Date: 4/7/2015

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?

Ludum 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

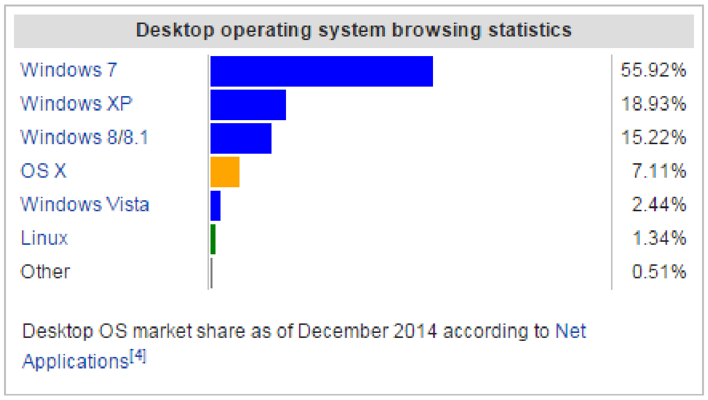
- it is easier to build a cross-platform game engine with it
- software development and debugging is faster
- has freely available tools and frameworks

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



Desktop operating system browsing statistics

Windows 7	55.92%
Windows XP	18.93%
Windows 8/8.1	15.22%
OS X	7.11%
Windows Vista	2.44%
Linux	1.34%
Other	0.51%

Desktop OS market share as of December 2014 according to Net Applications[4]

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.

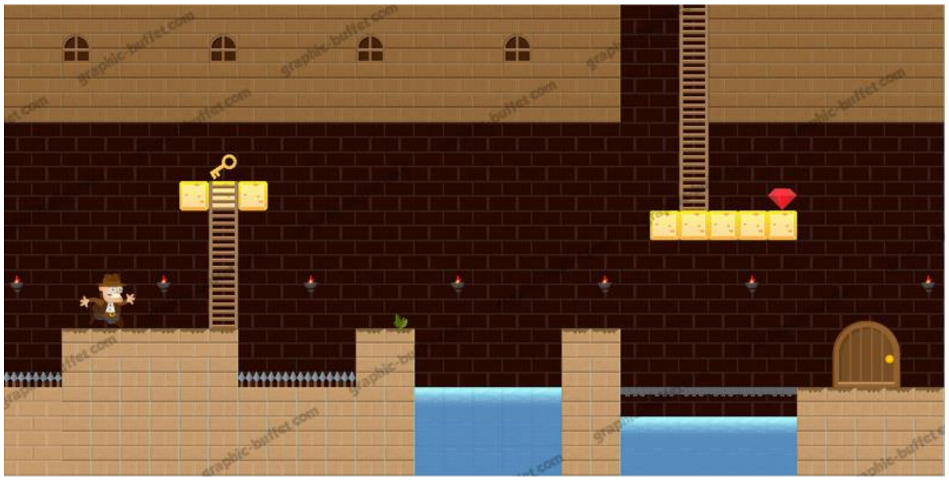


Figure 3 - 2D game look-and-feel

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.

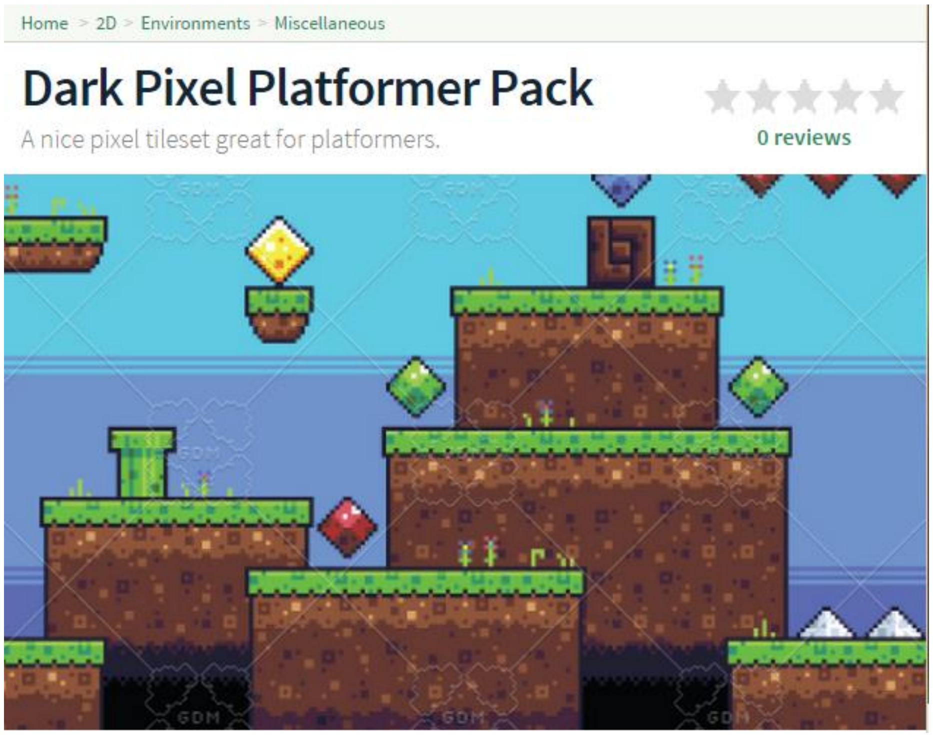

Home > 2D > Environments > Miscellaneous
Dark Pixel Platformer Pack
A nice pixel tileset great for platformers.
0 reviews
GDM

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.

