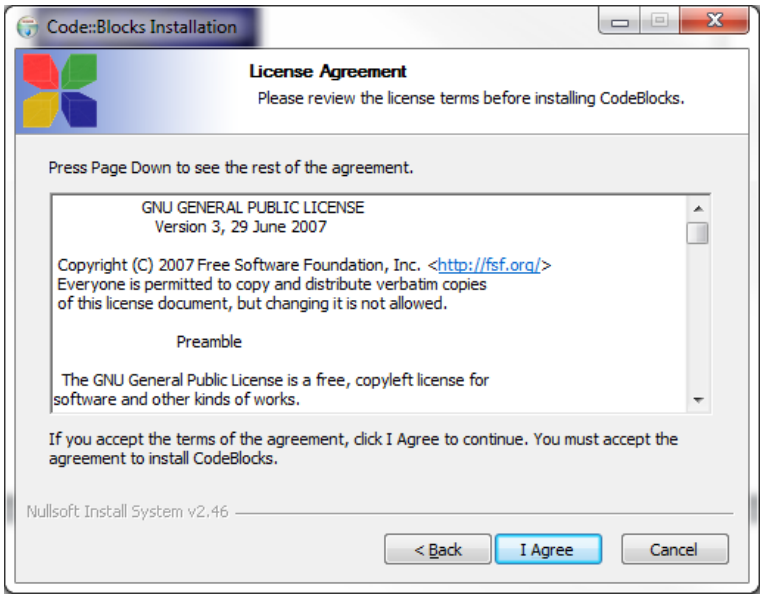


Figure 6 - Starting Code::Blocks Start Screen

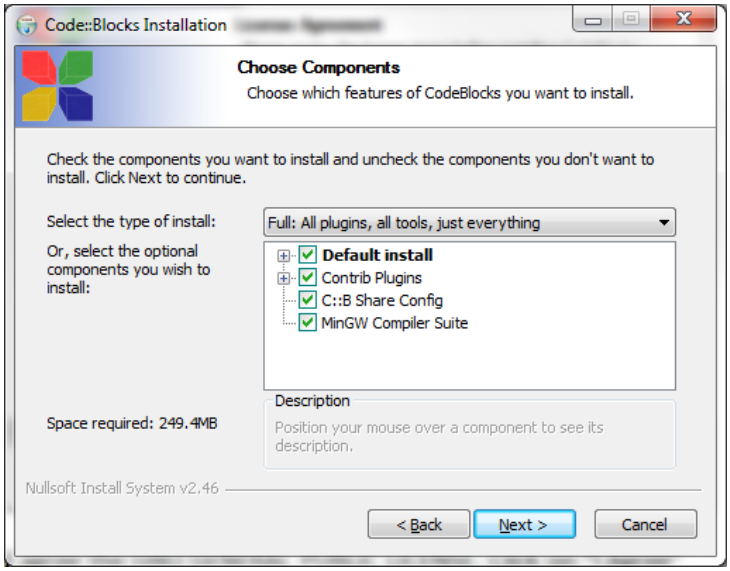
The Code::Blocks Installation Setup Wizard screen appears.

[5] I opted not to prepare these notes using Microsoft Visual Studio Community but opted instead to use an IDE available on Mac and Linux platforms.

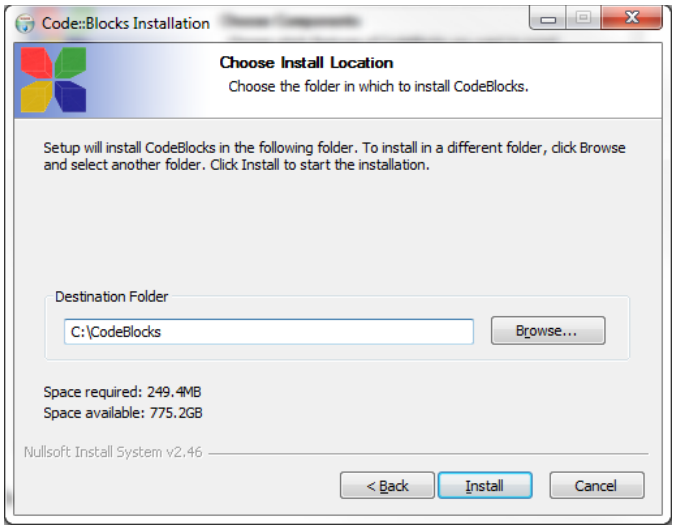
- Click Next >



- Read and agree the GNU GENERAL PUBLIC LICENSE. Click on “I Agree”



- Accept the defaults and click on “Next > “



- I elected to install under the C Drive C:\CodeBlocks rather than C:\Program Files. You can take the default and click on “Install”
- Allow the application to open after installation completes.

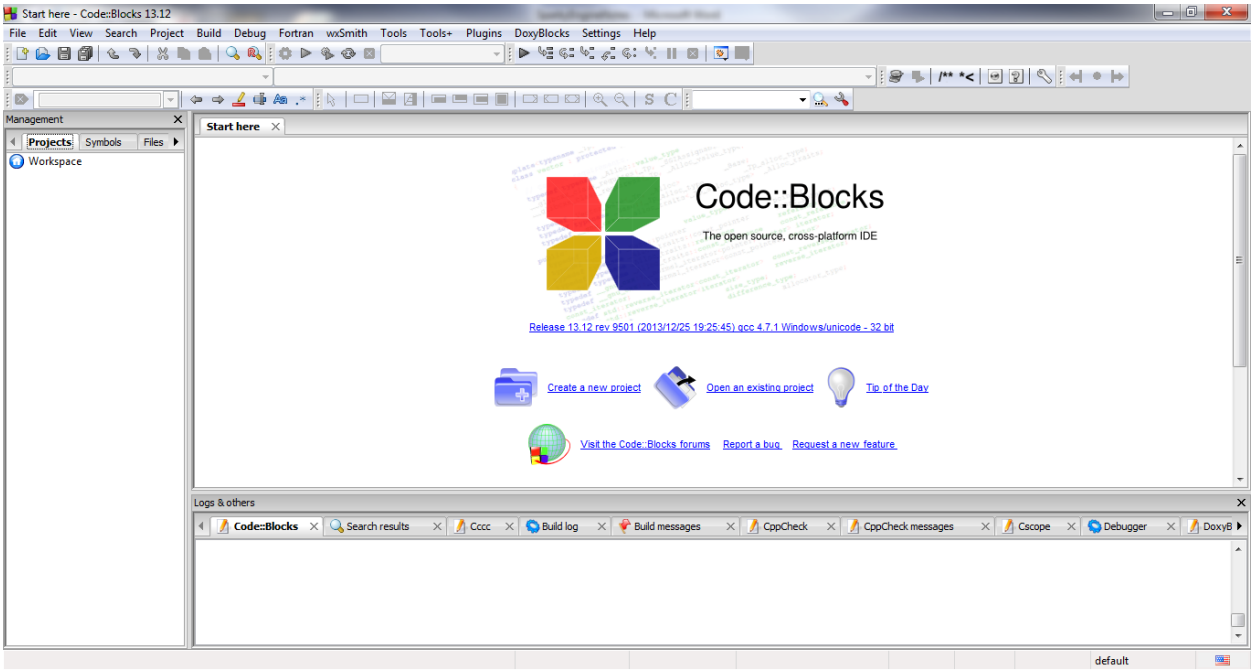


Figure 7 - Code::Blocks Start Screen

How to use Code::Blocks

Your typical application in Code::Blocks consists of a workspace made up of one or more projects. We will illustrate some of Code::Blocks features by building the following type of programs:

- A Hello, World Program
- A Simple Windows Application
- A Simple Windows OpenGL Application

The sample programs we will build in this section highlight the increasing amount of boilerplate[6] code required just to display a window. Episode 1 will discuss popular libraries that hide all the boilerplate code and provide cleaner APIs to make it easy to create graphics based Windows applications. You can opt out of dipping into the details in this section since its only intention is to generate an appreciation for what libraries such as OpenGL, SDL and GLFW provide us.

Create a directory to save to save your Code::Blocks projects. I will place all my Code::Blocks projects under C:\CodeBlocks\Workspaces.

Simple Console Application

1. Select File → Project from the top menu

[6] This is the typical or standard lines required for the platform under consideration

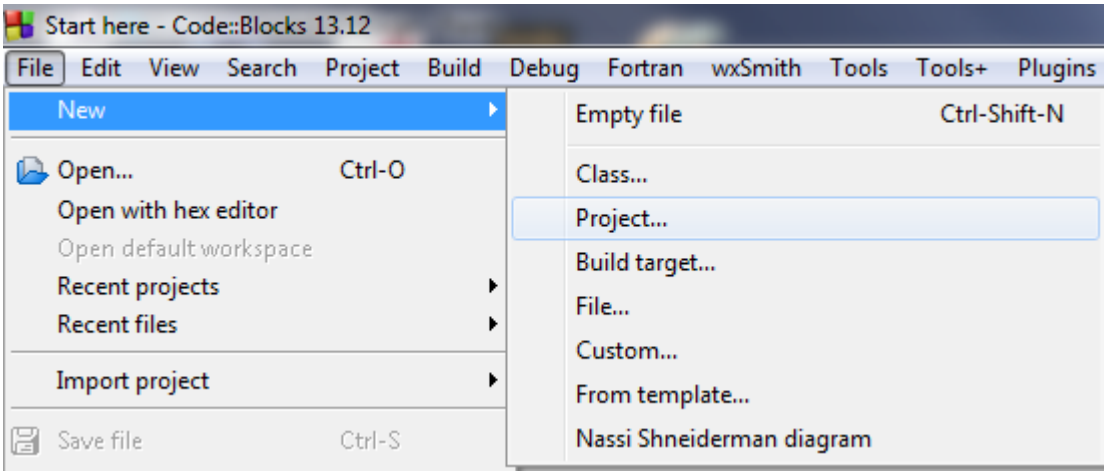


Figure 8 - Creating a new project

A "New from template" dialog box opens.

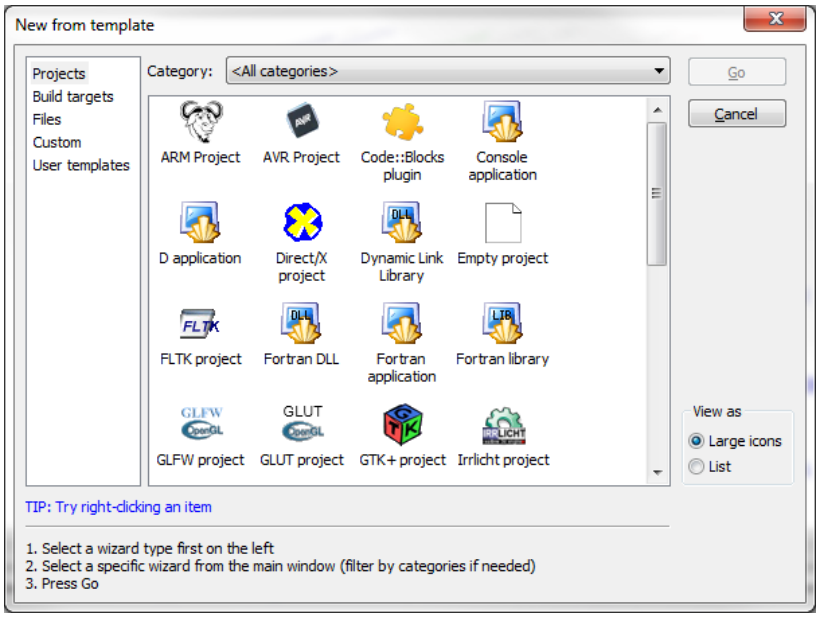


Figure 9 - New Project template dialog box

The list of project templates details are shown by the various icons displayed. You may notice that Code::Blocks has templates for some popular graphics packages – OpenGL, GLFW and GLUT! Selecting some of the templates (e.g. GLFW project and GLUT project) require that you download and install these library frameworks so we will not use these templates at this time.

2. Select/Highlight "Console application" and click on "Go"

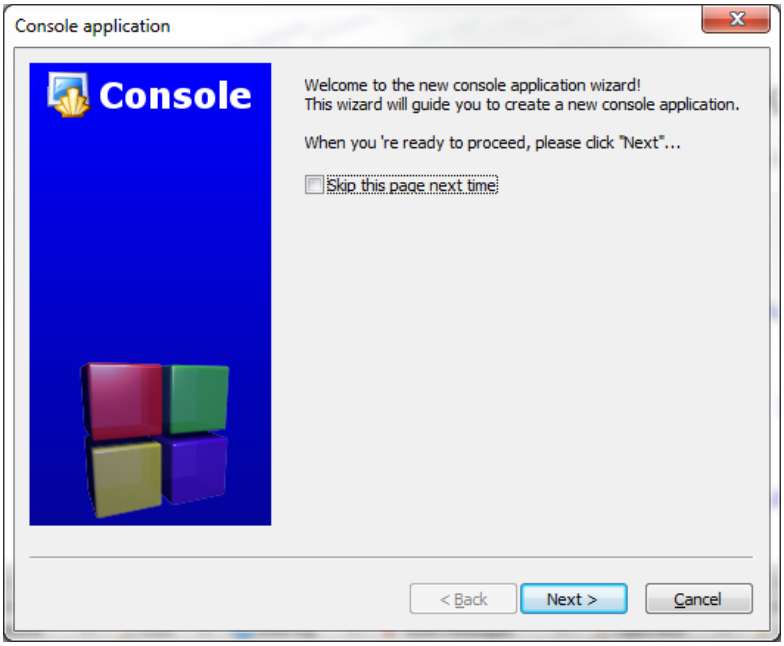


Figure 10 - Console application wizard

What is a Console Application[7]?

A console application is an application that takes input and displays output at a command line console with access to three basic data streams: standard input, standard output and standard error.

A console application facilitates the reading and writing of characters from a console – either individually of as an entire line. It is the simplest form of a C++ program and is typically invoked from the Windows command prompt. A console application usually exists in the form of a stand-alone executable file with minimal or no graphical user interface (GUI).

If you have been around computers for a while you will be reminded of the good ol' days of DOS when working with a console application. You may be surprised to learn that you can create quite impressive screens within the limitations of a console application given the range of characters and colors you can utilize when displaying screen contents.

[7] http://www.techopedia.com/definition/25593/console-application-c

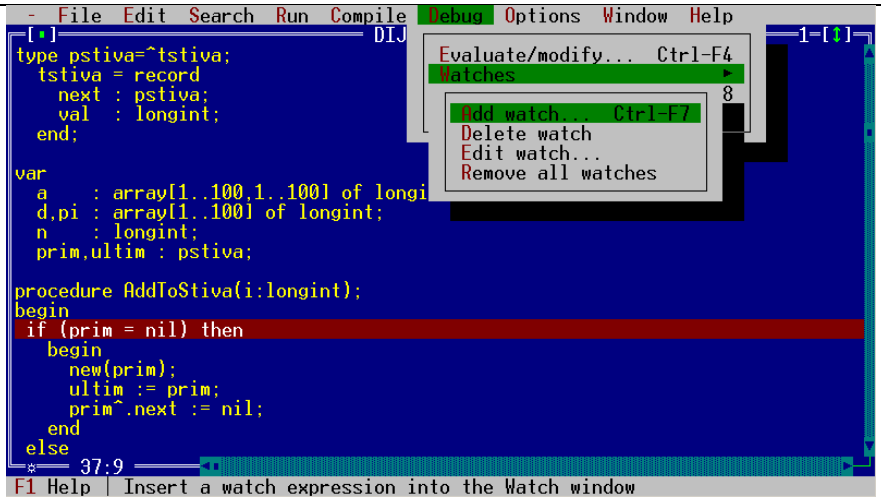


Figure 11 - Turbo Pascal using console characters and colors

I have seen Pong, Tetris and yes even a small Pac-man created within the limitations of a console application.

When we build a windows console program we are hiding all the details of the creation and management of the window to the operating system. We get to write `cout` statements to a simple character based display.

3. Click on “Next >”

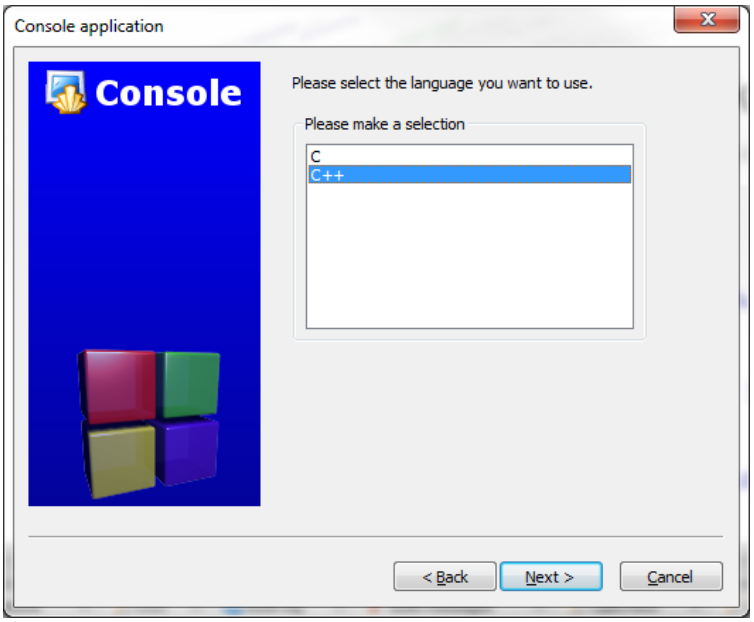


Figure 12 - Console application - Language selection

4. Select/Highlight C++ and click on “Next >”

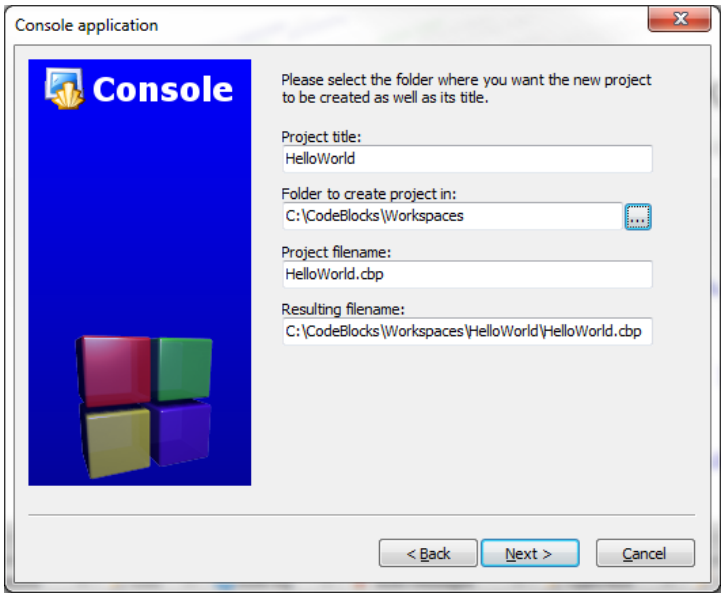


Figure 13 - Console application - creating project

Fill in the “Project title” and navigate to the location you want to save your project.

Note: The project is saved as a *.cbp file under a folder or directory having the same name as the “Project title”

5. Click “Next >”

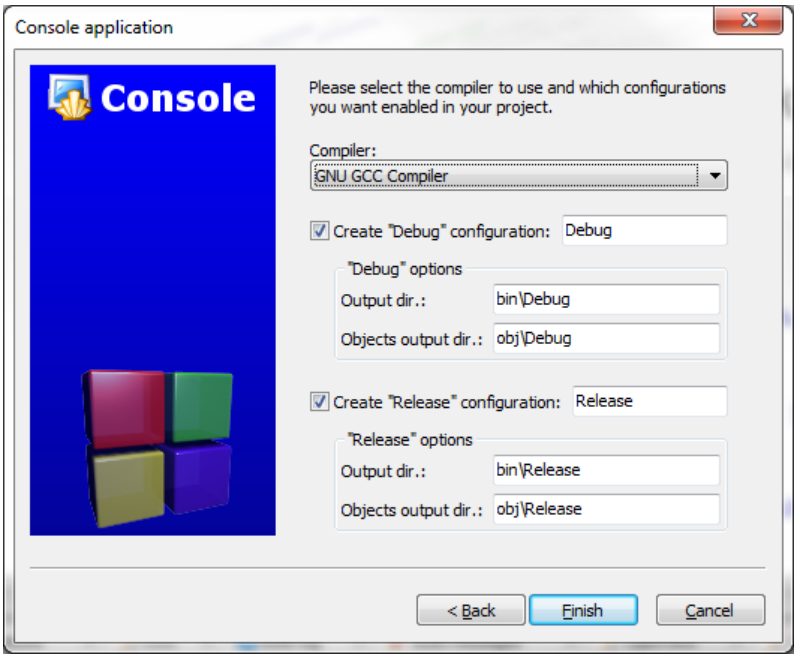


Figure 14 - Console application - configuration settings

Take the default “Debug” and “Release” configuration settings locations.