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

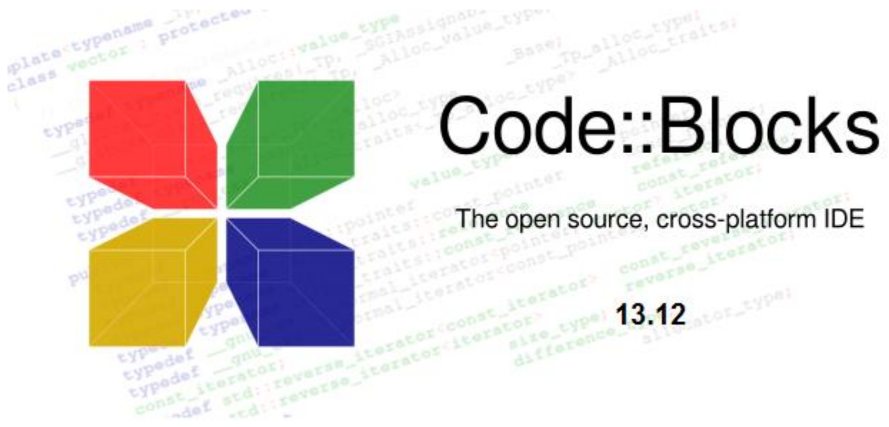
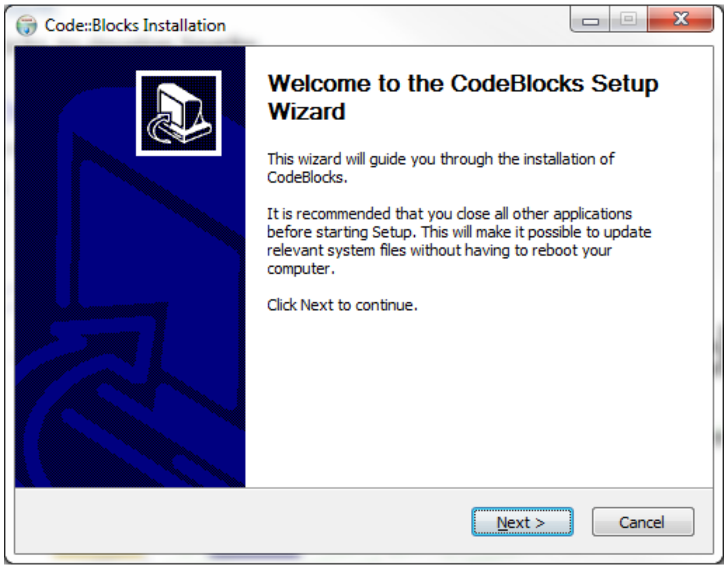


Figure 6 - Starting Code::Blocks Start Screen

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.


Code::Blocks Installation

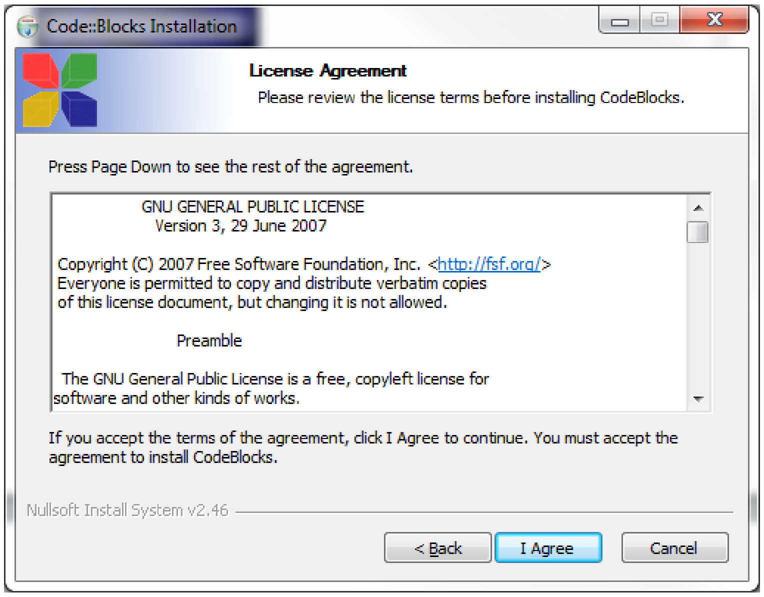
Welcome to the CodeBlocks Setup Wizard

This wizard will guide you through the installation of CodeBlocks.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

- Click Next >


Code::Blocks Installation

License Agreement

Please review the license terms before installing CodeBlocks.

Press Page Down to see the rest of the agreement.