6. Click “Finish”

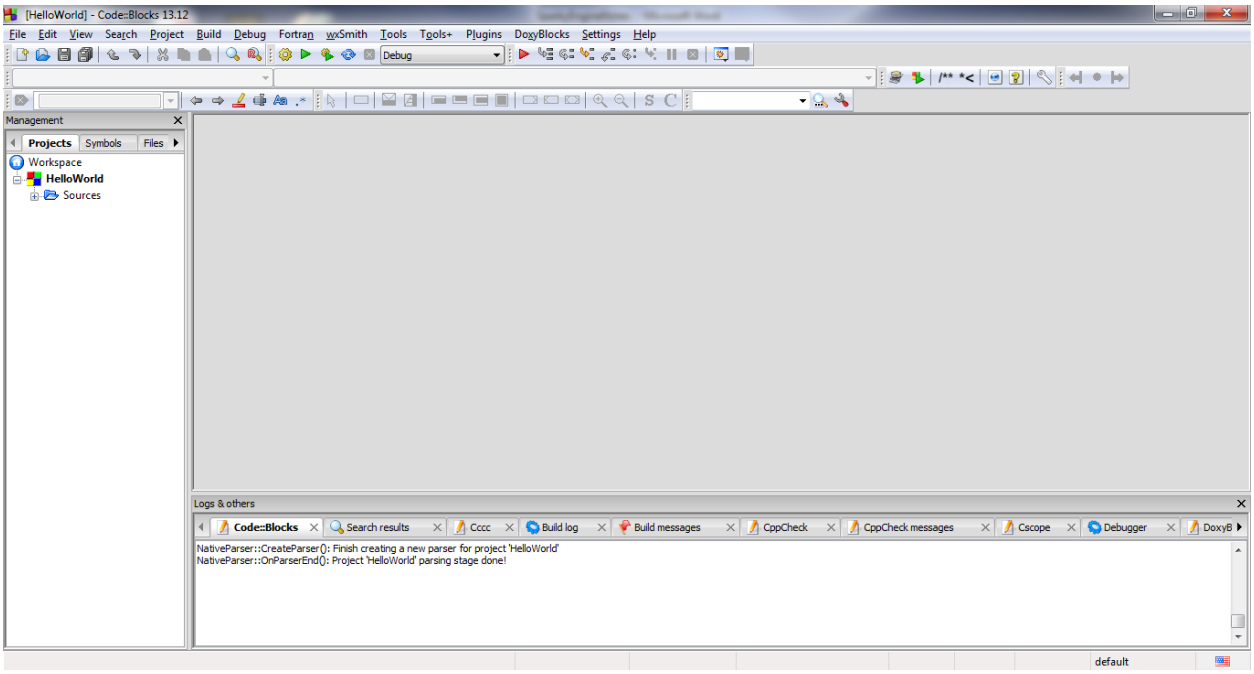


Figure 15 - Hello World Project Start

7. Click on the “Sources” folder in the Project tab in the Management View
8. Double click on the “main.cpp” file to open it up in the Editor.

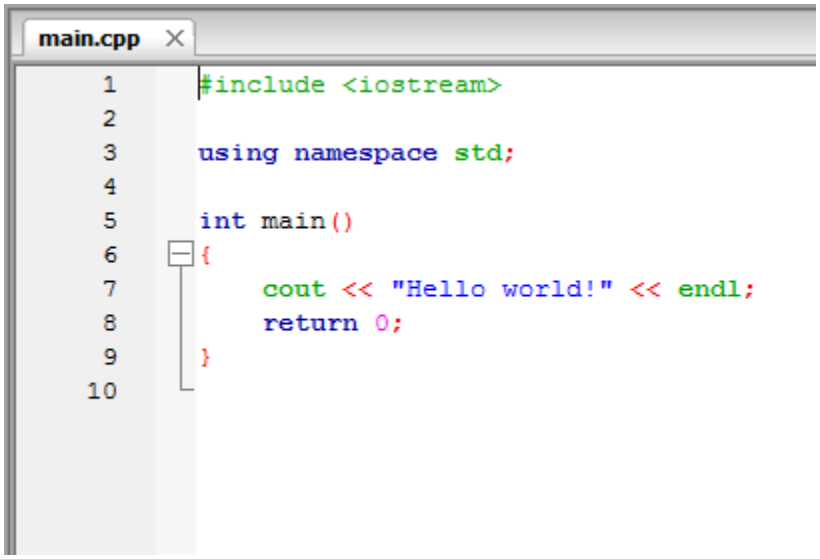


Figure 16 - Our “Hello world!” main.cpp

As you can see you the “console” template comes with one file named main.cpp.

Figure 17 - Build, Run, Build and Run

The icons shown above can be used to “Build”, “Run” or “Build and Run” the project, respectively.

9. Click on “Build” icon.

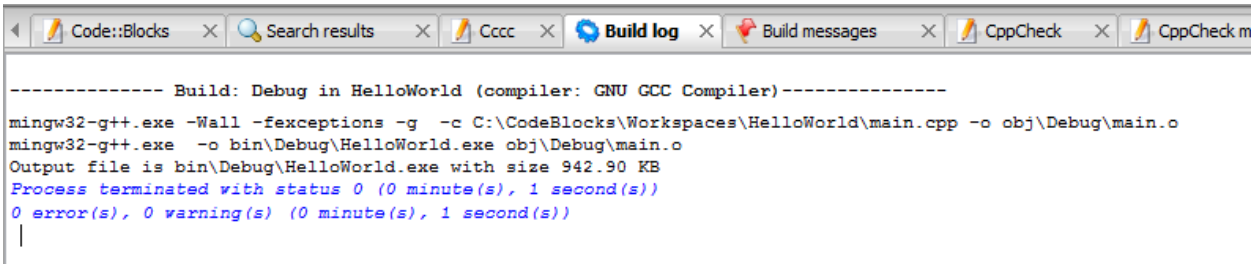


Figure 18 - Build log

The bottom of the IDE has several tab views. The “Build log” view displays information on the files complied (e.g. main.cpp), the object files created (e.g. main.o) and the linking to create the HelloWorld.exe file.

Click on the “Code::Blocks” and “Build messages” tab to view additional messages.

10. Click on the Run icon.

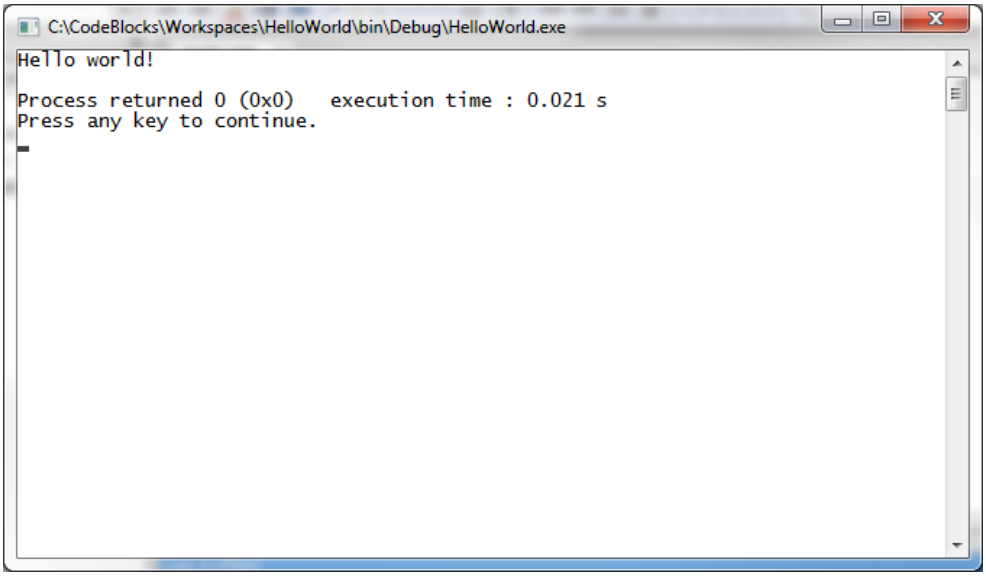


Figure 19 - Resulting console window

The resulting console window opens and displays the “Hello world!” message.

11. Close this project by selecting File → Close all projects.

Simple Win32 Windows Application

1. File → New Project
2. Scroll down to “Win32 GUI project”
3. Click “Next >” on the Win32 GUI project screen
4. Select “Frame based” and click on “Next >”

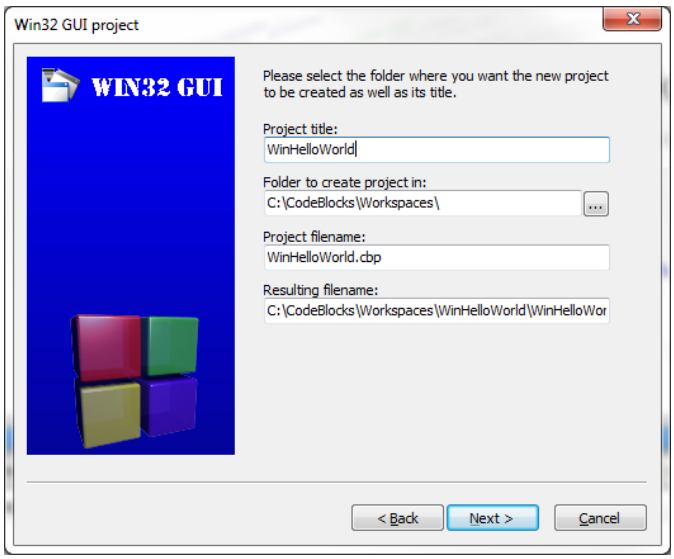


Figure 20 - Creating Win32 GUI project

5. Enter the “Project title” and click on “Next >”
6. Accept the default configuration information and click on “Finish”

Table 1 - main.cpp (for a Windows application)

```
#if defined(UNICODE) && !defined(_UNICODE)
    #define _UNICODE
#elif defined(_UNICODE) && !defined(UNICODE)
    #define UNICODE
#endif

#include <tchar.h>
#include <windows.h>

/*  Declare Windows procedure  */
LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM, LPARAM);

/*  Make the class name into a global variable  */
TCHAR szClassName[ ] = _T("CodeBlocksWindowsApp");

int WINAPI WinMain (HINSTANCE hThisInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszArgument,
                    int nCmdShow)
{
```

```
    HWND hwnd;               /* This is the handle for our window */
    MSG messages;            /* Here messages to the application are saved */
    WNDCLASSEX wincl;        /* Data structure for the windowclass */

    /* The Window structure */
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc = WindowProcedure;      /* This function is called by windows */
    wincl.style = CS_DBLCLKS;                 /* Catch double-clicks */
    wincl.cbSize = sizeof (WNDCLASSEX);

    /* Use default icon and mouse-pointer */
    wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
    wincl.lpszMenuName = NULL;                 /* No menu */
    wincl.cbClsExtra = 0;                      /* No extra bytes after the window class
*/
    wincl.cbWndExtra = 0;                      /* structure or the window instance */
    /* Use Windows's default colour as the background of the window */
    wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;

    /* Register the window class, and if it fails quit the program */
    if (!RegisterClassEx (&wincl))
        return 0;

    /* The class is registered, let's create the program*/
    hwnd = CreateWindowEx (
           0,                   /* Extended possibilites for variation */
           szClassName,         /* Classname */
           _T("Windows Hello World"),       /* Title Text */
           WS_OVERLAPPEDWINDOW, /* default window */
           CW_USEDEFAULT,       /* Windows decides the position */
           CW_USEDEFAULT,       /* where the window ends up on the screen */
           544,                 /* The programs width */
           375,                 /* and height in pixels */
           HWND_DESKTOP,        /* The window is a child-window to desktop */
           NULL,                /* No menu */
           hThisInstance,       /* Program Instance handler */
           NULL                 /* No Window Creation data */
           );

    /* Make the window visible on the screen */
    ShowWindow (hwnd, nCmdShow);

    /* Run the message loop. It will run until GetMessage() returns 0 */
    while (GetMessage (&messages, NULL, 0, 0))
    {
        /* Translate virtual-key messages into character messages */
        TranslateMessage(&messages);
        /* Send message to WindowProcedure */
        DispatchMessage(&messages);
    }

    /* The program return-value is 0 - The value that PostQuitMessage() gave */
    return messages.wParam;
}
```

```
/*  This function is called by the Windows function DispatchMessage()  */

LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    TCHAR strHello[ ] = _T("Hello, World");

    switch (message)                  /* handle the messages */
    {
        case WM_PAINT:
            hDC = BeginPaint(hwnd, &ps);
            SetBkMode(hDC, TRANSPARENT);
            TextOut(hDC, 50, 50, strHello, lstrlen(strHello));
            EndPaint(hwnd, &ps);
            break;
        case WM_DESTROY:
            PostQuitMessage (0);       /* send a WM_QUIT to the message queue */
            break;
        default:                      /* for messages that we don't deal with */
            return DefWindowProc (hwnd, message, wParam, lParam);
    }

    return 0;
}
```

The above represents the “simplest” Win32 program you can write.

7. Build and run.

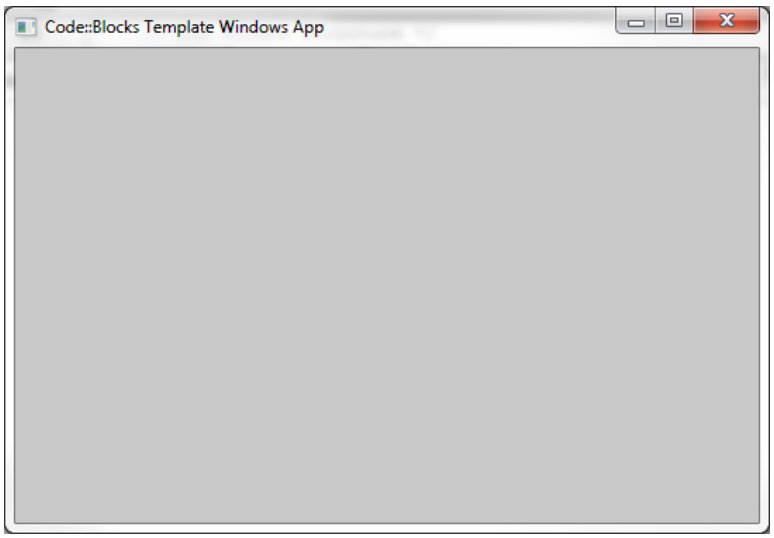


Figure 21 - Windows Program

TODO: Explain the details in this windows program

The scary thought is that this is the simplest Windows program you can create where you define and create your own window. What is so scary? It doesn't do anything yet. The program starts with preprocessor statements:

Table 2 - UNICODE directives

```
#if defined(UNICODE) && !defined(_UNICODE)
    #define _UNICODE
#elif defined(_UNICODE) && !defined(UNICODE)
    #define UNICODE
#endif
```

Let's make it say "Hello World" in the title and in the middle of the window.

Exercise 1: Change the Window title to "Windows Hello World"

TODO: Explain enough to write text "Hello World" on the screen.

Exercise 2: Write "Hello, World"

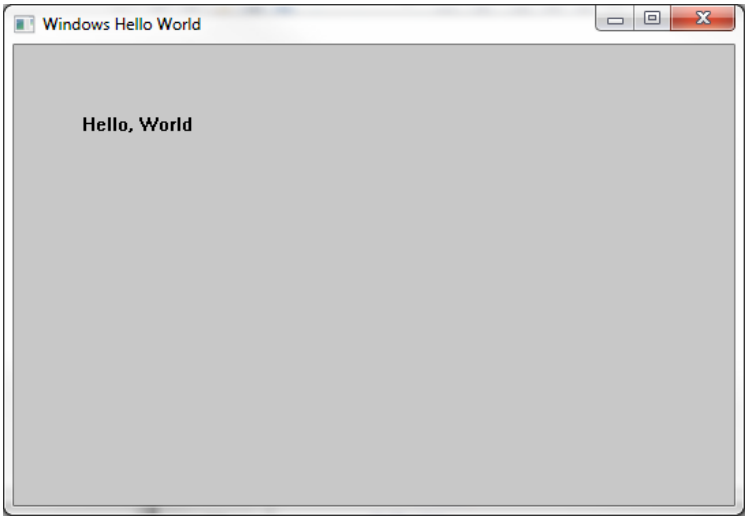


Figure 22 - Win32 (Hello, World) text

8. Close all Projects

Simple OpenGL Application

1. File → New Project
2. Select OpenGL Project template
3. Click on "Next >"
4. Enter the name "WinOpenGL"

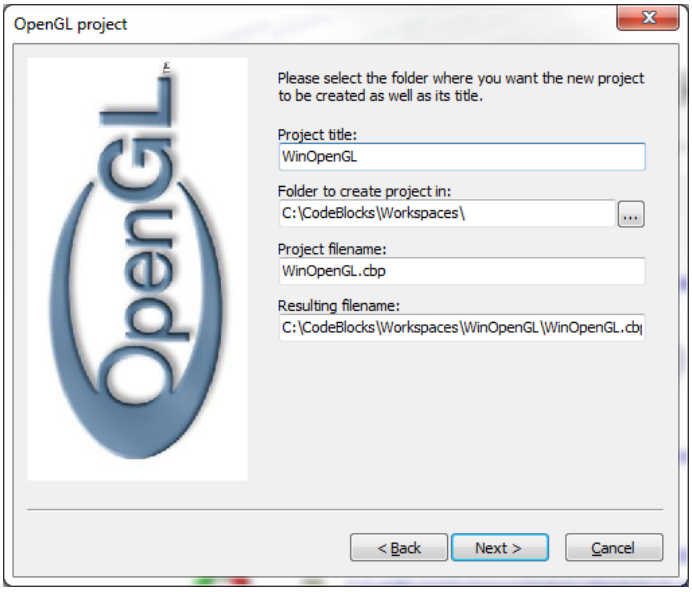


Figure 23 - Creating OpenGL Win32 Project

5. Click on “Next >”
6. Accept default configuration and click on “Finish”
7. Make the change noted in the code

Table 3 - main.cpp (OpenGL application)

```
#include <windows.h>
#include <gl/gl.h>

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
void EnableOpenGL(HWND hwnd, HDC*, HGLRC*);
void DisableOpenGL(HWND, HDC, HGLRC);


int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
{
    WNDCLASSEX wcex;
    HWND hwnd;
    HDC hDC;
    HGLRC hRC;
    MSG msg;
    BOOL bQuit = FALSE;
    //float theta = 0.0f;

    /* register window class */
    wcex.cbSize = sizeof(WNDCLASSEX);
    wcex.style = CS_OWNDC;
    wcex.lpfnWndProc = WindowProc;
    wcex.cbClsExtra = 0;
```

```
    wcex.cbWndExtra = 0;
    wcex.hInstance = hInstance;
    wcex.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
    wcex.lpszMenuName = NULL;
    wcex.lpszClassName = "GLSample";
    wcex.hIconSm = LoadIcon(NULL, IDI_APPLICATION);;


    if (!RegisterClassEx(&wcex))
        return 0;

    /* create main window */
    hwnd = CreateWindowEx(0,
                          "GLSample",
                          "OpenGL Sample",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          960,
                          540,
                          NULL,
                          NULL,
                          hInstance,
                          NULL);

    ShowWindow(hwnd, nCmdShow);

    /* enable OpenGL for the window */
    EnableOpenGL(hwnd, &hDC, &hRC);

    /* program main loop */
    while (!bQuit)
    {
        /* check for messages */
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            /* handle or dispatch messages */
            if (msg.message == WM_QUIT)
            {
                bQuit = TRUE;
            }
            else
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
        }
        else
        {
            /* OpenGL animation code goes here */

            glClearColor(0.2f, 0.3f, 0.8f, 1.0f);
            glClear(GL_COLOR_BUFFER_BIT);

            //glPushMatrix();
            //glRotatef(theta, 0.0f, 0.0f, 1.0f);
```

```
                    // We will draw a white rectangle
            glBegin(GL_QUADS);
            glVertex2f(-0.5f, -0.5f);
            glVertex2f(-0.5f,  0.5f);
            glVertex2f( 0.5f,  0.5f);
            glVertex2f( 0.5f, -0.5f);

            //  glColor3f(1.0f, 0.0f, 0.0f);   glVertex2f(0.0f,   1.0f);
            //  glColor3f(0.0f, 1.0f, 0.0f);   glVertex2f(0.87f,  -0.5f);
            //  glColor3f(0.0f, 0.0f, 1.0f);   glVertex2f(-0.87f, -0.5f);

            glEnd();

            //glPopMatrix();

            SwapBuffers(hDC);

            //theta += 1.0f;
            Sleep (1);
        }
    }

    /* shutdown OpenGL */
    DisableOpenGL(hwnd, hDC, hRC);

    /* destroy the window explicitly */
    DestroyWindow(hwnd);

    return msg.wParam;
}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_CLOSE:
            PostQuitMessage(0);
        break;

        case WM_DESTROY:
            return 0;

        case WM_KEYDOWN:
        {
            switch (wParam)
            {
                case VK_ESCAPE:
                    PostQuitMessage(0);
                break;
            }
        }
        break;

        default:
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }

    return 0;
```

```
}

void EnableOpenGL(HWND hwnd, HDC* hDC, HGLRC* hRC)
{
    PIXELFORMATDESCRIPTOR pfd;

    int iFormat;

    /* get the device context (DC) */
    *hDC = GetDC(hwnd);

    /* set the pixel format for the DC */
    ZeroMemory(&pfd, sizeof(pfd));

    pfd.nSize = sizeof(pfd);
    pfd.nVersion = 1;
    pfd.dwFlags = PFD_DRAW_TO_WINDOW |
                  PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
    pfd.iPixelType = PFD_TYPE_RGBA;
    pfd.cColorBits = 24;
    pfd.cDepthBits = 16;
    pfd.iLayerType = PFD_MAIN_PLANE;

    iFormat = ChoosePixelFormat(*hDC, &pfd);

    SetPixelFormat(*hDC, iFormat, &pfd);

    /* create and enable the render context (RC) */
    *hRC = wglCreateContext(*hDC);

    wglMakeCurrent(*hDC, *hRC);
}

void DisableOpenGL (HWND hwnd, HDC hDC, HGLRC hRC)
{
    wglMakeCurrent(NULL, NULL);
    wglDeleteContext(hRC);
    ReleaseDC(hwnd, hDC);
}
```

We commented out code and added new lines so rather than seeing the spinning triangle shown in Figure 24 we will display a white rectangle shown in XXX>

TODO: Explain OpenGL specific code.

8. Build and Run.

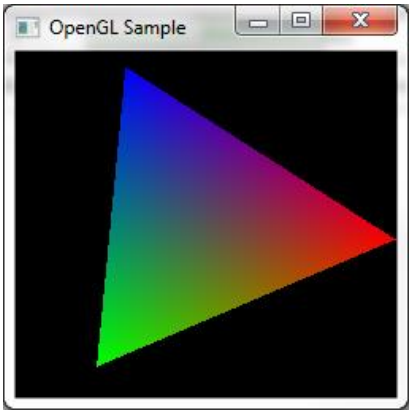


Figure 24 – Original OpenGL example

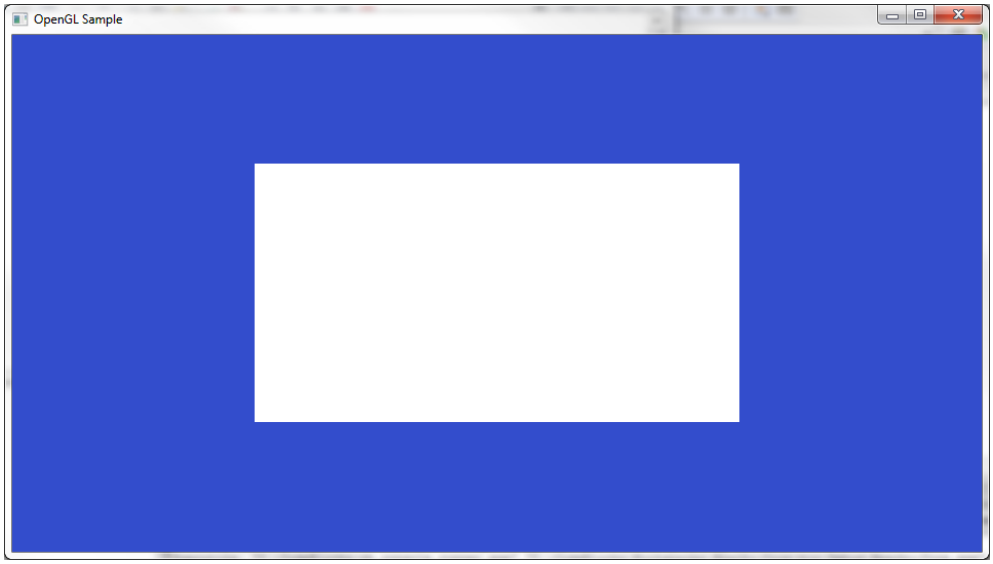


Figure 25- Modified OpenGL Example

Episode 1 – The Window

This is the first episode that discusses the design and construction of the Sparky Game Engine. We will discuss the following topics:

- What High-level Windows Graphics Library to use?
  - SDL
  - GLFW
- Start the Sparky Game Engine

The General Plan

It is a given that we will use OpenGL but as the last section illustrated we don't want to get into all that Windows boilerplate code that is required to get a window object defined and created. In addition, there is a lot of overhead in just getting text or graphics to show up on the display. Finally, if we clutter our code with all these Microsoft Windows specific details we can throw in the white flag right now about creating anything that will be remotely portable.

The fact is that the underlying details about how to create windows, dialogs, text, or draw can be hidden behind APIs that handle the platform specific details for us. That is why graphics libraries such as the Simple DirectMedia Layer (SDL) or GLFW library were created. These toolkits, SDKs, or libraries hide all the boilerplate code for us and allow us to concentrate on the core of the application we are building.

SDL

SDL has been around for quite some time and it has some of the ideal attributes for a windows-based library, it provides low-level access to:

- Audio
- Keyboard
- Mouse
- Joystick
- Graphics Hardware

SDL is cross-platform supported on Windows, Mac OS X, Linux, iOS and Android. It is free to use in any software application. The code is written in C with bindings for C++, C#, Python and more.

There are many games made today (many available on Steam) that use SDL so it would not be a bad choice for using in Sparky.

Some people consider SDL a bit too low-level but it has a low-learning curve and is usually combined with other libraries, e.g. OpenGL.

From: http://www.codedread.com/blog/archives/2005/08/23/sdl-in-games-introduction/

Actually SDL's approach to 2D graphics is fairly minimal. It allows you to configure your video settings (resolution, colour depth), load images (Windows BMP only), obtain pointers to surfaces that represent image data (both onscreen and in-memory), and perform blits to the screen. The blits can be standard

copies, colour-keyed copies (to provide simple transparencies or sprites) or more complex alpha-blended blits (to provide opacity and blending effects). There are a couple other lesser-used functions in SDL, but from a graphics perspective that's about it.

In some respects, since SDL needs to be cross-platform, it (unfortunately) needs to provide a "least-common denominator" approach for basic graphics support. For instance, SDL does not even provide a function to get or set an individual pixel colour, you actually have to write your own (or use the code provided in the SDL documentation here). However, it makes up for this sparseness by providing hardware accelerated support on platforms that can do this (for example, SDL uses DirectDraw on Windows platforms for fast hardware support on video cards that support this). SDL also has some good extension libraries you can add onto that provide you with richer functionality (more on these in later entries).

In addition, SDL provides the benefit of allowing you to handle events (keyboard, mouse, timer, joysticks) and audio in a cross-platform manner, which is a huge benefit if you're thinking of writing games once that can run everywhere. You can pick and choose the components that you need from SDL and there's not a lot of complicated "initialization" steps that need to happen before you can do something useful.

SDL does not provide support for 3D graphics, but it does integrate nicely with OpenGL. For example, you could use SDL to handle all your window initialization, event-handling, audio and then use OpenGL for your 3D graphics.

SDL is pretty easy to pick up since there's not a lot of complicated things to learn. Everything is straight-forward C-style API calls and you've only got to learn about a few structs (the most important of which is SDL_Surface) before you can start doing something useful.

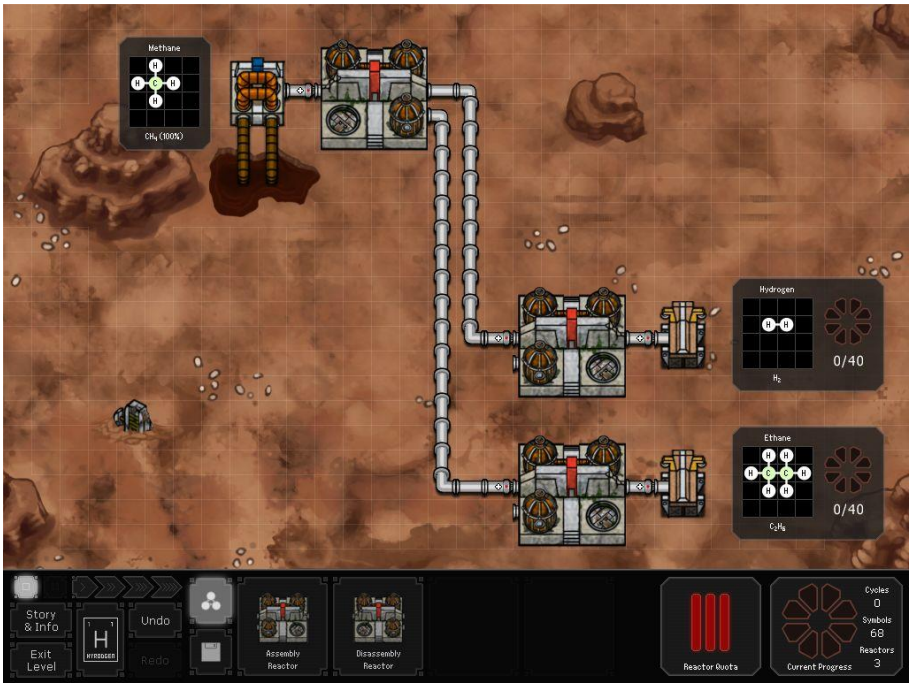


Figure 26 - Space Chem game (made with SDL)

GLFW

GLFW used to mean "OpenGL Framework" but just stands on its own as a free, open-source multi-platform library for OpenGL application development. It provides support for:

- Multiple windows and monitors
- Keyboard
- Mouse
- Gamepad
- Polling and Callbacks
- Window event Input

GLFW was designed for OpenGL development and since is the graphics option we have selected for this project this is probably the better choice to use.

Emsrcripten

A key consideration to the library we will use is if it works with Emsrcripten. Emsrcripten is a source-to-source compiler or transcompiler[8]. It takes C++ code and generates JavaScript so that our application can run within a browser with no changes.

Head over to http://beta.unity3d.com/jonas/AngryBots/ to play a browser version of the Unity game AngryBots.

[8] http://en.wikipedia.org/wiki/Emscripten

We will cover this tool in more detail when we get to the place in our project where we want to convert Sparky Game Engine to run a demo game on the browser. At this point in time there is no reason to believe that it will not be able to support GLFW with OpenGL.

Starting Sparky

Start with creating the Rendering Engine.

1. Download GLFW at http://www.glfw.org/.
2. Unzip the file.

The files that come with the Windows-based version are:

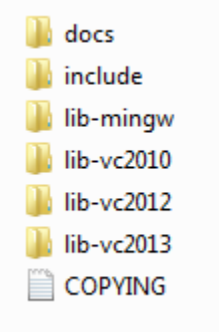


Figure 27 - Folders in GLFW

The folders we are interested in using Code::Blocks is the “include” and “lib-mingw” folders.

3. Copy the file in lib-mingw\glfw3.dll to your Code::Blocks MinGW bin directory. For me that will be C:\CodeBlocks\MinGW\bin.

A Simple GLFW Program

1. Open Code::Blocks
2. Create File → New Project → Empty Project named ExampleGLFW
3. Select File → New → File . . .

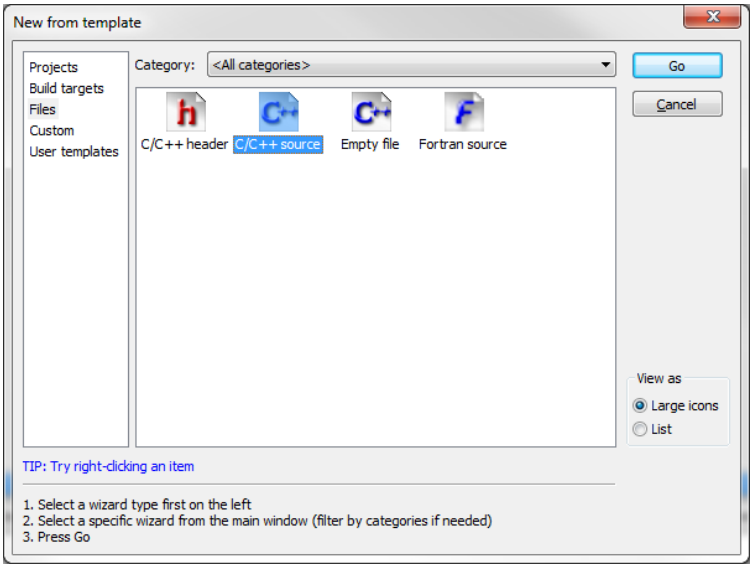


Figure 28 - Creating a C/C++ file

4. Highlight/Select C/C++ source icon as shown above and click on “Go”
5. Click “Next >” if you see the “Welcome to the new C/C++ source file wizard dialog box” (I recommend checking the “Skip this page next time” so it does not come up again.
6. Select C++ as the language option and click on “Next >”
7. Click on the … button near the “Filename with full path:” input box

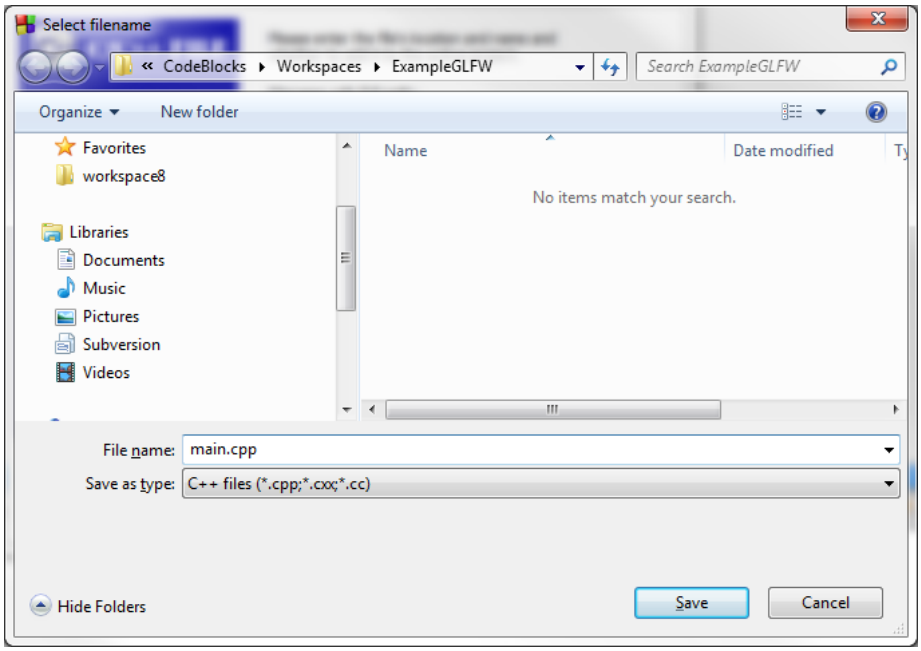


Figure 29 - Creating main.cpp for ExampleGLFW project

8. Enter the name of your *.cpp file. In our case it is main.cpp.
9. Click on “Save

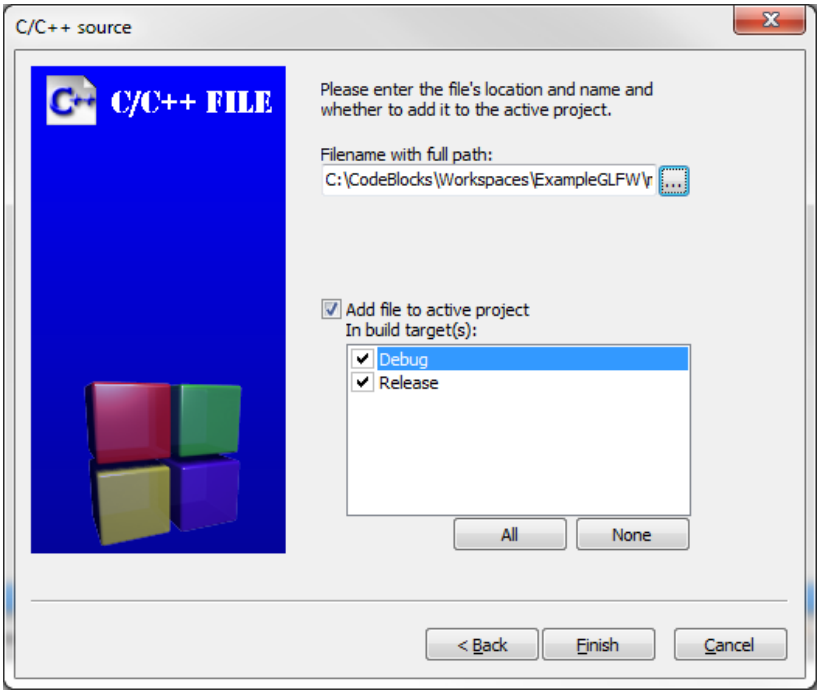


Figure 30 - Creating main.cpp file

10. Click on "All" to associate the file with the Debug and Release build targets.
11. Click on "Finish"

The editor will open with an empty C++ file.

12. Copy the code shown at: http://www.glfw.org/documentation.html that illustrates a simple GLFW program.

Table 4 - ExampleGLFW Program

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

int main(void)
{
    GLFWwindow* window;

    /* Initialize the library */
    if (!glfwInit())
        return -1;

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        return -1;
    }

    /* Make the window's context current */
    glfwMakeContextCurrent(window);
```

```
    /* Loop until the user closes the window */
    while (!glfwWindowShouldClose(window))
    {
        /* Render here */

        /* Swap front and back buffers */
        glfwSwapBuffers(window);