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

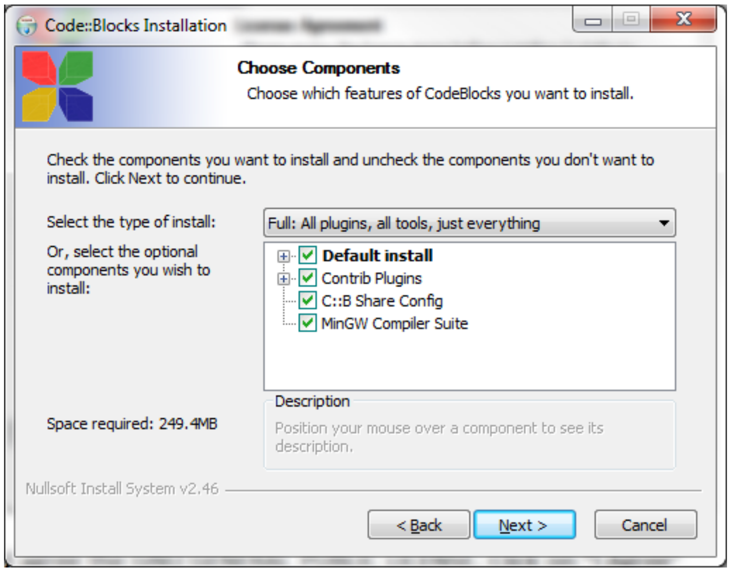
Preamble

The GNU General Public License is a free, copyleft license for
software and other kinds of works.

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install CodeBlocks.

Nullsoft Install System v2.46

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


Code::Blocks Installation

Choose Components

Choose which features of CodeBlocks you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select the type of install: Full: All plugins, all tools, just everything

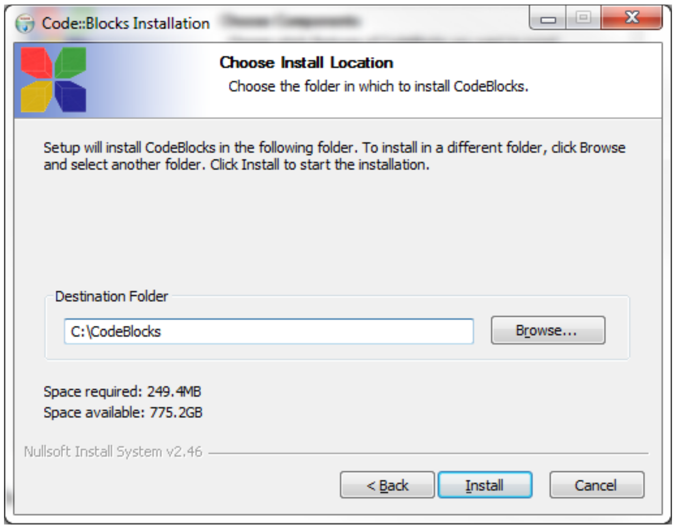
Or, select the optional components you wish to install:
- ✓ Default install
- ✓ Contrib Plugins
- ✓ C::B Share Config
- ✓ MinGW Compiler Suite

Description: Position your mouse over a component to see its description.

Space required: 249.4MB

Nullsoft Install System v2.46

- Accept the defaults and click on “Next > “


Code::Blocks Installation

Choose Install Location

Choose the folder in which to install CodeBlocks.

Setup will install CodeBlocks in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.

Destination Folder: C:\CodeBlocks

Space required: 249.4MB
Space available: 775.2GB

Nullsoft Install System v2.46

- 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.

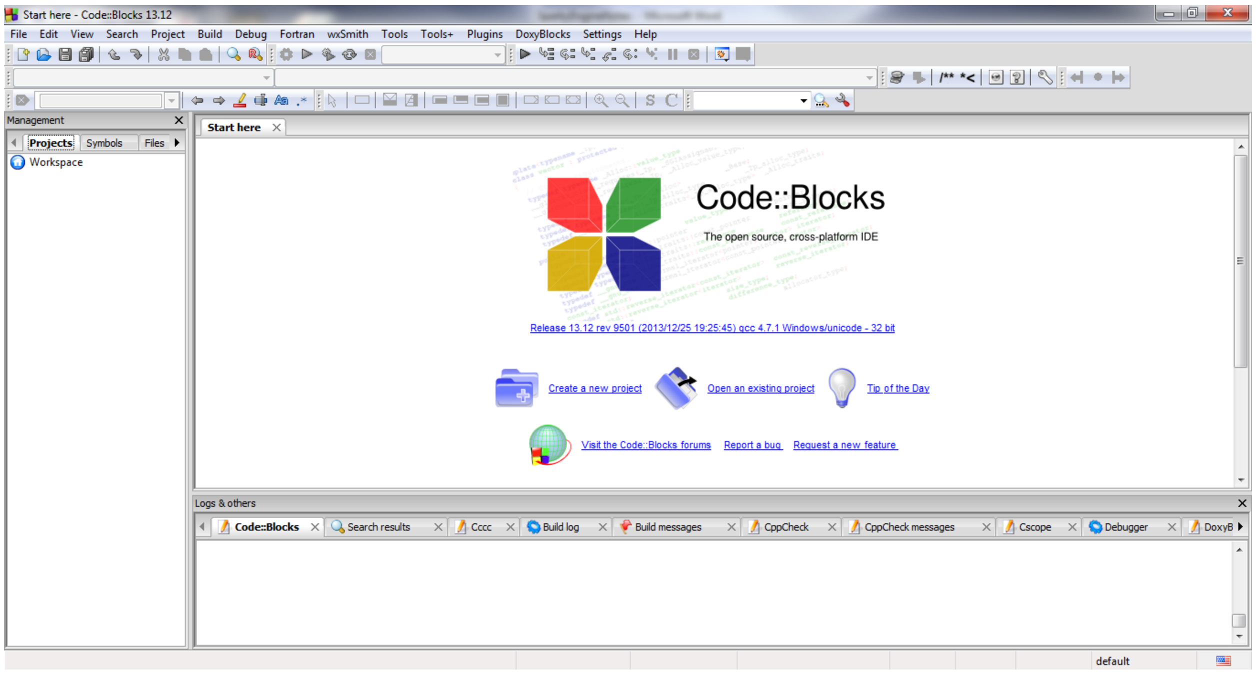

Start here - Code::Blocks 13.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Management
Projects Symbols Files
Workspace
Start here
Code::Blocks
The open source, cross-platform IDE
Release 13.12 rev 9501 (2013/12/25 19:25:45) gcc 4.7.1 Windows/unicode - 32 bit
Create a new project
Open an existing project
Tip of the Day
Visit the Code::Blocks forums
Report a bug
Request a new feature
Logs & others
Code::Blocks
Search results
Cccc
Build log
Build messages
CppCheck
CppCheck messages
Cscope
Debugger
DoxyB
default

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. The next section 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

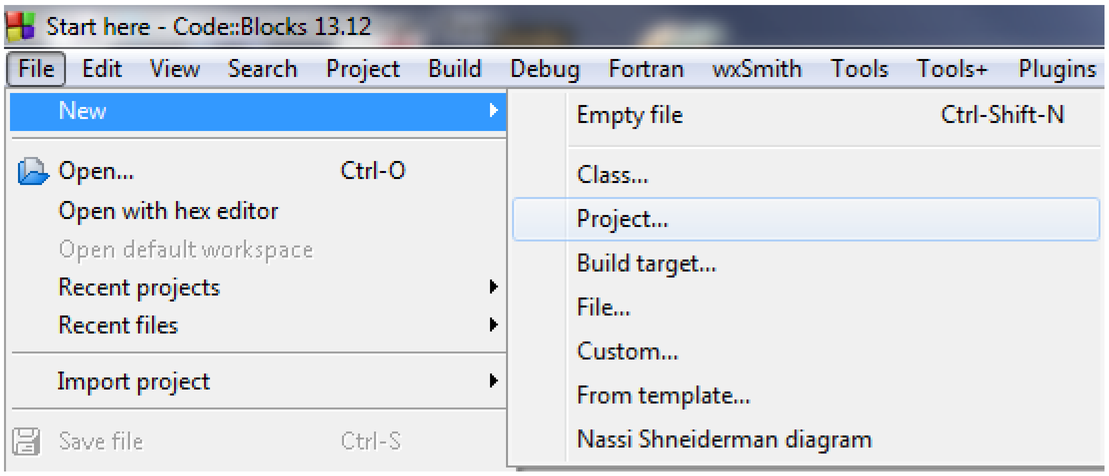
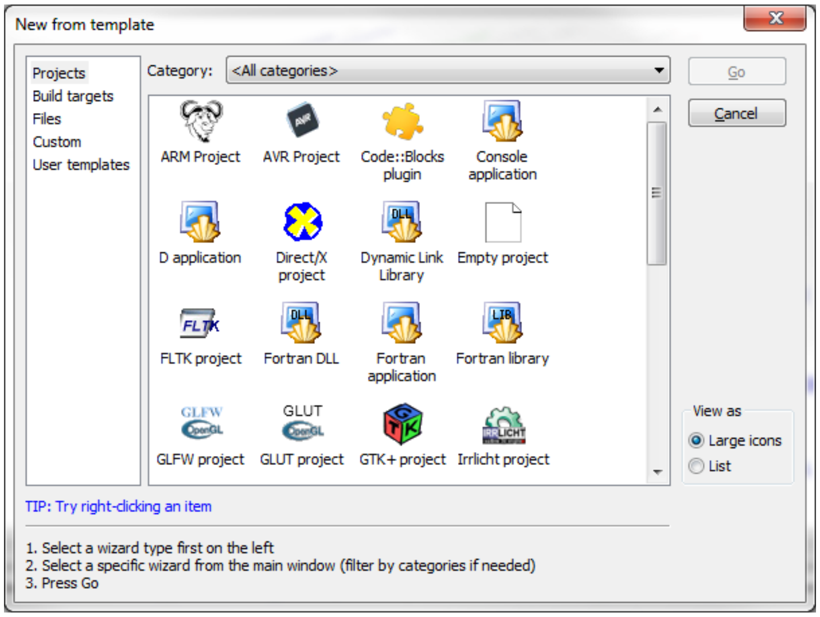


Figure 8 - Creating a new project

A "New from template" dialog box opens.