        /* Poll for and process events */
        glfwPollEvents();
    }

    glfwTerminate();
    return 0;
}
```

13. Run the Build.

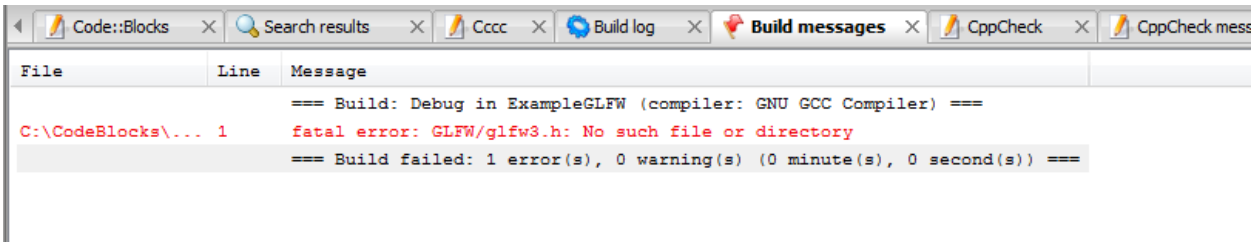


Figure 31 - Our first build error

The compiler cannot find the GLFW/glfw3.h include file referenced on the first line.

The convention we will use for our project will be to create a “Dependencies” folder in our project. Now we could opt to place all our dependencies under Code::Blocks compiler directories so we will not have to repeat these steps for every new project but then this will make our build environment dependent on our environment and IDE. I would like to upload as complete a version of the codebase to GitHub without references to any specific directory structure.

Create a folder by the name GLFW and copy the include directory shown in Figure 27.

14. Switch over to the Files Tab in the Management view window and navigate to the location of your current project.

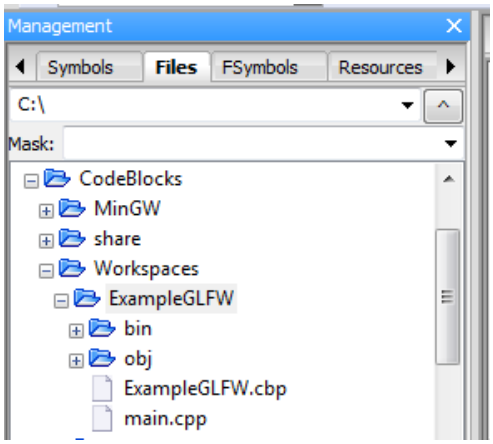


Figure 32 - Creating a folder

15. Right-click and select “New directory…”

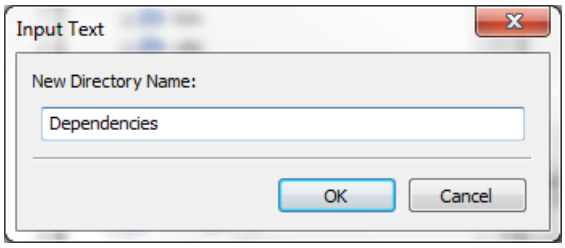


Figure 33 - Creating "Dependencies" directory

16. Highlight/Select the Dependencies directory and create the folder/directory named GLFW.

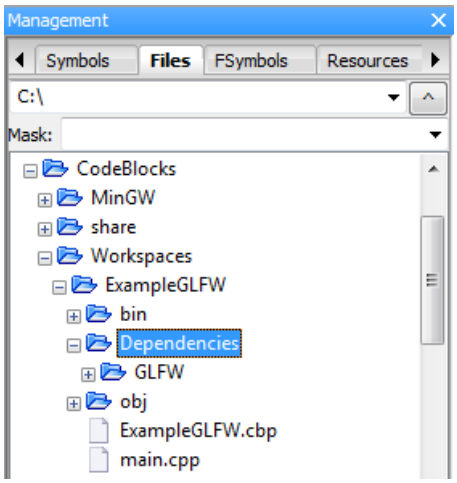


Figure 34 - Creating directory for include files

15. Copy the GLFW include directory to the GLFW Dependencies/GLFW folder.

16. Select Project → Build options… menu option

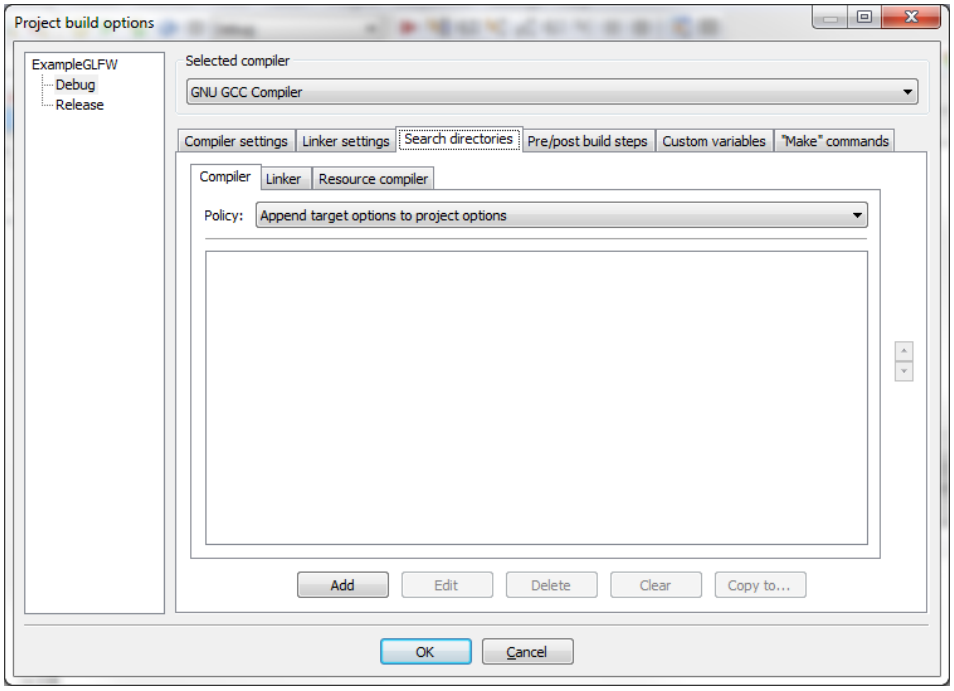


Figure 35 - Adding include to the search directories

17. Click on Add and enter: $(PROJECT_DIRECTORY)\Dependencies\GLFW\include

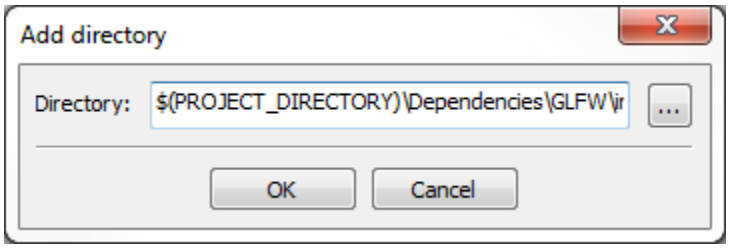


Figure 36 - Enter location of the GLFW include directories

18. Click OK, twice.
19. Now Build again.

The program successfully compiles but fails to locate the GLFW functions.

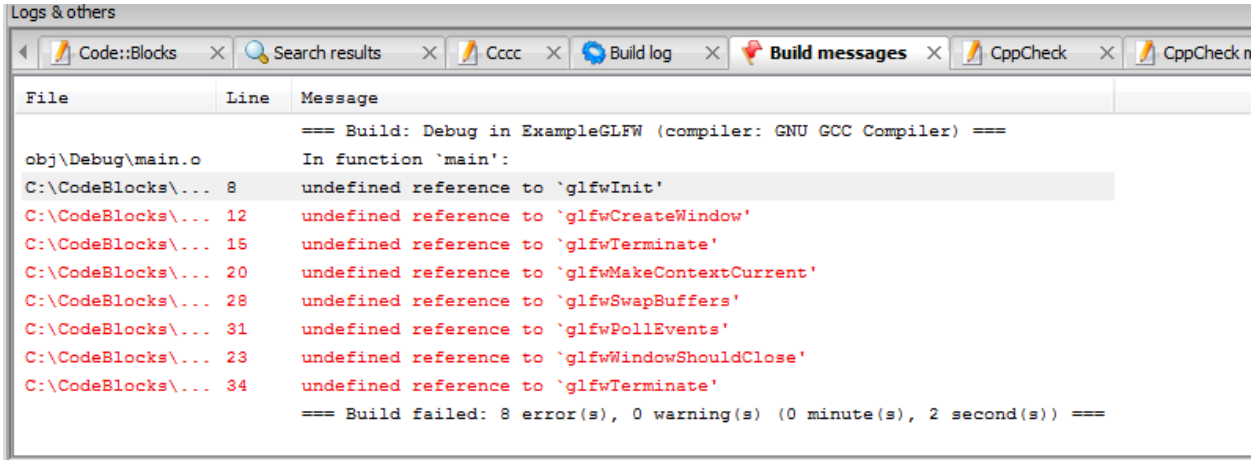


Figure 37 - Link failure

20. Copy the lib-mingw folder (from the unzipped files) to the Dependencies\GLFW folder.
21. Select Project → Build Options again.
22. Select the Linker settings tab

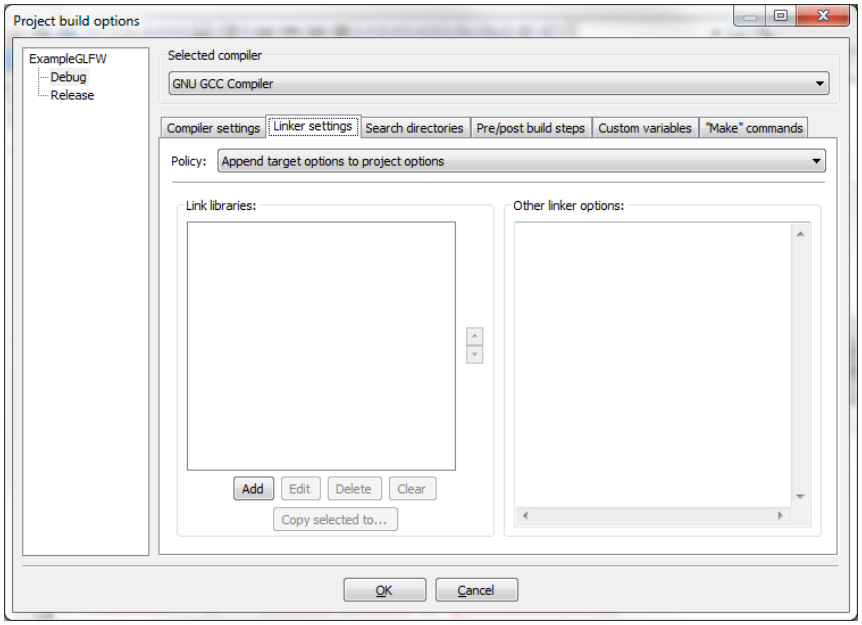


Figure 38 - Adding the GLFW link libraries

23. Click Add and enter
    - $(PROJECT_DIRECTORY)Dependencies\GLFW\lib-mingw\glfw3dll.a
    - $(PROJECT_DIRECTORY)Dependencies\GLFW\lib-mingw\libglfw3.a

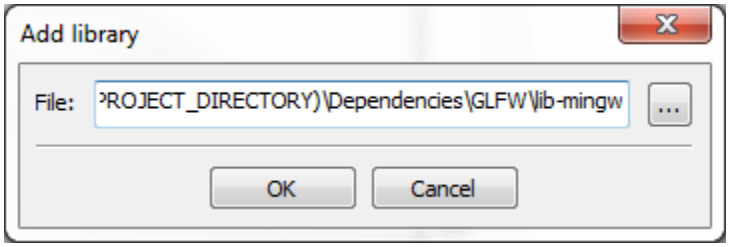


Figure 39 - Adding link libraries

24. Click OK, and Build again.

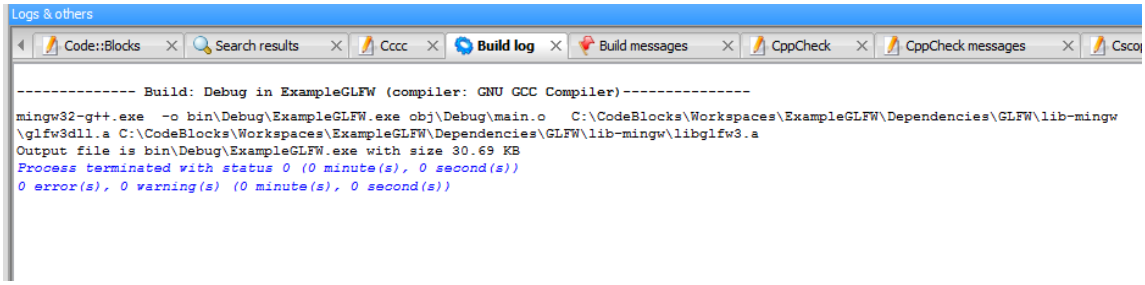


Figure 40 - Successful build

25. Run. A black window will be displayed.

Let's examine the program. First you should notice that unlike the Windows programs we examined earlier this file contains the standard and familiar `main` function.

```
int main(void)

{

	. . .

}
```

This is one of the key advantages to using a package such as SDL or GLFW they manage all the details of creating windows and drawing resources. All your GLFW based applications will require the `glfw3` header be included.

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

The header "defines all the constants, types and function prototypes of the GLFW API. It also includes the OpenGL header, and defines all the constants and types necessary for it to work on your platform." To keep your application cross-platform the GLFW website recommends the following:

- Do not include OpenGL headers since GLFW already handles them
- Do not include windows.h header since again GLFW already handles it
- If you do need to add these headers in order to access windows API not included by GLFW insert the headers before the glfw3.h.

```
GLFWwindow* window;
```

A `GLFWwindow` object encapsulates both an opaque window and a context. You will use the `glfwCreateWindow` function to create a window and the `glfwDestroyWindow` or `glfwTerminate` functions to destroy the window.

The first thing your program must to in order to get started using any GLFW functions is initialize the library.

```
    /* Initialize the library */
    if (!glfwInit())
        return -1;
```

If the function fails it will invoke the `glfwTerminate` function before returning. On success your code is responsible for invoking `glfwTerminate`.

The function `glfwInit` returns `GL_TRUE` on success or `GL_FALSE` if an error occurs.

The next step is to create the window using `glfwCreateWindow`.

```
/* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        return -1;
    }
```

```
GLFWwindow* glfwCreateWindow ( int              width,
                               int              height,
                               const char *     title,
                               GLFWmonitor *    monitor,
                               GLFWwindow *     share
                             )
```

Figure 41 - Format of glfwCreateWindow function

We specify the width and height of the drawing area in our window. You can specify the monitor (recommend you use NULL or the primary monitor). The share window is usually NULL or the windows to share resources with. The function returns the handle of the created window which is saved in our GLFWwindow object. If can use the function glfwWindowHint before window creation to set attributes that are different from the default (e.g GLFW_VISIBLE set to GL_FALSE to make your window invisble).

The next step is to make OpenGL context current on the specified window.

```
    /* Make the window's context current */

    glfwMakeContextCurrent(window);
```

Our application establishes a loop that exists when the flag indicating the window should be closed. "When the user attempts to close the window, either by pressing the close widget in the title bar or using a key combination like Alt+F4, this flag is set to 1." It is then the responsibility for the code to clean up by invoking glfwTerminate.

```
/* Loop until the user closes the window */

    while (!glfwWindowShouldClose(window))

                    :

    }
```

Inside the loop your code will:

- Draw or render the screen (absent from our example)
- Swap to the display the rendered screen
- Check for any events

"GLFW windows by default use double buffering. That means that each window has two rendered buffers; a front buffer and a back buffer. The front buffer is the one being displayed and the back buffer the one you render to."

```
        /* Swap front and back buffers */

        glfwSwapBuffers(window);
```

The glfwSwapBuffers function displays to the screen the image you rendered or drew and now want to show to the user in one complete transition.

What is double Buffering?

FROM: http://docs.oracle.com/javase/tutorial/extra/fullscreen/doublebuf.html

Double buffering is a technique for drawing graphics that show no flicker or tearing on the screen. The way it works is not to update the screen directly but to create an updated version of the screen in another area (a buffer) and when you have finished moving the aliens, killing or removing the debris and moving the player you then move or copy the updated screen to the video screen in one step or as quickly as possible when the video monitor is moving to re-set to draw a new screen.

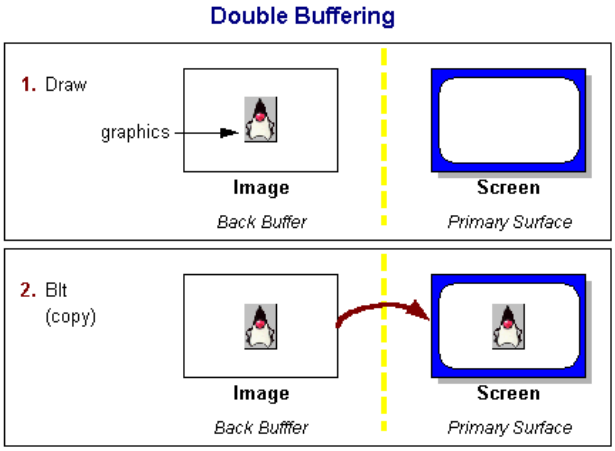


Figure 42 - Double buffering

In double buffering we reserve an area in memory (RAM) that we update and then copy or what most programmers refer to as blit (bit blit or bit block) the entire memory area into the video area. The screen surface is commonly referred to as the *primary surface*, and the offscreen image used for double-buffering is commonly referred to as the *back buffer*. You can actually (and often) use more than one back buffer.

We want our program to process any events (e.g. user closes the window, window is resized, etc) directed to the window we created.

```
        /* Poll for and process events */

        glfwPollEvents();
```

When the loop terminates all GLFW resources are cleaned up.

```
    glfwTerminate();

    return 0;
```

Creating Sparky Game Engine Project

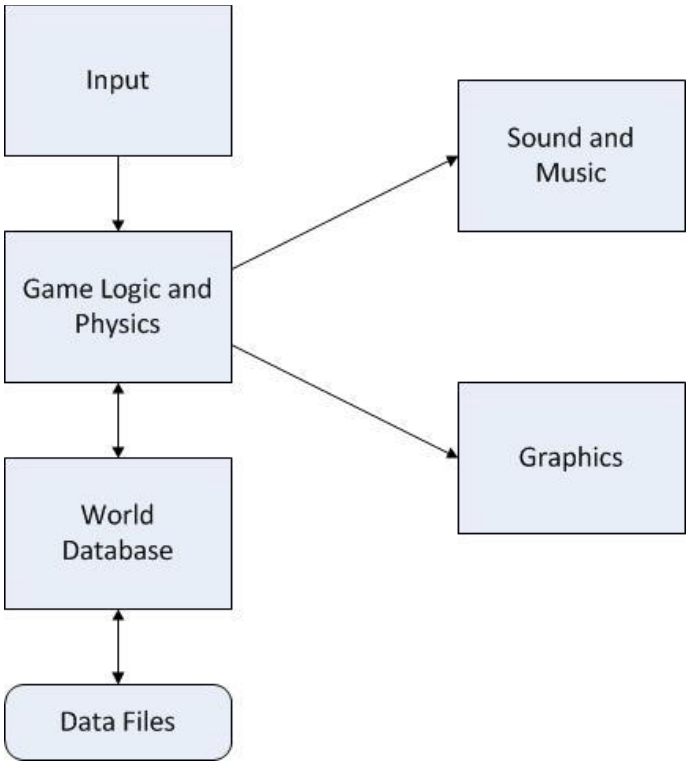
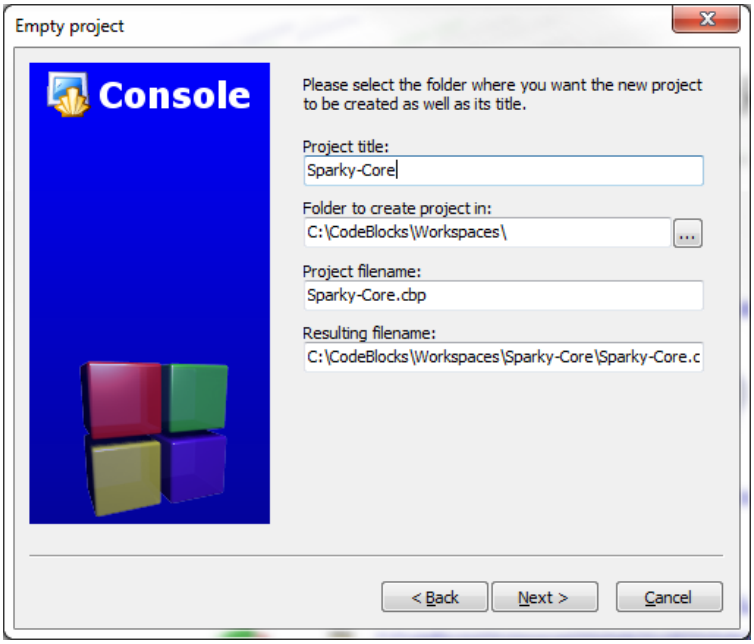


Figure 43 - Simple Game Architecture

The figure above illustrates the components one would what to have in a simple game architecture. The key aspects being implemented by the Sparky Game Engine is the Graphics, Input, and Sound Effects and Music. The specific game logic and physics will differ depending on the game being created.

In this episode we will begin our Creation of Sparky by creating our main program to initialize GLFW and a new Graphics class `Window` to handle the creation and management of the main game window.

1. Open Code::Blocks
2. Create File → New Project → Empty Project



3. As in our last project we will create a "Dependencies" folder to contain any additional libraries we use as part of our game engine.
4. In addition we will create a src/graphics folder. The src folder will contain all our game component folders and the graphics folder will hold all the graphic specific class files.

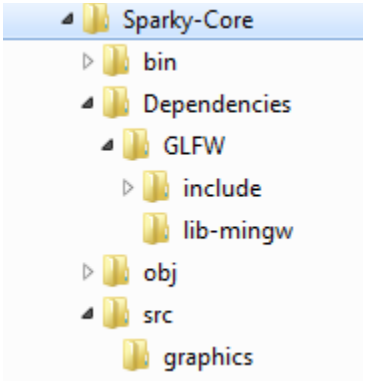


Figure 44 - Initial Sparky folder structure

5. Copy the required files into Dependencies\GLFW\include and lib-mingw (see previous program)
6. Set up the Project → Build Options → Search Directories to include Dependencies\GLFW\include
7. Set up the Project → Build Options → Linker Settings Tab with libraries opengl32, glfw3dll.a and libglfw3.a.
8. Create the file src/graphics/windows.h

Table 5- window.h

```
#pragma once

#include <iostream>
```

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

using namespace std;
namespace sparky { namespace graphics {

    class Window
    {
        private:
            const char *m_Title;
            int m_Width, m_Height;
            GLFWwindow* m_Window;

            bool m_Closed;

        public:
            Window(const char *name, int width, int height);
            ~Window();
            void clear() const;
            void update();
            bool closed() const;

            inline int getWidth() const
            {
                return m_Width;
            }
            inline int getHeight() const
            {
                return m_Height;
            }

        private:
            bool init();

    };

} }
```

The Window class will manage the creation and destruction of our GLFWwindow object.

9. Create the file src/graphics/window.cpp

Table 6 – window.cpp

```
#include "window.h"

namespace sparky { namespace graphics {

    void windowResize(GLFWwindow *window, int width, int height);

    Window::Window(const char *title, int width, int height)
    {
        m_Title = title;
        m_Width = width;
        m_Height = height;
```

```
        if (!init())
        {
            glfwTerminate();
        }
    }

    Window::~Window()
    {
        glfwTerminate();
    }

    bool Window::init()
    {
        /* Initialize the library */
        if (!glfwInit())
        {
            cerr << "Error Initializing GLFW library." << endl;
            return false;
        }

        m_Window = glfwCreateWindow(m_Width, m_Height, m_Title, NULL, NULL);

        if (!m_Window)
        {
            glfwTerminate();
            cerr << "Failed to create GLFW window!" << endl;
            return false;
        }

        glfwMakeContextCurrent(m_Window);
        glfwSetWindowSizeCallback(m_Window, windowResize);
        return true;
    }

    bool Window::closed() const
    {
        return glfwWindowShouldClose(m_Window) == 1;
    }
    void Window::clear() const
    {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    }
    void Window::update()
    {
        glfwPollEvents();
//        glfwGetFramebufferSize(m_Window, &m_Width, &m_Height);
        glfwSwapBuffers(m_Window);
    }

    void windowResize(GLFWwindow *window, int width, int height)
    {
         glViewport(0, 0, width, height);
    }

} }
```

The Window class handles:

- Initialization of the GLFW library
- The creation and destruction of the GLFWwindow object
- Manages system events
- Responds to window resizing

The Window class handles our drawing window.

The main is quite simple. We will augment it by drawing a white rectangle on a blue background to the screen using OpenGL commands.

10. Create the file main.cpp (outside our src directory).

Table 7 – main.cpp for episode 1

```
#include <GLFW/glfw3.h>
#include "src/graphics/window.h"

using namespace std;

int main(void)
{
    using namespace sparky;
    using namespace graphics;

    Window window("Sparky", 960, 540);
    glClearColor(0.2f, 0.3f, 0.8f, 1.0f);

    cout << "OpenGL version: " << glGetString(GL_VERSION) << endl;

    while (!window.closed())
    {
        window.clear();

        glBegin(GL_QUADS);
        glVertex2f(-0.5f, -0.5f);
        glVertex2f(-0.5f,  0.5f);
        glVertex2f( 0.5f,  0.5f);
        glVertex2f( 0.5f, -0.5f);
        glEnd();
        window.update();
    }
    return 0;
}
```

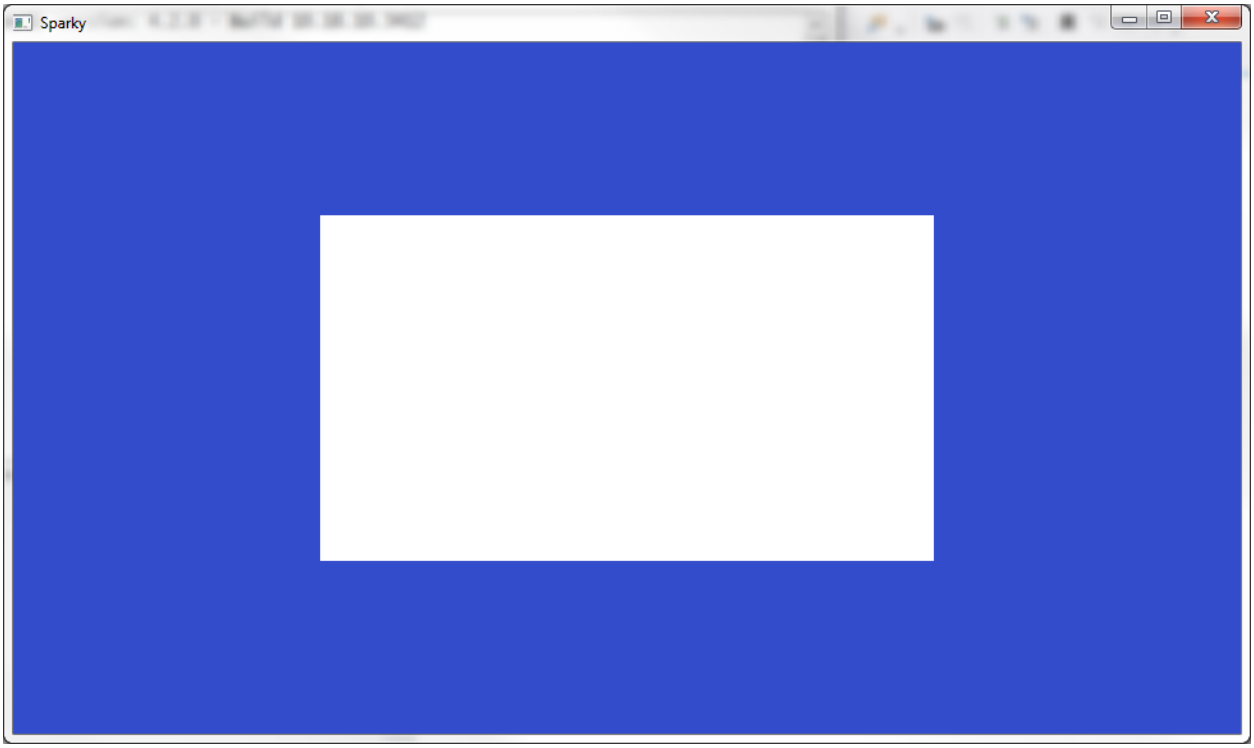


Figure 45- Episode 1 results

The final Workspace structure appears as shown below.

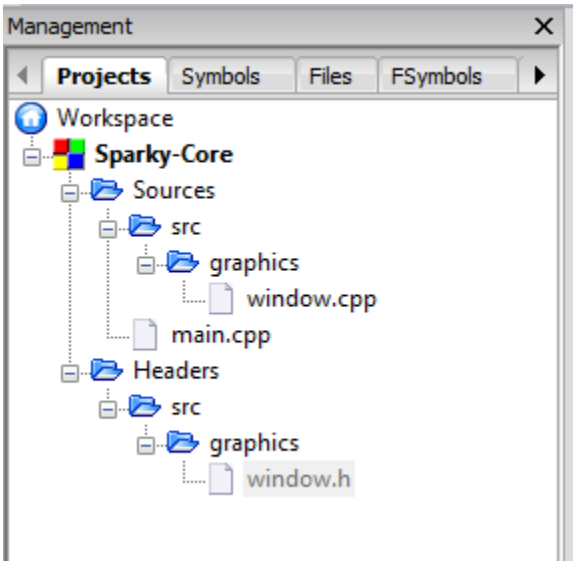


Figure 46 – Episode 1 Workplace structure

The objective of creating and displaying our game window is done.

Episode 2 – GLEW

GLEW stands for The OpenGL Extension Wrangler Library. GLEW is a cross-platform C/C++ library that helps in querying and loading OpenGL extensions. It provides a run-time mechanism for determining which OpenGL extensions are supported on the target platform.

GLEW is regarded as an OpenGL Loading Library. This type of library loads pointers to OpenGL functions at runtime, core as well as extensions[9]. This type of library is required in order to access functions from OpenGL versions above 1.1 on most platforms. Extension loading libraries also abstracts away the difference between the loading mechanisms on different platforms.

GLEW provides access to all GL entrypoints. It supports Windows, MaxOS X, Linux and FreeBSD.

1. Head over to http://glew.sourceforge.net/index.html to download the latest version.

Why use GLEW? The Microsoft Windows OS only provides OpenGL up to version 1.1. The latest release of OpenGL is at 4.5 (released August 11, 2014). You may have a graphics card that supports features and extensions available in OpenGL 4.5. GLEW handles the task of determining the features and functions supported by our graphics drivers and make them available to our programs via function pointers.

2. Download and unpack the Windows binaries.

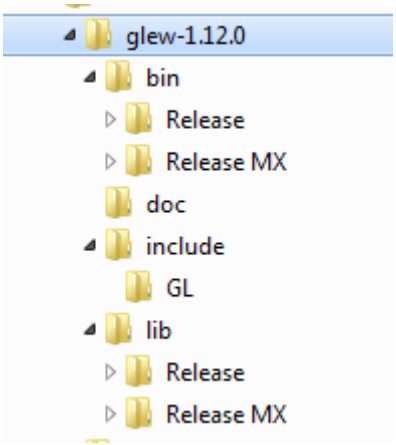


Figure 47 – GLEW file extraction

3. Open the directory/folder lib/Release/Win32

You will see two versions of the glew32 lib file. We will use the static version glew32s.

4. Create a folder named GLEW under your Dependencies folder.
5. Create a folder named lib under Dependencies/GLEW and copy the glew32s file from step 3.

[9] https://www.opengl.org/wiki/OpenGL_Loading_Library

6. Copy the include directory to Dependencies/GLEW.
7. Projects → Build Options…
    a. Select “Search directories” tab
    b. Add $(PROJECT_DIRECTORY)\Dependencies\GLEW\include

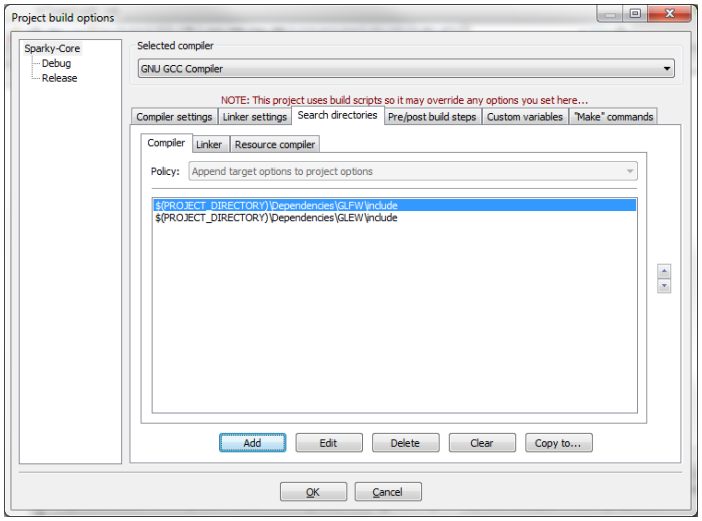


Figure 48 – Adding GLEW include directory

8. Projects → Build options..
    a. Select Linker settings tab
    b. Add $(PROJECT_DIRECTORY)Dependencies\GLEW\lib\glew32.lib

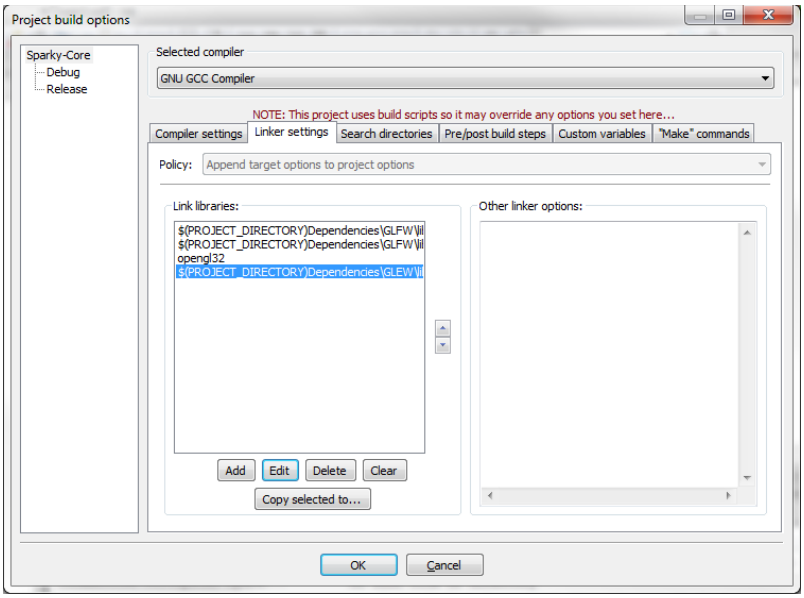


Figure 49 – Adding GLEW lib file

9. Copy the glew32.dll (at bin\Release\Win32) to your Code::Blocks mingw bin directory(e.g. C:\CodeBlocks\MinGW\bin directory)
10. Test that everything is set up by
    a. Build → Clean (to remove everything)
    b. Build

Table 8 – Adding GLEW to our build

```
-------------- Build: Debug in Sparky-Core (compiler: GNU GCC Compiler)---------------
mingw32-g++.exe -Wall -g -IC:\CodeBlocks\Workspaces\Sparky-Core\\Dependencies\GLFW\include -
IC:\CodeBlocks\Workspaces\Sparky-Core\\Dependencies\GLEW\include -c
C:\CodeBlocks\Workspaces\Sparky-Core\main.cpp -o obj\Debug\main.o
mingw32-g++.exe -Wall -g -IC:\CodeBlocks\Workspaces\Sparky-Core\\Dependencies\GLFW\include -
IC:\CodeBlocks\Workspaces\Sparky-Core\\Dependencies\GLEW\include -c
C:\CodeBlocks\Workspaces\Sparky-Core\src\graphics\window.cpp -o
obj\Debug\src\graphics\window.o
mingw32-g++.exe  -o bin\Debug\Sparky-Core.exe obj\Debug\main.o
obj\Debug\src\graphics\window.o   C:\CodeBlocks\Workspaces\Sparky-Core\Dependencies\GLFW\lib-
mingw\glfw3dll.a C:\CodeBlocks\Workspaces\Sparky-Core\Dependencies\GLFW\lib-mingw\libglfw3.a
-lopengl32 C:\CodeBlocks\Workspaces\Sparky-Core\Dependencies\GLEW\lib\glew32.lib
Output file is bin\Debug\Sparky-Core.exe with size 961.05 KB
Process terminated with status 0 (0 minute(s), 2 second(s))
0 error(s), 0 warning(s) (0 minute(s), 2 second(s))
```

We will now use it.

11. Open the window.h file and add the glew.h include file before the glfw include file as shown:

```
#pragma once

#include <iostream>
#include <GL/glew.h>
#include <GLFW/glfw3.h>
            :
            :
```

12. In our main.cpp file also add the same include before the glfw3.h include.

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include "src/graphics/window.h"
            :
            :
```

You need to initialize the GLEW library after the OpenGL context has been set. Add the following lines in the Windows init() function.

```
        glfwMakeContextCurrent(m_Window);
        glfwSetWindowSizeCallback(m_Window, windowResize);

        if (glewInit() != GLEW_OK)
        {
            cerr << "Could not initialize GLEW!" << endl;
            return false;
        }
        return true;
```

The next thing to do is to actually test things out by referencing an OpenGL function beyond version 1.1.

At this point I found it necessary to make sure my link libraries were ordered correctly and included the glew.dll file. This was required in to get the new code to run.

13. Copy the glew32.dll to the GLEW\lib directory under Dependencies.

14. Add glew32.dll to the linker settings

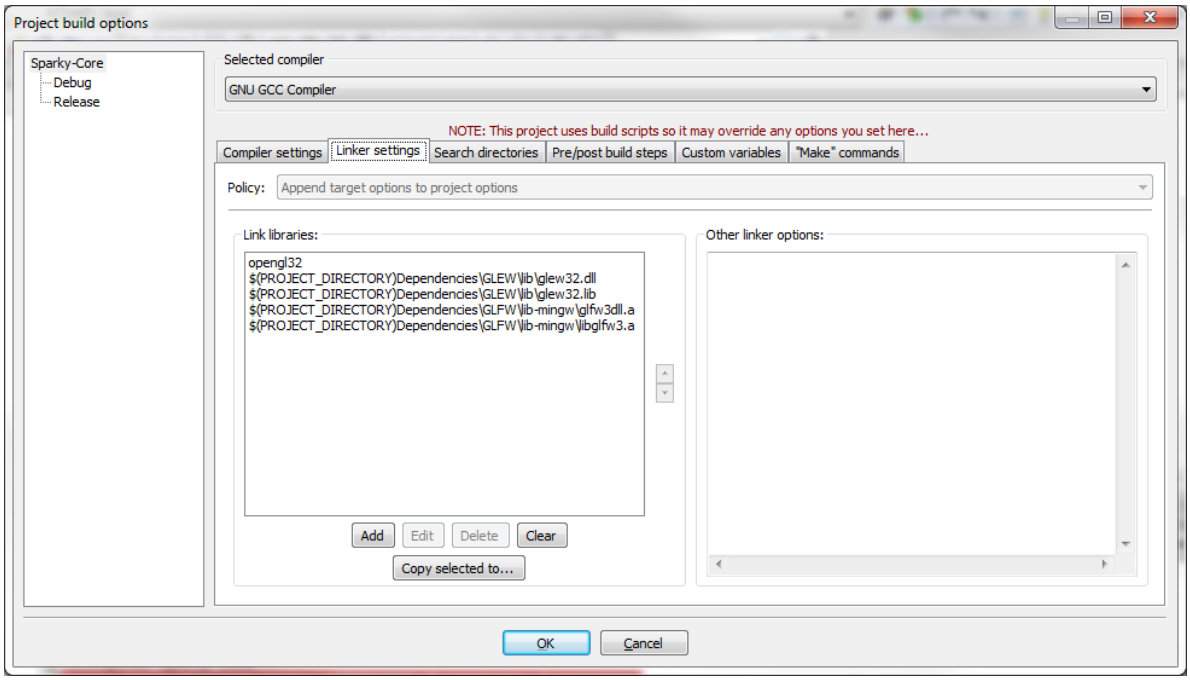


Figure 50 – Link libraries

Note the order of the libraries.

14. Change the main as shown below:

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include "src/graphics/window.h"

using namespace std;

int main(void)
{
    using namespace sparky;
    using namespace graphics;

    Window window("Sparky", 960, 540);
    glClearColor(0.2f, 0.3f, 0.8f, 1.0f);

    GLuint vao;
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    cout << "OpenGL version: " << glGetString(GL_VERSION) << endl;

    while (!window.closed())
    {
        window.clear();
#if 0
        glBegin(GL_QUADS);
        glVertex2f(-0.5f, -0.5f);
        glVertex2f(-0.5f,  0.5f);
        glVertex2f( 0.5f,  0.5f);
        glVertex2f( 0.5f, -0.5f);
        glEnd();
#endif
        glDrawArrays(GL_ARRAY_BUFFER, 0, 6);
        window.update();
    }
    return 0;
}
```

15. Clean, build and run the code.

You will see an empty blue screen.

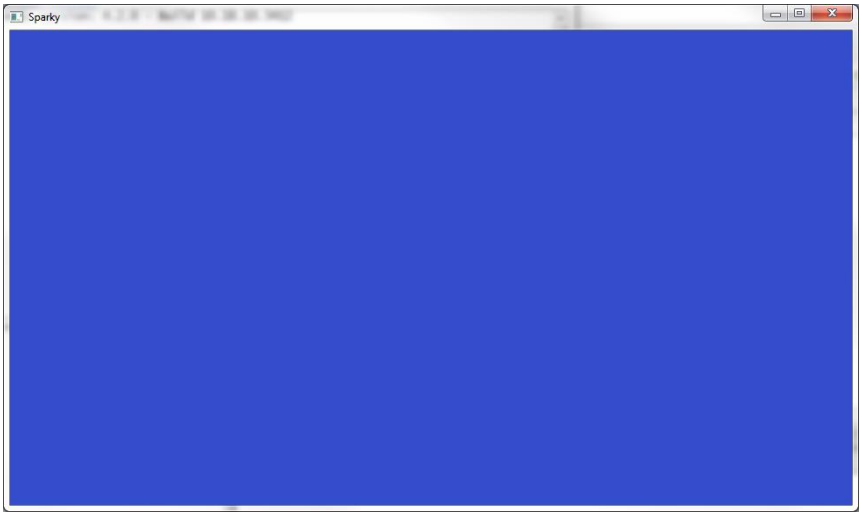


Figure 51 - Testing OpenGL features using GREW.

16. Restore the code that displays a rectangle by removing the `#if 0` and `#endif`.

Everything should still build and run as before but now the white rectangle shows up again.

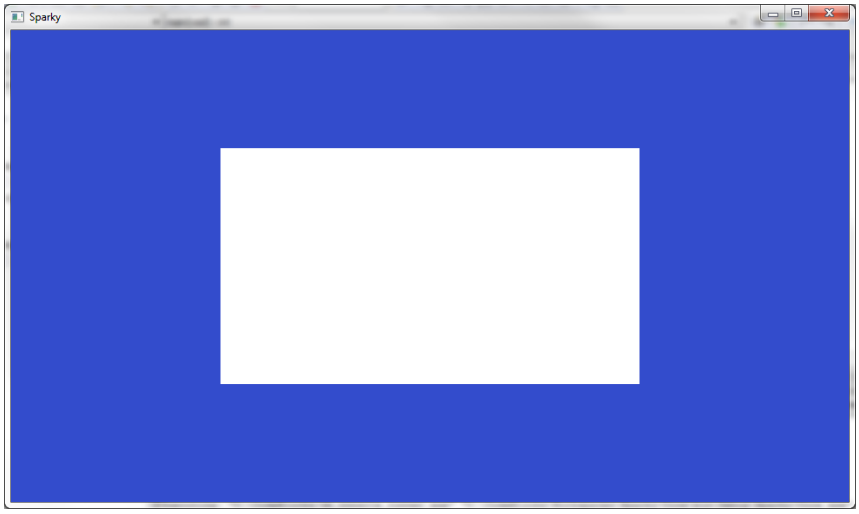


Figure 52 - Sparky Game Engine at end of Episode #2

Episode 3 – Handling Input

In this episode we add input handling to the Sparky Game Engine.

We will add keyboard and mouse handling in this episode.

1. Open up the Sparky-Core project in Code::Blocks.
2. Change to the Files tab and navigate to the Sparky-Core project directory on your file system.
3. Create a new folder under src directory called input

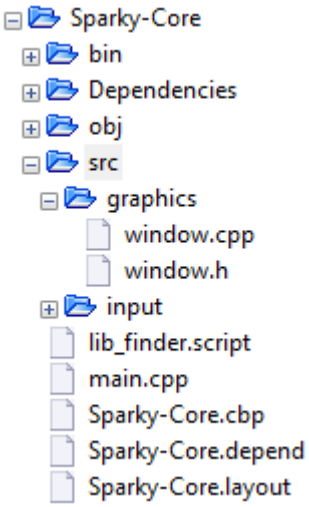


Figure 53 - Adding a new src/Input folder

Most of this material comes from the GLFW documentation at http://www.glfw.org/docs/latest/input.html.

GLFW handles many types of inputs. Some of the input processes can be polled (e.g. time) or only be received via callback functions (e.g. scrolling) and some that can be handled by polling and callbacks. It is recommended that if GLFW provides a callback mechanism that it be preferred over polling.

Using a callback function will require that we do things:

- Create the callback function with the parameter values expected by GLFW
- Inform GLFW of our callback function

As an example if we want GLFW to invoke our keyboard call back function we must create a function in this format:

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods);
```

The name does not have to be key_callback but the return value of void and the five parameters must all be in our callback function.

The next thing is to inform GLFW to invoke our callback function:

```
glfwSetKeyCallback(window, key_callback);
```

The first argument is a GLFWwindow* and the second is our call back function.

When does our callback function get invoked?

It is our responsibility to either use `glfwPollEvents()` or `glfwWaitEvents()` to get GLFW to process events with the underlying windowing system.

All callback functions receive a window handle. Each window has a user pointer that can be set with the function `glfwSetWindowUserPointer` and fetched with the function `glfwGetWindowUserPointer`. This can be used by developers to access non-global structures or objects from your callbacks.

We will use the above feature to associate with each GLFWwindow our own object window class Window.

GLFW needs to communicate regularly with the window system both in order to receive events and to show that the application hasn't locked up. Event processing must be done regularly while you have visible windows and is normally done each frame after buffer swapping.

There are two functions for processing pending events. `glfwPollEvents()`, processes only those events that have already been received and then returns immediately. If you only need to update the contents of the window when you receive new input, `glfwWaitEvents()` is a better choice.

Do not assume that callback functions will ONLY be called through using the two functions above. Some window systems will directly send events to the application, which will in turn invoke a callback function outside the event processing phase.

Keyboard Input

Keyboard input is divided input two categories: key events and character events. Key events relate to actual physical keyboard keys, whereas character events relate to the Unicode code points generated by pressing some of them.

Keys and characters do not map one-to-one. A single key press may generate several characters, and a single character may require several keys to produce.

Key input

We will use the `glfwSetKeyCallback` to be notified when a physical key is pressed or released or when it repeats.

```
glfwSetKeyCallback(window, key_callback);
```

The callback function receives the keyboard key, a platform-specific scancode, key action and modifier bits. See Appendix D for the GLFW macros to use to detect key input.

This is how your callback function will look like:

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if (key == GLFW_KEY_E && action == GLFW_PRESS)
        activate_airship();
}
```

The action can be one of the following:

- GLFW_PRESS
- GLFW_REPEAT
- GLFW_RELEASE

If the value for the key is unknown (corresponds to a special keyboard key) its value will be `GLFW_KEY_UNKNOWN`. The scancode is unique platform-specific code.

You can obtain the state of a particular key by polling the key using `glfwGetKey()`.

```
int state = glfwGetKey(window, GLFW_KEY_E);
if (state == GLFW_PRESS)
    activate_airship();
```

The state can be either `GLFW_PRESS` or `GLFW_RELEASE`, which is the current cached key event state not the existing key state. It is possible to miss a key event so it may make sense to establish a key callback. Another method to use is to establish `GLFW_STICKY_KEYS` input mode.

```
glfwSetInputMode(window, GLFW_STICKY_KEYS, 1);
```

When the sticky keys mode is enabled, the pollable state of a key will remain `GLFW_PRESS` until the state of that key is polled with `glfwGetKey()`.