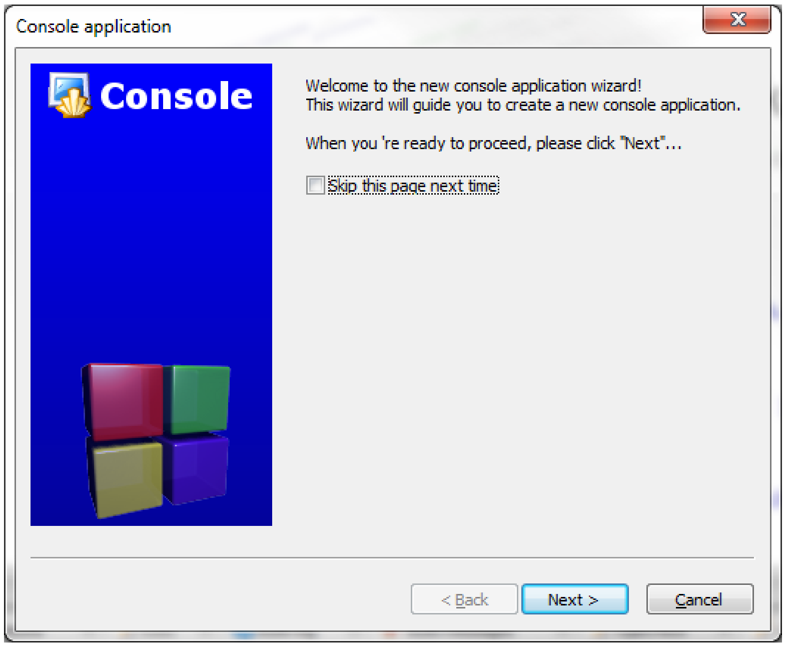
TIP: Try right-clicking an item

1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

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"



Welcome to the new console application wizard!
This wizard will guide you to create a new console application.

When you 're ready to proceed, please click "Next"...

Figure 10 - Console application wizard

What is a Console Application[7]?

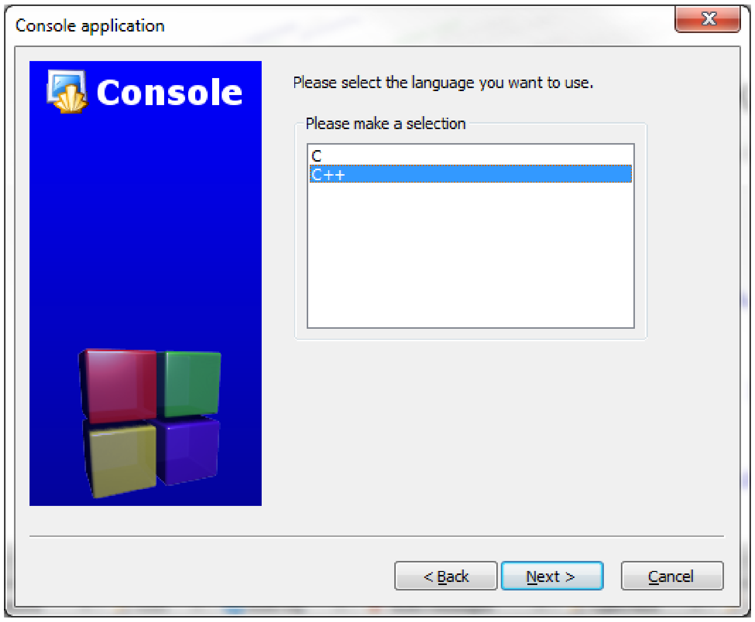
A console application is an application that takes input and displays output at a command line console with access to three basic data streams: standard input, standard output and standard error.

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).

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 >"

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



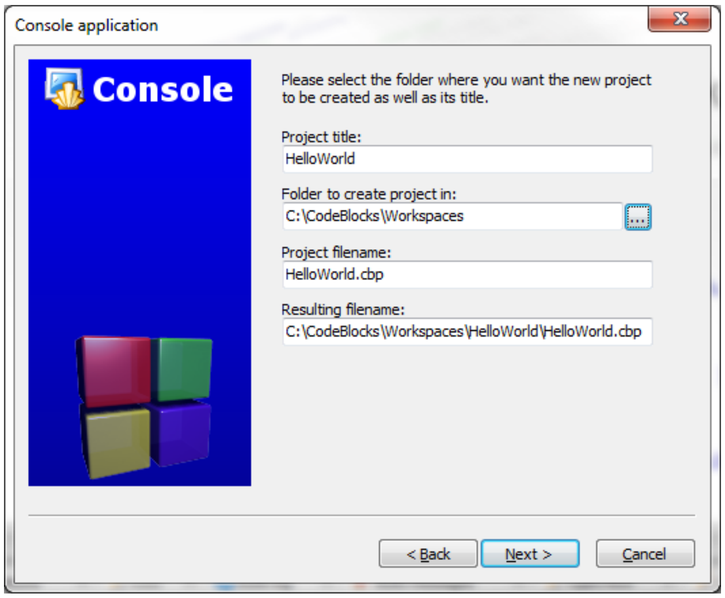
Please select the language you want to use.