(can be skipped since not used at this time for this Episode)

Text input

GLFW supports text input in the form of a stream of Unicode code points, as produced by the operating system text input system.

There are two callbacks for receiving Unicode code points. If you wish to offer regular text input, set a character callback.

```
glfwSetCharCallback(window, character_callback);
```

The callback function receives Unicode code points for key events that would have led to regular text input and generally behaves as a standard text field on that platform.

```
void character_callback(GLFWwindow* window, unsigned int codepoint)
{
}
```

If you wish to receive even those Unicode code points generated with modifier key combinations that a plain text field would ignore, or just want to know exactly what modifier keys were used, set a character with modifiers callback.

```
glfwSetCharModsCallback(window, charmods_callback);
```

The callback function receives Unicode code points and modifier bits.

```
void charmods_callback(GLFWwindow* window, unsigned int codepoint, int mods)
{
}
```

Since it is possible for multiple keys to be pressed we want to be able to capture/remember all the keys being pressed. The same thing applies to the mouse buttons. We will create an array to hold all the possible key press key values.

```
bool m_Keys[MAX_KEYS];
```

The array will be added to the Window class as a private member variable.

```
#define MAX_KEYS    1024
```

The define statement is used to determine the size of the array. The idea behind this array is that when our keyboard callback function is invoked by GLFW we will update the corresponding m_Keys[key] entry in the array with `true` if the key is pressed (or held down) and `false` when the key is released. The GLFW values we will examine are: GLFW_PRESS, GLFW_REPEAT (for key press) and GLFW_RELEASE (for key release).

Note: Our program is going to be slightly different than the one created for this episode since gcc does not support `static friend` functions. We will define and create the following public class methods:

```
void setKey(unsigned int keyCode, bool val);
```

```
bool isKeyPressed(unsigned int keyCode) const;
```

The `setKey` method is invoked to update the corresponding array entry and `isKeyPressed` returns the current value (`true` or `false`) in the `m_Keys` array.

The next definition to add to our window.h file is the for the keyboard callback function.

```
static void key_callback(GLFWwindow* window, int key, int scancode, int action, int
mods);
```

We define this function as static. In order, to get it to work we need the function to have access to our Window object.

We will use the glfwSetWindowUserPointer() function in the initialization Window method to establish an association between the GLFW window and our window object.

```
glfwSetWindowUserPointer(m_Window, this);
```

We need to add this *after* the window context is created. In the same init() method for the Window object we will establish the keyboard callback function:

```
glfwSetKeyCallback(m_Window, key_callback);
```

The new functions we add to our window.cpp class file are:

```
void Window::setKey(unsigned int keyCode, bool val)
{
	if (keyCode >= 0 && keyCode < MAX_KEYS)
	{
		m_Keys[keyCode] = val;
	}
}
```

The method checks that the keycode will fall into the array and updates the keyCode entry with the value provided (remember `true` when key is pressed or held/repeated and `false`, otherwise).

```
bool Window::isKeyPressed(unsigned int keyCode) const
{
	if (keyCode >= 0 && keyCode < MAX_KEYS)
	{
		return m_Keys[keyCode];
	}
	// TODO: Log this!
	return false;
}
```

This method returns the current value for the key of interest.

In order to make this work we need to initialize the m_Keys array in the Window constructor:

```
    for (int i=0; i < MAX_KEYS; i++)
    {
        m_Keys[i] = false;
    }
```

The final method in this section to add is our keyboard callback function.

```
void Window::key_callback(GLFWwindow* window, int key, int scancode, int action,
                                int mods)
{
    Window* win = (Window*) glfwGetWindowUserPointer(window);
    win->setKey(key, action != GLFW_RELEASE);
}
```

The function uses the `glfwGetWindowUserPointer` to obtain the Window object associated with the current `GLFWWindow*` where the key event took place (this allows us to create more than one window!). We then use the `setKey` method to update the corresponding `m_Keys` entry.

We can test our code by adding the following code to our main.cpp file:

Table 9 - Testing keyboard presses

```
                        :
                        :
while (!window.closed())
{
   window.clear();
   if (window.isKeyPressed(GLFW_KEY_A))
   {
          cout << "Key A pressed" << endl;
   }
                        :
                        :
```

Mouse Input

Mouse input comes in the form of:

- Cursor motion
- Button presses
- Scrolling offsets

We can also change the mouse cursor appearance.

Cursor position

To obtain notification when the cursor moves over the game window, set the cursor position callback.

```
glfwSetCursorPosCallback(window, cursor_pos_callback);
```

The function receives the sub-pixel cursor position:

```
static void cursor_position_callback(GLFWwindow* window, double xpos, double ypos)
{
}
```

The cursor position can also be polled using `glfwGetCursorPos()`.

```
double xpos, ypos;
glfwGetCursorPos(window, &xpos, &ypos);
```

We will create two new member variables to track the current location of the mouse in the window generating the event:

```
double mx, my;
```

The following public methods shall be added to our window.h file:

```
void setMousePosition(double x, double y);
void getMousePosition(double& x, double& y) const;
```

The setMousePosition will be used by our mouse position callback function to update the mx and my. The getMousePosition will be used by our application to obtain the current location of the mouse in our window.

The callback function for the cursor position shall be added as a private static function.

```
static void cursor_position_callback(GLFWwindow* window, double xpos, double ypos);
```

The function follows the GLFW prescription for setting up this callback function.

In our windows.cpp file we had the following details:

```
void Window::setMousePosition(double x, double y)
{
	mx = x;
	my = y;
}

void Window::getMousePosition(double& x, double& y) const
{
```

```
            x = mx;
            y = my;
    }

void Window::cursor_position_callback(GLFWwindow* window, double xpos, double ypos)
{
    Window* win = (Window*) glfwGetWindowUserPointer(window);
    win->setMousePosition(xpos, ypos);
}
```

The functions are self-explanatory. The only missing elements now is establishing our cursor position callback function with GLFW and testing.

We add the following line to our Window init() method:

```
glfwSetCursorPosCallback(m_Window, cursor_position_callback);
```

We can test that it works by adding the following code to the same location in main.cpp we tested our keyboard functions:

```
double x,y;
window.getMousePosition(x, y);
cout << "x: " << x << "\ty: " << y << endl;
```

(can be skipped since not used at this time for this Episode)

Cursor modes

The GLFW_CURSOR input mode provides several cursor modes for special forms of mouse motion input. By default, the cursor mode is GLFW_CURSOR_NORMAL, meaning the regular arrow cursor (or another cursor set with `glfwSetCursor`) is used and cursor motion is not limited.
If you wish to implement mouse motion based camera controls or other input schemes that require unlimited mouse movement, set the cursor mode to GLFW_CURSOR_DISABLED.

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

This will hide the cursor.
If you wish the cursor to become hidden when it is over a window, set the cursor mode to GLFW_CURSOR_HIDDEN.

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_HIDDEN);
```

To restore the cursor use:

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
```

Cursor objects

You can elect to set up a custom or system theme cursor image, using the `glfwCreateCursor` or `glfCreateStandardCursor` and destroyed with `glfwDestroyCursor` or `glfwTerminate`.

Custom cursor creation

```
unsigned char pixels[16 * 16 * 4];
memset(pixels, 0xff, sizeof(pixels));

GLFWimage image;
image.width = 16;
image.height = 16;
image.pixels = pixels;

GLFWcursor* cursor = glfwCreateCursor(&image, 0, 0);
```

Standard cursor creation

A cursor with a standard shape from the current system cursor theme can be created with glfwCreateStandardCursor.

```
GLFWcursor* cursor = glfwCreateStandardCursor(GLFW_HRESIZE_CURSOR);
```

Cursor destruction

```
glfwDestroyCursor(cursor);
```

Cursor setting

A cursor can be set as current for a window with glfwSetCursor.

```
glfwSetCursor(window, cursor);
```

Once set, the cursor image will be used as long as the system cursor is over the client area of the window and the cursor mode is set to GLFW_CURSOR_NORMAL.
To remove a cursor from a window:

```
glfwSetCursor(window, NULL);
```

Cursor enter/leave events

If you wish to be notified when the cursor enters or leaves the client area of a window, set a cursor enter/leave callback.

```
glfwSetCursorEnterCallback(window, cursor_enter_callback);
```

The callback function receives the new classification of the cursor.

```
void cursor_enter_callback(GLFWwindow* window, int entered)
{
    if (entered)
    {
        // The cursor entered the client area of the window
    }
    else
    {
        // The cursor left the client area of the window
    }
}
```

Mouse button input

Set a mouse button callback to be notified when the mouse button is pressed.

```
glfwSetMouseButtonCallback(window, mouse_button_callback);
```

The callback function receives the mouse button action and modifier bits.

```
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
    if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS)
        popup_menu();
}
```

The action is either GLFW_PRESS or GLFW_RELEASE.

Mouse button states can be polled with glfwGetMouseButton.

```
int state = glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT);
if (state == GLFW_PRESS)
    upgrade_cow();
```

The function only returns cached mouse button event state. It does not poll the system for the current state of the mouse button.

Just as the keys, mouse event state changes can be missed. It is recommended that a mouse callback function be used and/or GLFW_STICKY_MOUSE_BUTTONS input mode.

We will track the mouse buttons using a similar concept we used for the key board. We will create an array to hold the "state" of the mouse button where `true` indicates it is being pressed and `false` to indicate otherwise.

```
bool m_MouseButtons[MAX_BUTTONS];
```

We will use a define statement to hold the number of buttons we think our mouse will ever have:

```
#define MAX_BUTTONS 32
```

We will create the following public methods to handle setting and getting mouse button state:

```
void setMouseButton(unsigned int button, bool val);
bool isMouseButtonPressed(unsigned int button) const;
```

The setMouseButton shall be used by our mouse button callback function and the application shall obtain the state of the mouse button of interest by using the isMouseButtonPressed.

The callback function follows the GLFW rules for this type of callback:

```
static void mouse_button_callback(GLFWwindow* window, int button, int action, int mods);
```

We will initialize the m_MouseButtons array in the Window constructor:

```
for (int i=0; i < MAX_BUTTONS; i++)
{
    m_MouseButtons[i] = false;
}
```

And as before establish the callback in the Window init() method:

```
glfwSetMouseButtonCallback(m_Window, mouse_button_callback);
```

The details of the new functions can be found below in the detailed code for this episode.

(can be skipped since not used at this time for this Episode)

Scroll input

If you wish to be notified when the user scrolls the mouse wheel or uses the touchpad set up a scroll callback:

```
glfwSetScrollCallback(window, scroll_callback);
```

The callback function receives two-dimension scroll offsets:

```
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
}
```

Joystick Input

The joystick functions support up to sixteen joysticks, ranging from GLFW_JOYSTICK_1, … GLFW_JOYSTICK_LAST.
You can test if a joystick is present by using the glfwJoystickPresent function as shown.

```
int present = glfwJoystickPresent(GLFW_JOYSTICK_1);
```

Joystick state is updated/obtained when a joystick function is called.

Joystick axis states

The positions of all axes of a joystick are returned by the function glfwGetJoystickAxes.

```
int count;
const float* axes = glfwGetJoystickAxes(GLFW_JOYSTICK_1, &count);
```

Each element in the returned array is between -1.0 to 1.0.

Joystick button states

The states of all buttons of a joystick are returned by the function glfwGetJoystickButtons.

```
int count;
const unsigned char* axes = glfwGetJoystickButtons(GLFW_JOYSTICK_1, &count);
```

Each element in the returned array is either GLFW_PRESS or GLFW_RELEASE.

Joystick name

The name of the joystick is returned using the function glfwGetJoystickName().

```
const char* name = glfwGetJoystickName(GLFW_JOYSTICK_1);
```

The joystick name is not guaranteed to be unique but the joystick token is.

Time Input

GLFW provides high-resolution time input, in seconds, with glfwGetTime.

```
double seconds = glfwGetTime();
```

The function returns the number of seconds since the timer was started and that would start when the GLFW gets initialized using glfwInit().
You can also modify/update the reference time with the function glfwSetTime().

```
glfwSetTime(4.0);
```

Clipboard Input and Output

You can obtain the clipboard data if it contains UTF-8 encoded string using the GLFW function glfwGetClipboardString.

```
const char* clipboard = glfwGetClipboardString(window);
```

You can also send string data to the clipboard.

```
glfwSetClipboardString(window, "A string with words in it");
```

Path drop Input

Most platforms allow users to drag and drop files or folders on a window (usually the window e.g. Windows explorer will move or copy the file or folder). You can set up the callback function glfwSetDropCallback() to obtain the file or folder path name.

```
glfwSetDropCallback(window, drop_callback);
```

The callback function will have the following format:

```
void drop_callback(GLFWwindow* window, int count, const char** paths)
{
    int i;
    for (i = 0;  i < count;  i++)
        handle_dropped_file(paths[i]);
}
```

You code will need to save the contents of the array paths since it will no longer be available after the callback function completes.

Episode 3 final code files:

Table 10 - Episode 3 main.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include "src/graphics/window.h"

using namespace std;

int main(void)
{
    using namespace sparky;
```

```
    using namespace graphics;

    Window window("Sparky", 960, 540);
    glClearColor(0.2f, 0.3f, 0.8f, 1.0f);

    GLuint vao;
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    cout << "OpenGL version: " << glGetString(GL_VERSION) << endl;

    while (!window.closed())
    {
        window.clear();
        /*
        if (window.isKeyPressed(GLFW_KEY_A))
        {
            cout << "Key A pressed" << endl;
        }

        if (window.isMouseButtonPressed(GLFW_MOUSE_BUTTON_LEFT))
        {
            cout << "Left Mouse pressed" << endl;
        }
        */
        if (window.isMouseButtonPressed(GLFW_MOUSE_BUTTON_LEFT))
        {
            double x,y;
            window.getMousePosition(x, y);
            cout << "x: " << x << "\ty: " << y << endl;
        }
        glBegin(GL_QUADS);
        glVertex2f(-0.5f, -0.5f);
        glVertex2f(-0.5f,  0.5f);
        glVertex2f( 0.5f,  0.5f);
        glVertex2f( 0.5f, -0.5f);
        glEnd();

        glDrawArrays(GL_ARRAY_BUFFER, 0, 6);
        window.update();
    }
    return 0;
}
```

We did not make any substantial changes to main.cpp other than to add test code for our new keyboard, mouse position and mouse button methods.

Table 11 - Episode 3 window.h file

```
#pragma once
#include <iostream>

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

```
#define MAX_KEYS    1024
#define MAX_BUTTONS 32

using namespace std;
namespace sparky { namespace graphics {

    class Window
    {
        private:
            const char *m_Title;
            int m_Width, m_Height;
            GLFWwindow* m_Window;

            bool m_Closed;

            bool m_Keys[MAX_KEYS];
            bool m_MouseButtons[MAX_BUTTONS];

            double mx, my;

        public:
            Window(const char *name, int width, int height);
            ~Window();
            void clear() const;
            void update();
            bool closed() const;

            inline int getWidth() const
            {
                return m_Width;
            }
            inline int getHeight() const
            {
                return m_Height;
            }

            void setKey(unsigned int keyCode, bool val);
            void setMouseButton(unsigned int button, bool val);
            void setMousePosition(double x, double y);

            bool isKeyPressed(unsigned int keyCode) const;
            bool isMouseButtonPressed(unsigned int button) const;
            void getMousePosition(double& x, double& y) const;

        private:
            bool init();
            static void key_callback(GLFWwindow* window, int key, int scancode, int
action, int mods);
            static void mouse_button_callback(GLFWwindow* window, int button, int action,
int mods);
            static void cursor_position_callback(GLFWwindow* window, double xpos, double
ypos);

    };

} }
```

And finally:

Table 12 - Episode 3 window.cpp file

```
#include "window.h"

namespace sparky { namespace graphics {


    void window_resize(GLFWwindow *window, int width, int height);

    Window::Window(const char *title, int width, int height)
    {
        m_Title = title;
        m_Width = width;
        m_Height = height;
        if (!init())
        {
            glfwTerminate();
        }
        for (int i=0; i < MAX_KEYS; i++)
        {
            m_Keys[i] = false;
        }
        for (int i=0; i < MAX_BUTTONS; i++)
        {
            m_MouseButtons[i] = false;
        }
   }

    Window::~Window()
    {
        glfwTerminate();
    }

    bool Window::init()
    {
        /* Initialize the library */
        if (!glfwInit())
        {
            cerr << "Error Initializing GLFW library." << endl;
            return false;
        }

        m_Window = glfwCreateWindow(m_Width, m_Height, m_Title, NULL, NULL);

        if (!m_Window)
        {
            glfwTerminate();
            cerr << "Failed to create GLFW window!" << endl;
            return false;
        }

        glfwMakeContextCurrent(m_Window);
        glfwSetWindowUserPointer(m_Window, this);

        // Setting up our callback functions
```

```
        glfwSetWindowSizeCallback(m_Window, window_resize);
        glfwSetKeyCallback(m_Window, key_callback);
        glfwSetMouseButtonCallback(m_Window, mouse_button_callback);
        glfwSetCursorPosCallback(m_Window, cursor_position_callback);

        if (glewInit() != GLEW_OK)
        {
            cerr << "Could not initialize GLEW!" << endl;
            return false;
        }
        return true;
    }

    bool Window::closed() const
    {
        return glfwWindowShouldClose(m_Window) == 1;
    }
    void Window::clear() const
    {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    }
    void Window::update()
    {
        glfwPollEvents();
//        glfwGetFramebufferSize(m_Window, &m_Width, &m_Height);
        glfwSwapBuffers(m_Window);
    }

    void window_resize(GLFWwindow *window, int width, int height)
    {
         glViewport(0, 0, width, height);
    }

    void Window::setKey(unsigned int keyCode, bool val)
    {
       if (keyCode >= 0 && keyCode < MAX_KEYS)
       {
             m_Keys[keyCode] = val;
       }
    }

    bool Window::isKeyPressed(unsigned int keyCode) const
    {
       if (keyCode >= 0 && keyCode < MAX_KEYS)
       {
             return m_Keys[keyCode];
       }
       // TODO: Log this!
       return false;
    }

    void Window::key_callback(GLFWwindow* window, int key, int scancode, int action, int
mods)
    {
        Window* win = (Window*) glfwGetWindowUserPointer(window);
        win->setKey(key, action != GLFW_RELEASE);
    }
```

```
    void Window::mouse_button_callback(GLFWwindow* window, int button, int action, int
mods)
    {
       Window* win = (Window*) glfwGetWindowUserPointer(window);
       win->setMouseButton(button, action != GLFW_RELEASE);
    }

    void Window::setMousePosition(double x, double y)
    {
       mx = x;
       my = y;
    }

    void Window::getMousePosition(double& x, double& y) const
    {
       x = mx;
       y = my;
    }

    void Window::cursor_position_callback(GLFWwindow* window, double xpos, double ypos)
    {
        Window* win = (Window*) glfwGetWindowUserPointer(window);
        win->setMousePosition(xpos, ypos);
    }


    bool Window::isMouseButtonPressed(unsigned int button) const
    {
        if (button >=0 && button < MAX_BUTTONS)
        {
            return m_MouseButtons[button];
        }
        // TODO: Log this!
        return false;
    }



    void Window::setMouseButton(unsigned int button, bool val)
    {
        if (button >= 0 && button < MAX_BUTTONS)
        {
            m_MouseButtons[button] = val;
        }
    }


} }
```

Web Site References

General Sites

1. Figueroa, Lorraine. (2015) BRAINYCODE – www.brainycode.com. Website on software development projects and interests.
2. TheCherno. (2012-2015). https://www.youtube.com/channel/UCQ-W1KE9EYfdxhL6S4twUNw. Website on Learning Programming and Game Development.

Learning OpenGL

1. Greg, Sidelnikov. (2015) www.falloutsoftware.com/. OpenGL Tutorials.
2. OGLDev, Modern OpenGL Tutorials. http://ogldev.atspace.co.uk/.

OpenGL Libraries

1. GLUT - https://www.opengl.org/resources/libraries/glut/
2. GLFW - http://www.glfw.org/
3. GLEW –
4. freeGLUT -

References and Books

Appendix A – Using GitHub

GitHub (https://github.com/) is a popular and well-known website that developers use to host their projects. It supports individual and group efforts to manage version control for documents and software development projects.