Please make a selection

C
C++

Figure 11 - Console application - Language selection

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



Please select the folder where you want the new project to be created as well as its title.

Project title: HelloWorld

Folder to create project in: C:\CodeBlocks\Workspaces

Project filename: HelloWorld.cbp

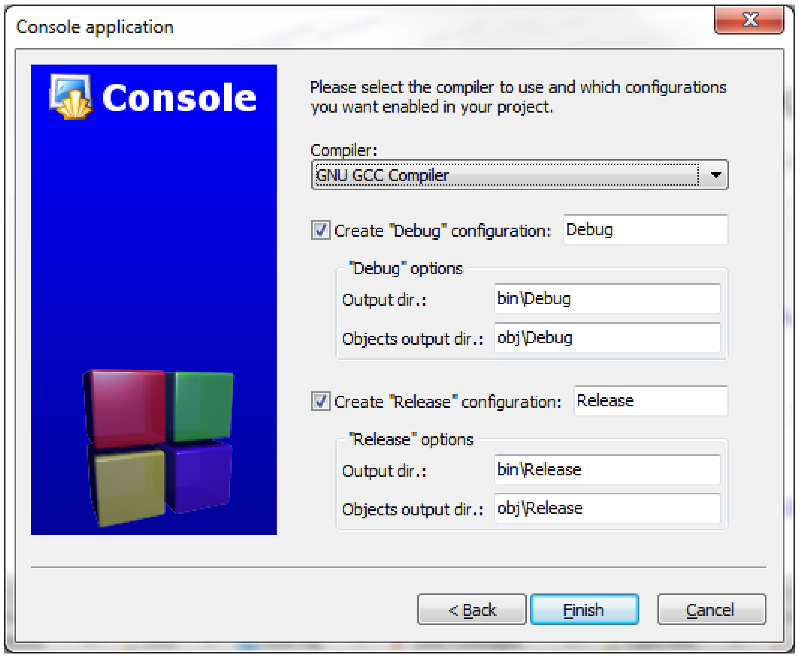
Resulting filename: C:\CodeBlocks\Workspaces\HelloWorld\HelloWorld.cbp

Figure 12 - 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 >”



Console application

Console

Please select the compiler to use and which configurations you want enabled in your project.

Compiler: GNU GCC Compiler

[✓] Create "Debug" configuration: Debug

"Debug" options

Output dir.: bin\Debug

Objects output dir.: obj\Debug

[✓] Create "Release" configuration: Release

"Release" options

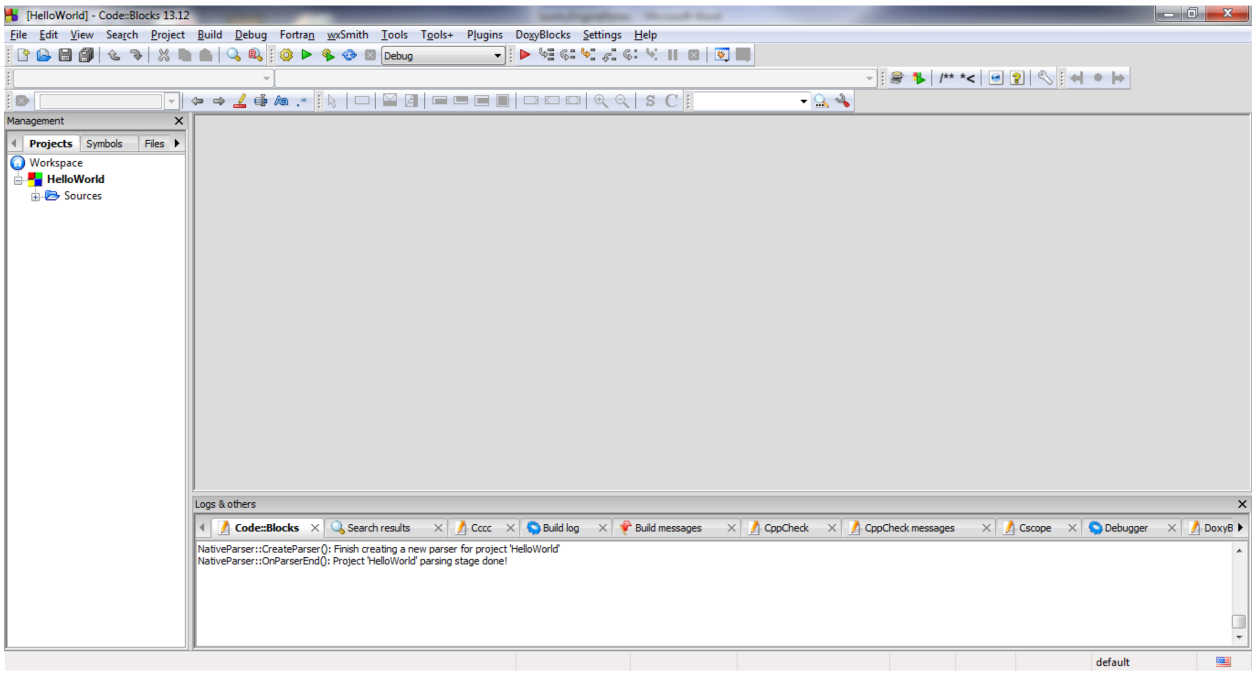
Output dir.: bin\Release

Objects output dir.: obj\Release

Figure 13 - Console application - configuration settings

Take the default "Debug" and "Release" configuration settings locations.

6. Click "Finish"

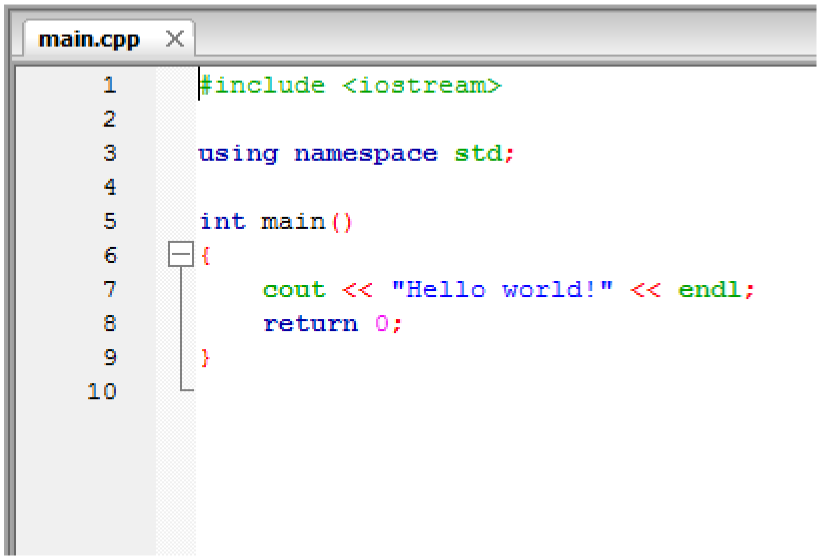


[HelloWorld] - Code::Blocks 13.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Debug
Management
Projects Symbols Files
Workspace
HelloWorld
Sources
Logs & others
Code::Blocks Search results Cccc Build log Build messages CppCheck CppCheck messages Cscope Debugger DoxyB
default

```
NativeParser::CreateParser(): Finish creating a new parser for project 'HelloWorld'
NativeParser::OnParserEnd(): Project 'HelloWorld' parsing stage done!
```

Figure 14 - 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.



```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}

```

Figure 15 - Our “Hello world!” main.cpp

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

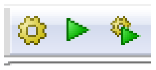
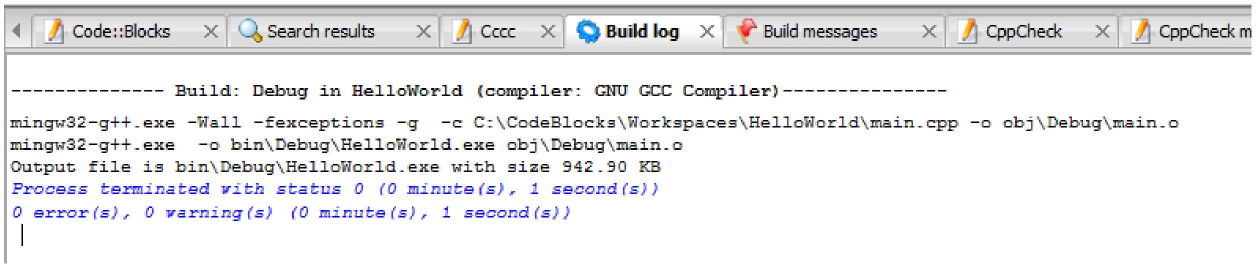