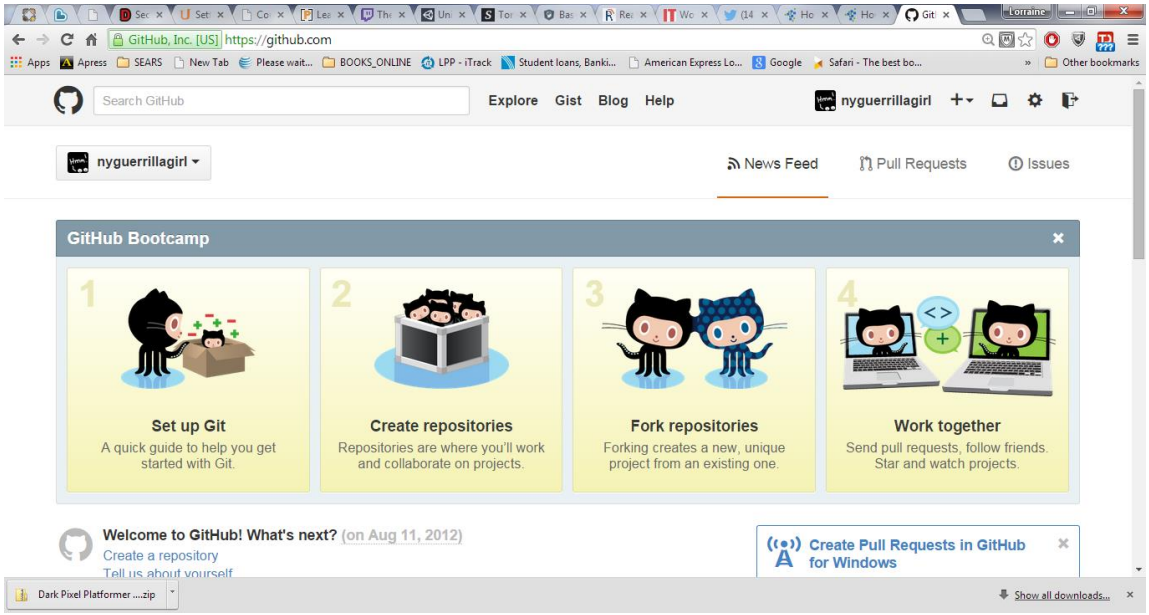


Figure 54 - GitHub Home Page

If you are a software developer still pondering the question "Why use version control?" then you either have very little experience or have yet to build software with a team of developers. If you ever had to go back and figure out "what changed about this file?" and are eyeballing it and going line-by-line looking for changes you really have to find better ways to waste your time. Version control software such as Git and a repository manager such as GitHub make these tasks manageable and seamless.

History of Git

The version control software named Git was started by the same team that worked on the Linux kernel. The goals were to develop a tool that was fast, simple in design, supported the ability to create branches, was fully distributed and could handle large projects – Git was born.

If you are familiar with other version control systems (e.g. SVN) Git may take some getting used to.

History of GitHub

GitHub is a web-based Git repository. It is free for use but does have paid services for companies who want to use it for their software development projects.

You can create your own repository on GitHub and then clone that repository on your own desktop.

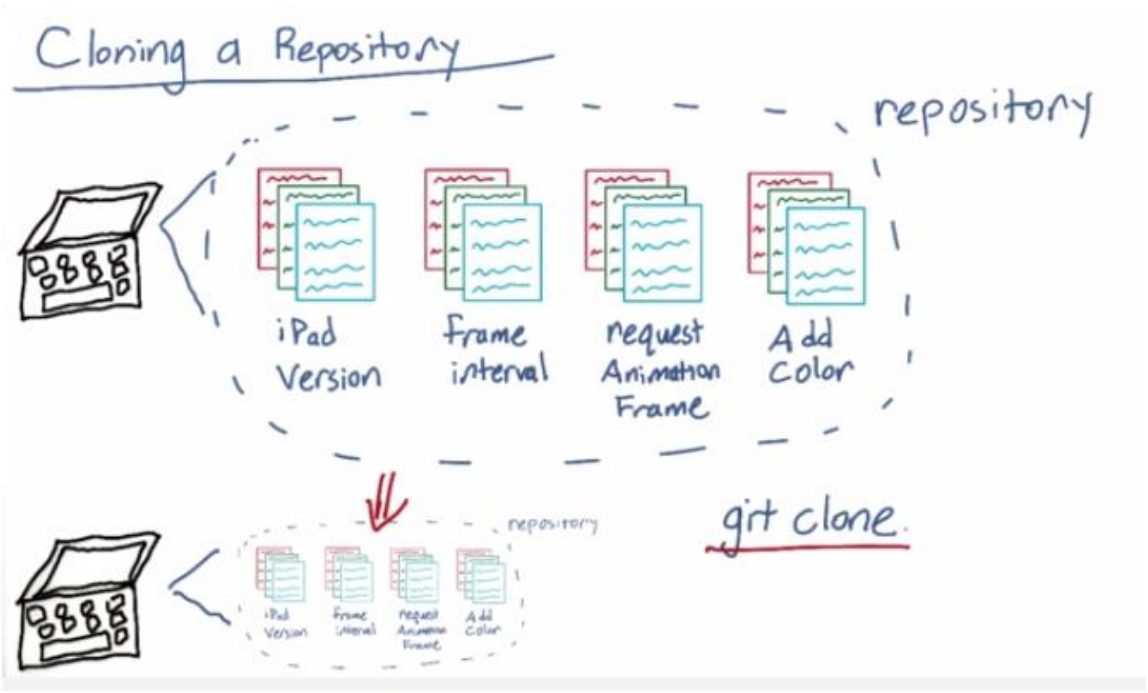


Figure 55 - Cloning a repository

If you find a repository project you like (e.g. TBD)

One typically works on a local/desktop copy of a project (usually a branch) and then push or integrates the changes into their GitHub repository.

Installing Git Bash

Obtain GitHub Account

Fork or Clone a Repository

Where to find good Tutorials

The best places on the Internet I have found to learn more about Git and GitHub are:

- http://git-scm.com – the host a great book on this website that you should spend a couple of hours reading.

How I typically work on a simple project

I created a repository on GitHub to host these notes and any code I develop as part of this project - https://github.com/nyguerrillagirl/Sparky-Engine.

1. I cloned the repository on my desktop (one time)
    a. Obtain HTTPS clone URL
    b. Start Git Bash on desktop
    c. Navigate to location of where you want to clone GitHub repository
    d. Issue “git clone <url>”
2. Whenever I update the software or document
    a. Update in git repository
    b. Example changes “git status”
    c. Add changes “git add <file>”
    d. Commit “git commit”
3. Upload changes to GitHub
    a. Issue “git push origin master”

Things get a bit more involved if I decided to branch off and test some new concepts (rather than adding to main branch).

Appendix B – Installing a C++ Development Environment

The C++ development system we use is Code::Blocks. It is free, extensible and fully configurable.

It runs on Linux, Mac and Windows (using wxWidgets). In addition, you can use your choice of compiler. The default is GCC (MingW/ GNU CSS).

MinGW is a contraction of “Minimalist GNU for Windows”, a minimalist development environment for native Microsoft Windows applications.

Appendix C – A Quick Introduction to OpenGL

What is OpenGL?

"OpenGL provides the programmer with an interface to graphics hardware. It is a powerful, low-level rendering and modeling software library, available on all major hardware platforms, with wide hardware support."

URL: http://en.wikipedia.org/wiki/OpenGL

OpenGL (Open Graphics Library) is a cross-language, multi-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU) to achieve hardware-accelerated rendering.

OpenGL was originally developed by the company Silicon Graphics, Inc. (SGI), running on their top-of-the-line workstations in 1991. OpenGL is now managed by the non-profit technology consortium Khronos Group.

OpenGL is used in many games going far back (e.g. the entire Quake series).

Figure 56 - Quake 3

OpenGL provides low-level rendering functions that give the programmer complete control. It is usually used with other libraries (e.g. GLUT, GLFW, etc) that manage other aspects of the application since OpenGL does not come with any APIs to create and manipulate windows.

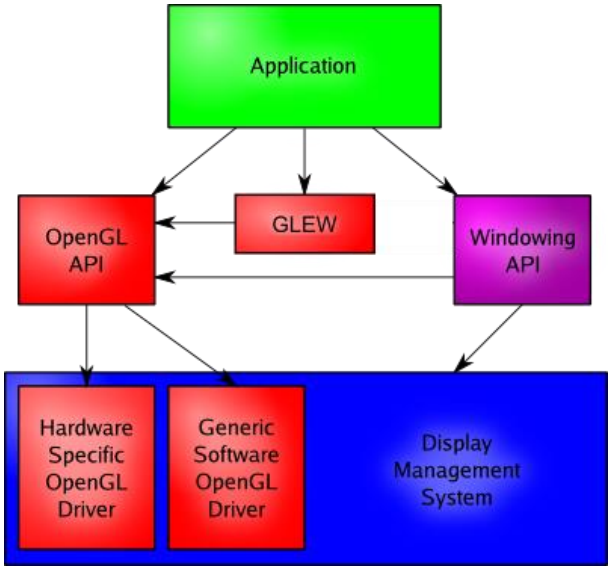


Figure 57 - OpenGL Architecture

The above image[10] depicts how your application uses the OpenGL API. You application will need to use a library such as GLUT or GLFW to get to the Windowing API (or use a platform specific API such as Win32). We have GLFW manage the specific details of the windowing system and only need to set up callback functions to our code to obtain events (e.g. user keypresses) and window updates (e.g. window resizes). GLEW provides access to OpenGL extension (across various platforms) without our code having to remember the various platform specific function names.

OpenGL comes with hundreds of functions that provide support for drawing 2D and 3D graphics.

The earliest versions of OpenGL were released with a companion library called GLU (the OpenGL Utility Library). GLU provided high-level functions and features that are unlikely to be available in modern hardware – such as mipmap generation, tessellation and generation of primitive shapes. The GLU specification was last updated in 1998, and the latest version depends on features which were deprecated with the release of OpenGL 3.1 in 2009[11]. Another library that may come up in your search for OpenGL facts is GLUT (OpenGL Utility Toolkit). It provided a set of support libraries that allowed the developer to create and manage windows, menus and input on various platforms. It is no longer maintained or recommended, developers should move to newer alternatives such as GLFW.

These notes will start with pre-shader OpenGL – that is, we will start with classic OpenGL and then segue into more modern OpenGL concept.

Prerequisites:

This section assumes you have:

[10] From http://www.cs.uregina.ca/Links/class-info/315/
[11] http://en.wikipedia.org/wiki/OpenGL

- Installed Code::Blocks
- Downloaded GLFW
- Downloaded GLEW

We will assume you have read up to Episode 2 covering how to set up a C++ application to compile and link to OpenGL, GLFW and GLEW.

In order to make all our development easier we will create our own OpenGL project template that will contain links to GLFW and GLEW that all our projects can use.

1. Create a Code::Blocks an emptykind project named OurOpenGLTemplate
2. Navigate to the Management → Files tab; Find the project file "OurOpenGLTemplate"
3. Create a new folder named "dependencies"
4. Create the folders GLFW and GLEW under dependencies
5. For GLFW download the windows version of GLFW and copy the include (mingw) files to dependencies\include and the
6. Create a lib folder under dependencies\GLFW and copy glfw3.dll, glfw3dll.a and libglfw3.a files there.
7. Also copy glfw3.dll to your MinGW\bin directory so that your applications run within the Code::Blocks GUI.
8. For GLEW download the Windows version
9. Copy the include directories to dependencies\GLEW
10. Create a lib directory under dependencies\GLEW and copy the files lib\Release\Win32\glew32 and bin\Release\Win32\glew32.dll.
11. Copy the glew32.dll to the MingGW\bin directory

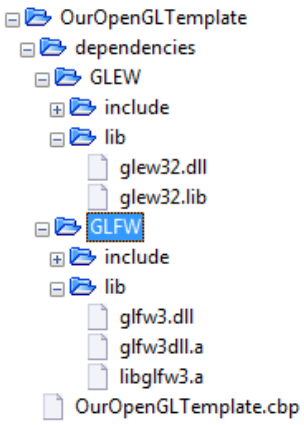


Figure 58 - Our OpenGL template files

12. Return back to Management → Projects tab
13. File → New → Empty File

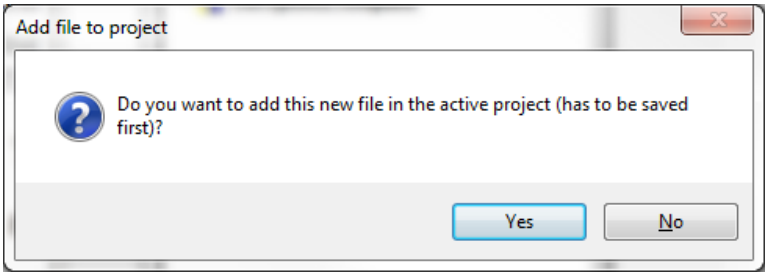


Figure 59- Add file to project dialog message

14. Click “Yes”

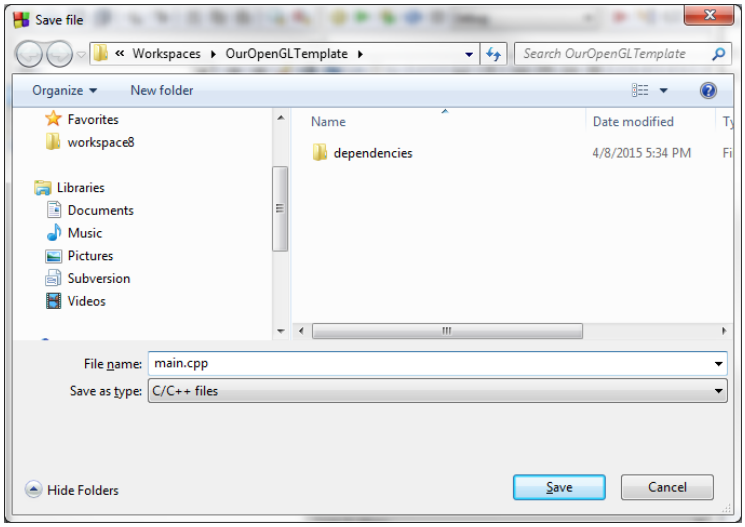


Figure 60 - Adding our main.cpp file

15. Enter the name main.cpp and click on “Save”

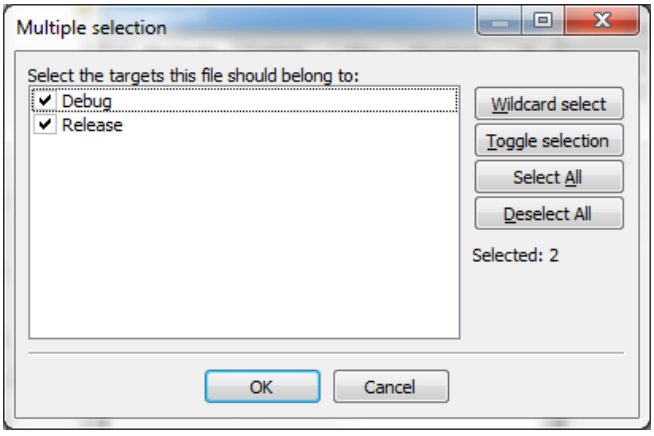


Figure 61 - Target selection

16. Take the configuration target defaults and click on “OK”

A new file main.cpp will be added to your project. Add the following code:

Table 13 - Simple "Hello World" OpenGL project

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>

using namespace std;

/* Function Prototypes */
void windowResize(GLFWwindow *window, int width, int height);
void drawSquare();

int main(void)
{
    GLFWwindow* window;

    /* Initialize the library */
    if (!glfwInit())
        return -1;

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        return -1;
    }

    /* Make the window's context current */
    glfwMakeContextCurrent(window);
    windowResize(window, 640, 480);
    /* Establish windows resize callback function */
    glfwSetWindowSizeCallback(window, windowResize);

    /* Initialize GLEW library */
    if (glewInit() != GLEW_OK)
    {
        cerr << "Could not initialize GLEW!" << endl;
        return -2;
    }
    /* Loop until the user closes the window */
    while (!glfwWindowShouldClose(window))
    {
        /* Render here */
        drawSquare();

        /* Swap front and back buffers */
        glfwSwapBuffers(window);

        /* Poll for and process events */
        glfwPollEvents();
    }

    glfwTerminate();
    return 0;
}
void drawSquare()
{
```

```
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_POLYGON);
        glVertex3f(20, 20.0, 0.0);
        glVertex3f(80.0, 20.0, 0.0);
        glVertex3f(80.0,80.0, 0.0);
        glVertex3f(20.0,80.0, 0.0);
    glEnd();

    glFlush();
}

void windowResize(GLFWwindow *window, int width, int height)
{
    cout << "width: " << width << "\theight: " << height << endl;
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 100.0, 0.0, 100.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glClearColor(0.0, 1.0, 1.0, 0.0);
}
```

If we try to build now we will see many error messages because we have not updated the search directories and linker settings.

17. Project → Build Options… ; select the Search directories tab

Add the entries:

$(PROJECT_DIRECTORY)dependencies\GLEW\include
$(PROJECT_DIRECTORY)dependencies\GLFW\include

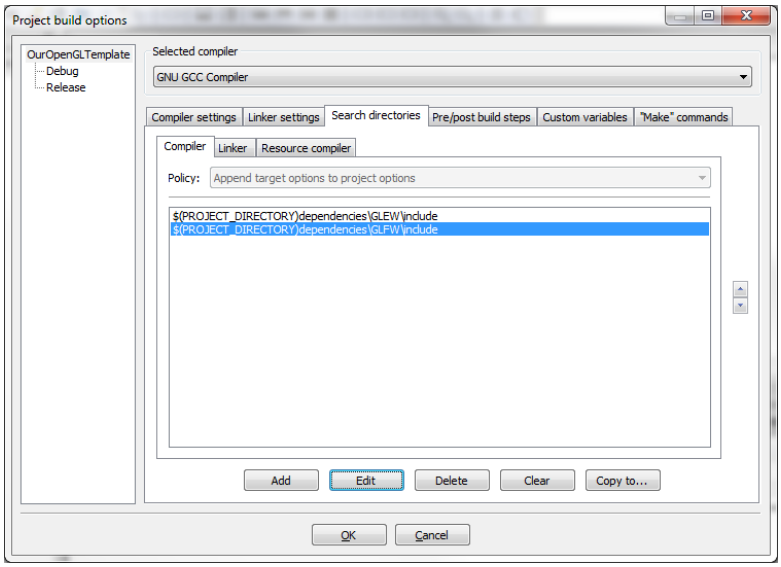


Figure 62 – Adding the GLEW and GLFW search directories

18. Project → Build Options… and click on the Linker Settings tab.
19. Add the following libraries:
    a. opengl32
    b. $(PROJECT_DIRECTORY)dependencies\GLEW\lib\glew32.lib
    c. $(PROJECT_DIRECTORY)dependencies\GLEW\lib\glew32.dll
    d. $(PROJECT_DIRECTORY)dependencies\GLFW\lib\glfw3dll.a
    e. $(PROJECT_DIRECTORY)dependencies\GLFW\lib\libglfw3.a

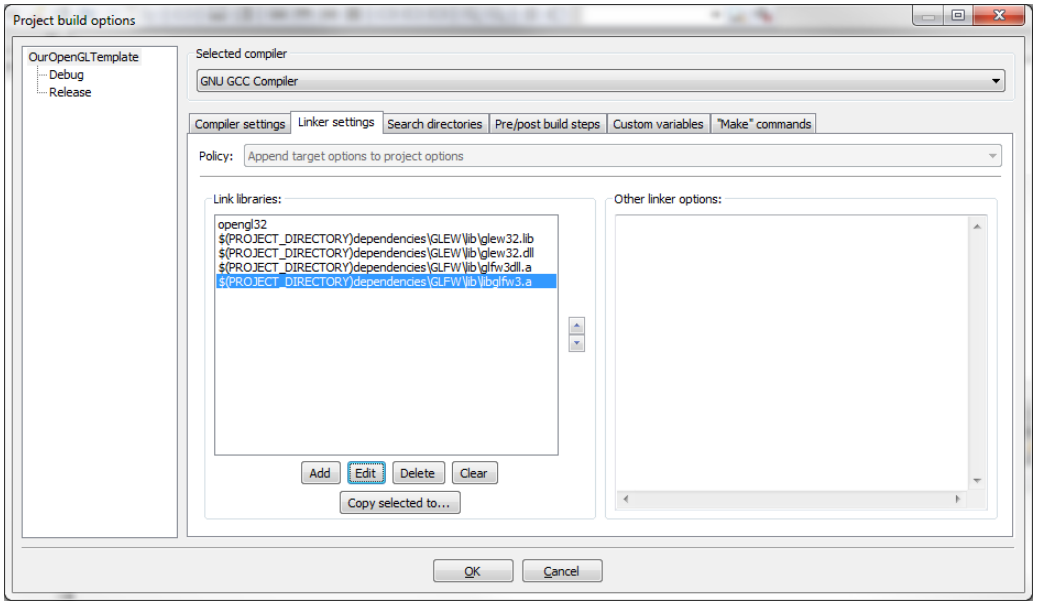


Figure 63 - Link libraries

20. Compile/Build and Run the program.

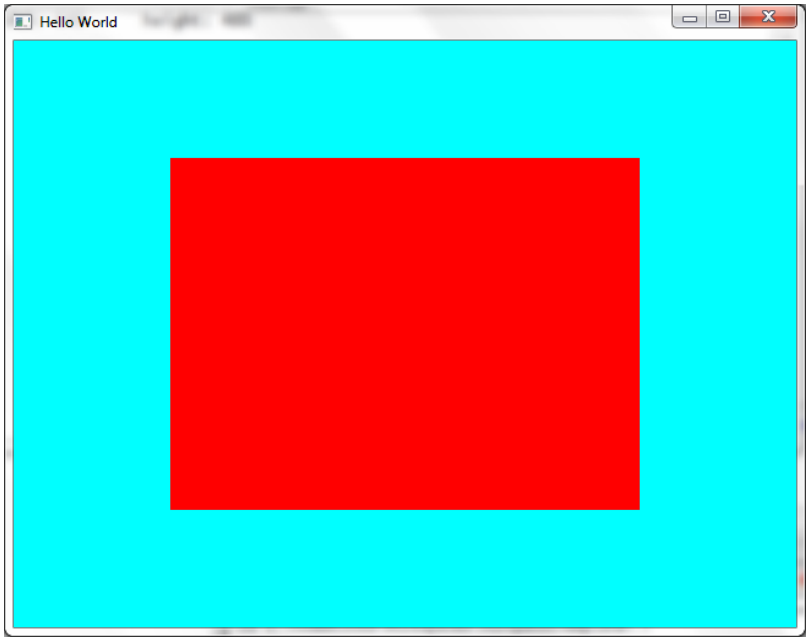


Figure 64 - Hello World OpenGL/GLFW/GLEW Program

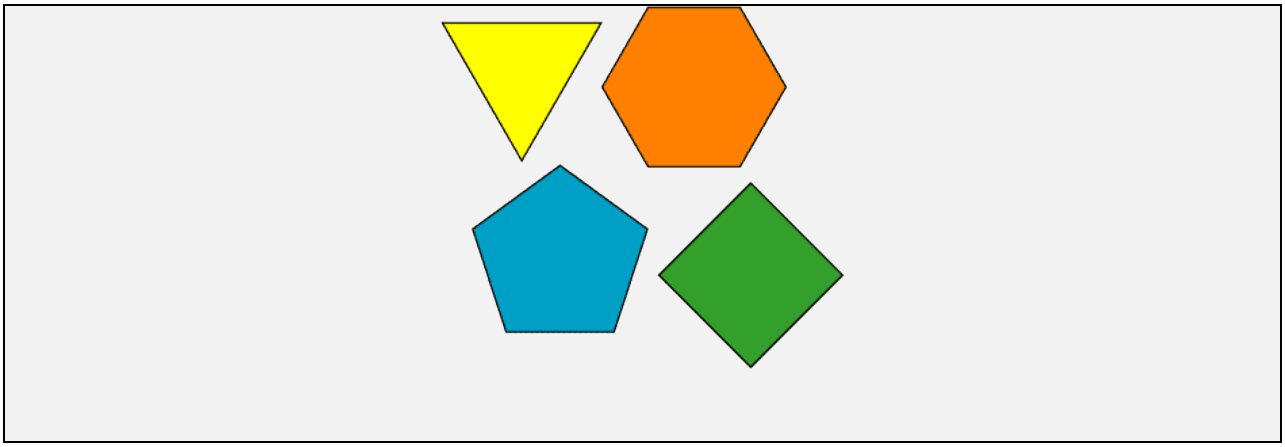
The program prints a red square on a light sky blue background. The code OpenGL code that draws the red square on the screen:

```
    glBegin(GL_POLYGON);
        glVertex3f(20, 20.0, 0.0);
        glVertex3f(80.0, 20.0, 0.0);
        glVertex3f(80.0,80.0, 0.0);
        glVertex3f(20.0,80.0, 0.0);
    glEnd();
```

The above uses an older but still valid OpenGL way of drawing that you will see in documentation and examples of how to code in OpenGL. We will discuss in these notes more modern techniques but it pays to learn older techniques if only to appreciate how OpenGL evolved. The first command is `glBegin()` that is always accompanied by an ending `glEnd()` and a set of vertices in between. `glBegin()` is provided with one parameter describing the object primitive the vertices describe. In using the `GL_POLYGON` we are stating that our vertices are the endpoints of a polygon.

URL: http://en.wikipedia.org/wiki/Polygon

In geometry, a polygon is traditionally a plane figure that is bounded by a finite chain of straight line segments closing in a loop to form a closed chain or circuit. These segments are called its edges or sides, and the points where two edges meet are the polygon's vertices (singular: vertex) or corners. The interior of the polygon is sometimes called its body.



The object primitives are just basic shapes that can easily be drawn. The options are a single point, polygon, or triangle. The use of glBegin() is referred to as *immediate mode*. It pays to mention again that this style of drawing is deprecated and no longer recommended.

glBegin — delimit the vertices of a primitive or a group of like primitives

C Specification

void **glBegin**(GLenum mode);

Parameters
mode
Specifies the primitive or primitives that will be created from vertices presented between glBegin and the subsequent glEnd. Ten symbolic constants are accepted: GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP, and GL_POLYGON.

C Specification

void glEnd(void);
Description

glBegin and glEnd delimit the vertices that define a primitive or a group of like primitives. glBegin accepts a single argument that specifies in which of 10 ways the vertices are interpreted. Taking n as an integer count starting at one, and N as the total number of vertices specified, the interpretations are as follows:

GL_POINTS
Treats each vertex as a single point. Vertex n defines point n. N points are drawn.

GL_LINES
Treats each pair of vertices as an independent line segment. Vertices 2 ☒ n - 1 and 2 ☒ n define line n. N 2 lines are drawn.

GL_LINE_STRIP
Draws a connected group of line segments from the first vertex to the last. Vertices n and n + 1 define line n. N - 1 lines are drawn.

GL_LINE_LOOP
Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices n and n + 1 define line n. The last line, however, is defined by vertices N and 1 . N lines are drawn.

GL_TRIANGLES
Treats each triplet of vertices as an independent triangle. Vertices 3 ☒ n - 2 , 3 ☒ n - 1 , and 3 ☒ n define triangle n. N 3 triangles are drawn.

GL_TRIANGLE_STRIP
Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. For odd n, vertices n, n + 1 , and n + 2 define triangle n. For even n, vertices n + 1 , n, and n + 2 define triangle n. N - 2 triangles are drawn.

GL_TRIANGLE_FAN
Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. Vertices 1 , n + 1 , and n + 2 define triangle n. N - 2 triangles are drawn.

GL_QUADS
Treats each group of four vertices as an independent quadrilateral. Vertices 4 ☒ n - 3 , 4 ☒ n - 2 , 4 ☒ n - 1 , and 4 ☒ n define quadrilateral n. N 4 quadrilaterals are drawn.

GL_QUAD_STRIP
Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices 2 ☒ n - 1 , 2 ☒ n , 2 ☒ n + 2 , and 2 ☒ n + 1 define quadrilateral n. N 2 - 1 quadrilaterals are drawn. Note that the order in which vertices are used to construct a quadrilateral from strip data is different from that used with independent data.

GL_POLYGON
Draws a single, convex polygon. Vertices 1 through N define this polygon.

Only a subset of GL commands can be used between glBegin and glEnd. The commands are glVertex, glColor, glSecondaryColor, glIndex, glNormal, glFogCoord, glTexCoord, glMultiTexCoord, glVertexAttrib, glEvalCoord, glEvalPoint, glArrayElement, glMaterial, and glEdgeFlag. Also, it is acceptable to use glCallList or glCallLists to execute display lists that include only the preceding commands. If any other GL command is executed between glBegin and glEnd, the error flag is set and the command is ignored.

Regardless of the value chosen for mode, there is no limit to the number of vertices that can be defined between glBegin and glEnd. Lines, triangles, quadrilaterals, and polygons that are incompletely specified are not drawn. Incomplete specification results when either too few vertices are provided to specify even a single primitive or when an incorrect multiple of vertices is specified. The incomplete primitive is ignored; the rest are drawn.

The minimum specification of vertices for each primitive is as follows: 1 for a point, 2 for a line, 3 for a triangle, 4 for a quadrilateral, and 3 for a polygon. Modes that require a certain multiple of vertices are GL_LINES (2), GL_TRIANGLES (3), GL_QUADS (4), and GL_QUAD_STRIP (2).

Errors

GL_INVALID_ENUM is generated if mode is set to an unaccepted value.

GL_INVALID_OPERATION is generated if glBegin is executed between a glBegin and the corresponding execution of glEnd.

GL_INVALID_OPERATION is generated if glEnd is executed without being preceded by a glBegin.

GL_INVALID_OPERATION is generated if a command other than glVertex, glColor, glSecondaryColor, glIndex, glNormal, glFogCoord, glTexCoord, glMultiTexCoord, glVertexAttrib, glEvalCoord, glEvalPoint, glArrayElement, glMaterial, glEdgeFlag, glCallList, or glCallLists is executed between the execution of glBegin and the corresponding execution glEnd.

Execution of glEnableClientState, glDisableClientState, glEdgeFlagPointer, glFogCoordPointer, glTexCoordPointer, glColorPointer, glSecondaryColorPointer, glIndexPointer, glNormalPointer, glVertexPointer, glVertexAttribPointer, glInterleavedArrays, or glPixelStore is not allowed after a call to glBegin and before the corresponding call to glEnd, but an error may or may not be generated.

See Also

glArrayElement, glCallList, glCallLists, glColor, glEdgeFlag, glEvalCoord, glEvalPoint, glFogCoord, glIndex, glMaterial, glMultiTexCoord, glNormal, glSecondaryColor, glTexCoord, glVertex, glVertexAttrib

The vertices of the polygon are specified using the `glVertex3f` function. There are many flavors of this function that depend on the number of coordinates and the datatype of the arguments being provided. The general format is:

glVertex<numberOfCoordinates><datatype>

Using the above we know that xxxx means three coordinates (therefore three parameters)

Your first question may be the coordinates.

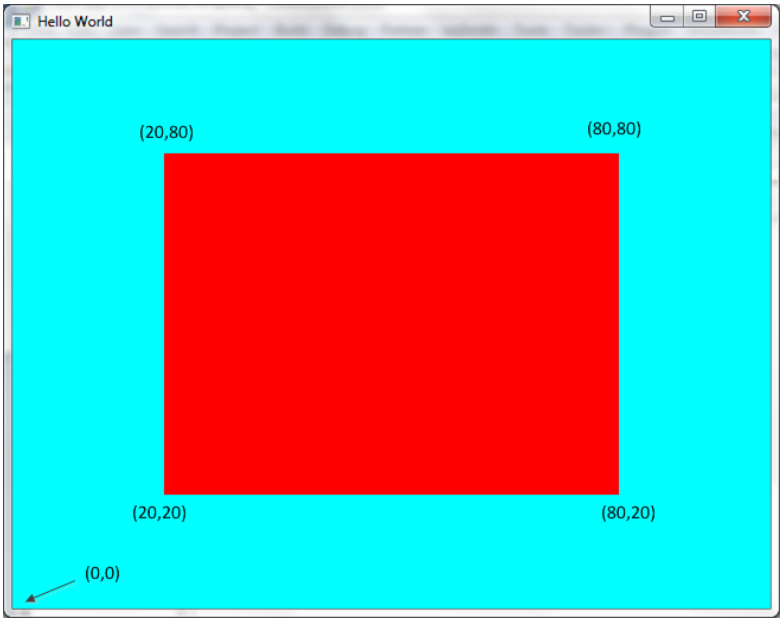


Figure 65 - The screen coordinates

The illustration below shows the uniform coordinate system assumed for an OpenGL frame vs how they may map to the actual device screen.

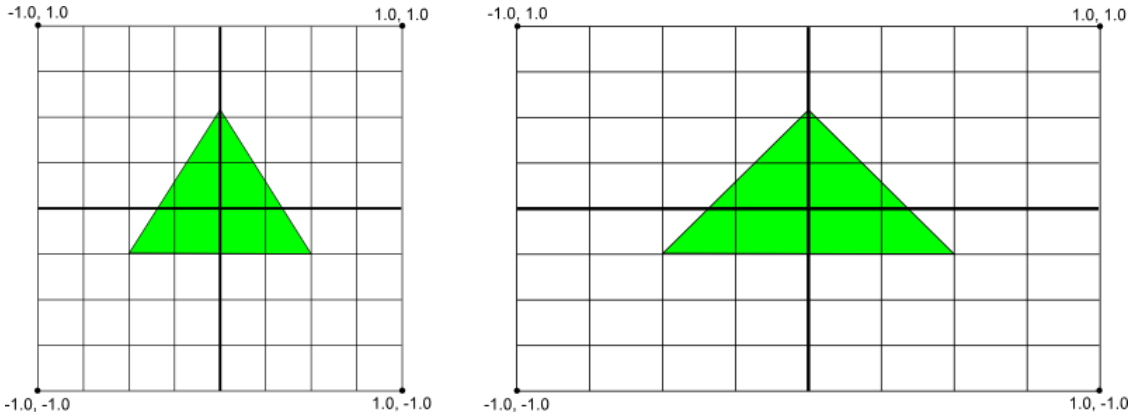


Figure 66 - Normal OpenGL coordinate system

We changed the coordinate system by changing the projection. In our program this statement is:

```
glOrtho(0.0, 100.0, 0.0, 100.0, -1.0, 1.0);
```

glOrtho — multiply the current matrix with an orthographic matrix

C Specification

void **glOrtho**(GLdouble left,

```
        GLdouble    right,
        GLdouble    bottom,
        GLdouble    top,
        GLdouble    nearVal,
        GLdouble    farVal);
```

Parameters

left, right

Specify the coordinates for the left and right vertical clipping planes.

bottom, top

Specify the coordinates for the bottom and top horizontal clipping planes.

nearVal, farVal

Specify the distances to the nearer and farther depth clipping planes. These values are negative if the plane is to be behind the viewer.

glOrtho sets up a viewing box.

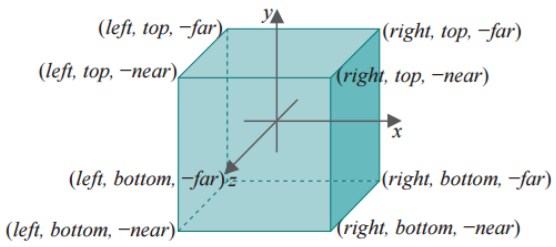


Figure 2.5: Viewing box of `glOrtho(left, right, bottom, top, near, far)`.

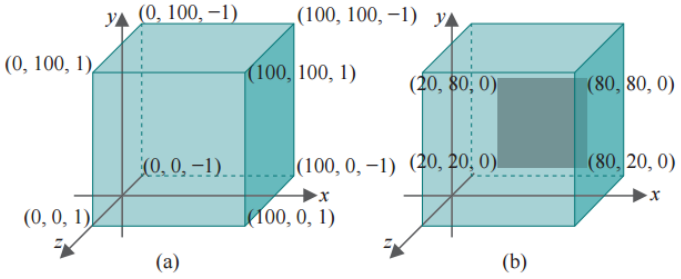


Figure 67 - From http://www.sumantaguha.com/files/materials/ch2.pdf

Programming Exercises

1. Create a new project named RANDOM_DOTS that uses glBegin(GL_POINTS) and prints out 1000 randomly colored dots on a 640 x480 screen. (Note: the default point size is 1 pixel, if your points are difficult to see you can change the size by using the glPointSize(2.0f) to change it to 2 pixels.

2. xx

Appendix D – GLFW Keyboard Macros

```
#define      GLFW_KEY_UNKNOWN   -1
#define      GLFW_KEY_SPACE   32
#define      GLFW_KEY_APOSTROPHE   39 /* ' */
#define      GLFW_KEY_COMMA   44 /* , */
#define      GLFW_KEY_MINUS   45 /* - */
#define      GLFW_KEY_PERIOD   46 /* . */
#define      GLFW_KEY_SLASH   47 /* / */
#define      GLFW_KEY_0   48
#define      GLFW_KEY_1   49
#define      GLFW_KEY_2   50
#define      GLFW_KEY_3   51
#define      GLFW_KEY_4   52
#define      GLFW_KEY_5   53
#define      GLFW_KEY_6   54
#define      GLFW_KEY_7   55
#define      GLFW_KEY_8   56
#define      GLFW_KEY_9   57
#define      GLFW_KEY_SEMICOLON   59 /* ; */
#define      GLFW_KEY_EQUAL   61 /* = */
#define      GLFW_KEY_A   65
#define      GLFW_KEY_B   66
#define      GLFW_KEY_C   67
#define      GLFW_KEY_D   68
#define      GLFW_KEY_E   69
#define      GLFW_KEY_F   70
#define      GLFW_KEY_G   71
#define      GLFW_KEY_H   72
#define      GLFW_KEY_I   73
#define      GLFW_KEY_J   74
#define      GLFW_KEY_K   75
#define      GLFW_KEY_L   76
#define      GLFW_KEY_M   77
#define      GLFW_KEY_N   78
#define      GLFW_KEY_O   79
#define      GLFW_KEY_P   80
#define      GLFW_KEY_Q   81
#define      GLFW_KEY_R   82
#define      GLFW_KEY_S   83
#define      GLFW_KEY_T   84
#define      GLFW_KEY_U   85
#define      GLFW_KEY_V   86
#define      GLFW_KEY_W   87
#define      GLFW_KEY_X   88
#define      GLFW_KEY_Y   89
```

```
#define      GLFW_KEY_Z   90
#define      GLFW_KEY_LEFT_BRACKET   91 /* [ */
#define      GLFW_KEY_BACKSLASH   92 /* \ */
#define      GLFW_KEY_RIGHT_BRACKET   93 /* ] */
#define      GLFW_KEY_GRAVE_ACCENT   96 /* ` */
#define      GLFW_KEY_WORLD_1   161 /* non-US #1 */
#define      GLFW_KEY_WORLD_2   162 /* non-US #2 */
#define      GLFW_KEY_ESCAPE   256
#define      GLFW_KEY_ENTER   257
#define      GLFW_KEY_TAB   258
#define      GLFW_KEY_BACKSPACE   259
#define      GLFW_KEY_INSERT   260
#define      GLFW_KEY_DELETE   261
#define      GLFW_KEY_RIGHT   262
#define      GLFW_KEY_LEFT   263
#define      GLFW_KEY_DOWN   264
#define      GLFW_KEY_UP   265
#define      GLFW_KEY_PAGE_UP   266
#define      GLFW_KEY_PAGE_DOWN   267
#define      GLFW_KEY_HOME   268
#define      GLFW_KEY_END   269
#define      GLFW_KEY_CAPS_LOCK   280
#define      GLFW_KEY_SCROLL_LOCK   281
#define      GLFW_KEY_NUM_LOCK   282
#define      GLFW_KEY_PRINT_SCREEN   283
#define      GLFW_KEY_PAUSE   284
#define      GLFW_KEY_F1   290
#define      GLFW_KEY_F2   291
#define      GLFW_KEY_F3   292
#define      GLFW_KEY_F4   293
#define      GLFW_KEY_F5   294
#define      GLFW_KEY_F6   295
#define      GLFW_KEY_F7   296
#define      GLFW_KEY_F8   297
#define      GLFW_KEY_F9   298
#define      GLFW_KEY_F10   299
#define      GLFW_KEY_F11   300
#define      GLFW_KEY_F12   301
#define      GLFW_KEY_F13   302
#define      GLFW_KEY_F14   303
#define      GLFW_KEY_F15   304
#define      GLFW_KEY_F16   305
#define      GLFW_KEY_F17   306
#define      GLFW_KEY_F18   307
#define      GLFW_KEY_F19   308
#define      GLFW_KEY_F20   309
#define      GLFW_KEY_F21   310
#define      GLFW_KEY_F22   311
```

```
#define      GLFW_KEY_F23   312
#define      GLFW_KEY_F24   313
#define      GLFW_KEY_F25   314
#define      GLFW_KEY_KP_0   320
#define      GLFW_KEY_KP_1   321
#define      GLFW_KEY_KP_2   322
#define      GLFW_KEY_KP_3   323
#define      GLFW_KEY_KP_4   324
#define      GLFW_KEY_KP_5   325
#define      GLFW_KEY_KP_6   326
#define      GLFW_KEY_KP_7   327
#define      GLFW_KEY_KP_8   328
#define      GLFW_KEY_KP_9   329
#define      GLFW_KEY_KP_DECIMAL   330
#define      GLFW_KEY_KP_DIVIDE   331
#define      GLFW_KEY_KP_MULTIPLY   332
#define      GLFW_KEY_KP_SUBTRACT   333
#define      GLFW_KEY_KP_ADD   334
#define      GLFW_KEY_KP_ENTER   335
#define      GLFW_KEY_KP_EQUAL   336
#define      GLFW_KEY_LEFT_SHIFT   340
#define      GLFW_KEY_LEFT_CONTROL   341
#define      GLFW_KEY_LEFT_ALT   342
#define      GLFW_KEY_LEFT_SUPER   343
#define      GLFW_KEY_RIGHT_SHIFT   344
#define      GLFW_KEY_RIGHT_CONTROL   345
#define      GLFW_KEY_RIGHT_ALT   346
#define      GLFW_KEY_RIGHT_SUPER   347
#define      GLFW_KEY_MENU   348
#define      GLFW_KEY_LAST   GLFW_KEY_MENU
```

Appendix E – GLFW Mouse Macros

Macros

```
#define GLFW_MOUSE_BUTTON_1 0

#define GLFW_MOUSE_BUTTON_2 1

#define GLFW_MOUSE_BUTTON_3 2

#define GLFW_MOUSE_BUTTON_4 3

#define GLFW_MOUSE_BUTTON_5 4

#define GLFW_MOUSE_BUTTON_6 5

#define GLFW_MOUSE_BUTTON_7 6

#define GLFW_MOUSE_BUTTON_8 7

#define GLFW_MOUSE_BUTTON_LAST GLFW_MOUSE_BUTTON_8

#define GLFW_MOUSE_BUTTON_LEFT GLFW_MOUSE_BUTTON_1

#define GLFW_MOUSE_BUTTON_RIGHT GLFW_MOUSE_BUTTON_2

#define GLFW_MOUSE_BUTTON_MIDDLE GLFW_MOUSE_BUTTON_3
```

Appendix F – GLFW Joystick Macros

```
Macros
#define      GLFW_JOYSTICK_1   0

#define      GLFW_JOYSTICK_2   1

#define      GLFW_JOYSTICK_3   2

#define      GLFW_JOYSTICK_4   3

#define      GLFW_JOYSTICK_5   4

#define      GLFW_JOYSTICK_6   5

#define      GLFW_JOYSTICK_7   6

#define      GLFW_JOYSTICK_8   7

#define      GLFW_JOYSTICK_9   8

#define      GLFW_JOYSTICK_10   9

#define      GLFW_JOYSTICK_11   10

#define      GLFW_JOYSTICK_12   11

#define      GLFW_JOYSTICK_13   12

#define      GLFW_JOYSTICK_14   13

#define      GLFW_JOYSTICK_15   14

#define      GLFW_JOYSTICK_16   15

#define      GLFW_JOYSTICK_LAST   GLFW_JOYSTICK_16
```