Figure 16 - Build, Run, Build and Run

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

9. Click on “Build” icon.



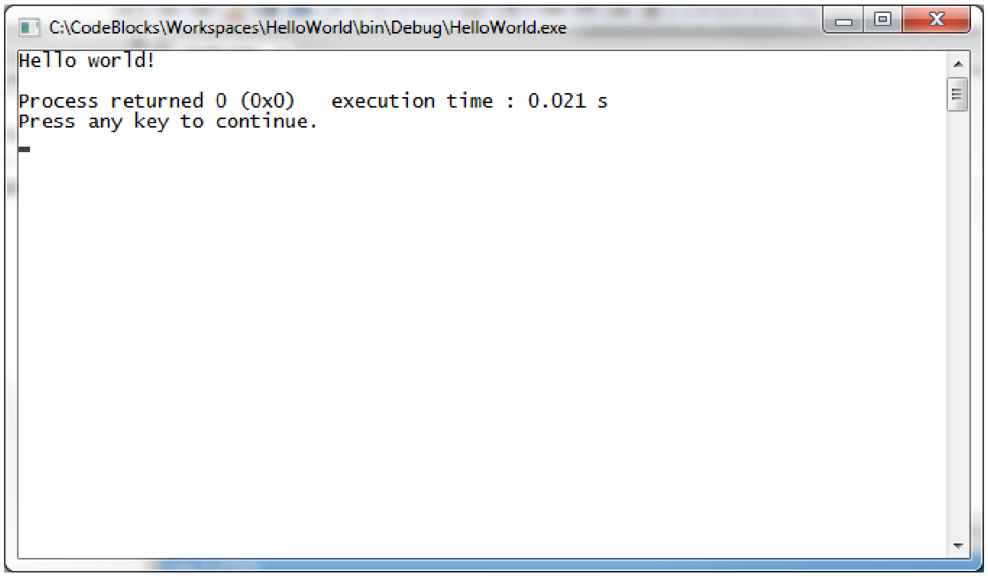
```
-------------- Build: Debug in HelloWorld (compiler: GNU GCC Compiler)---------------
mingw32-g++.exe -Wall -fexceptions -g  -c C:\CodeBlocks\Workspaces\HelloWorld\main.cpp -o obj\Debug\main.o
mingw32-g++.exe  -o bin\Debug\HelloWorld.exe obj\Debug\main.o
Output file is bin\Debug\HelloWorld.exe with size 942.90 KB
Process terminated with status 0 (0 minute(s), 1 second(s))
0 error(s), 0 warning(s) (0 minute(s), 1 second(s))
```

Figure 17 - 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.



C:\CodeBlocks\Workspaces\HelloWorld\bin\Debug\HelloWorld.exe

```
Hello world!

Process returned 0 (0x0)   execution time : 0.021 s
Press any key to continue.
```

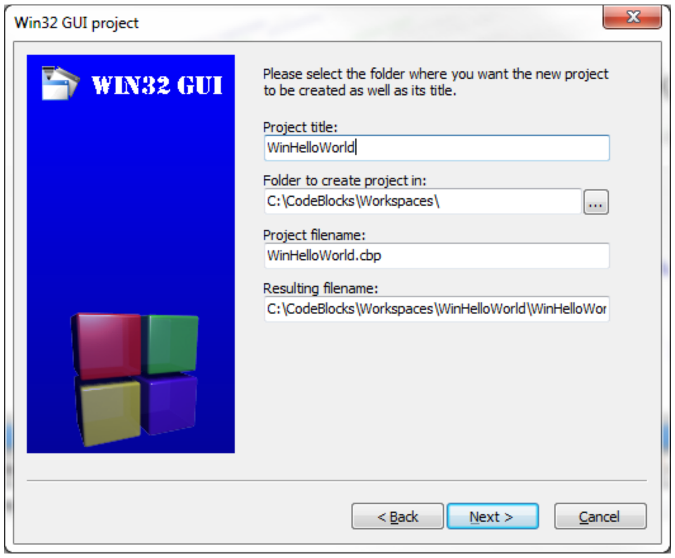
Figure 18 - Resulting console window

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 >”



Please select the folder where you want the new project to be created as well as its title.

Project title:
WinHelloWorld

Folder to create project in:
C:\CodeBlocks\Workspaces\

Project filename:
WinHelloWorld.cbp

Resulting filename:
C:\CodeBlocks\Workspaces\WinHelloWorld\WinHelloWor

Figure 19 - 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.

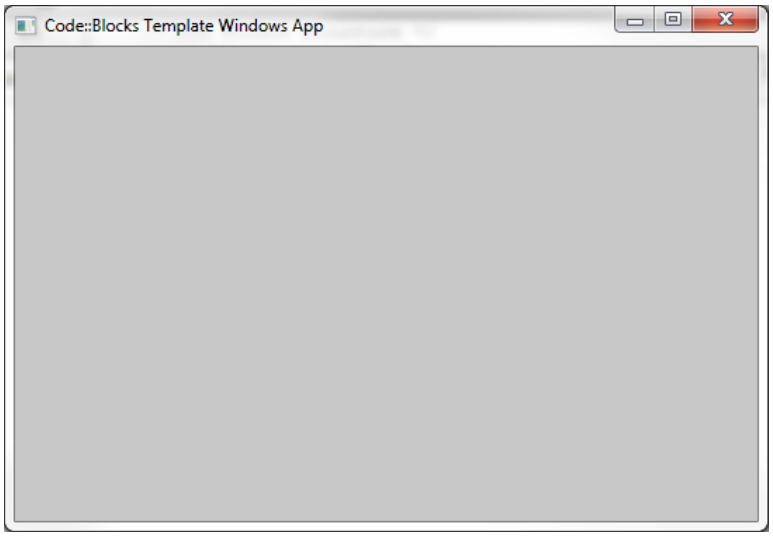


Figure 20 - Windows Program

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"

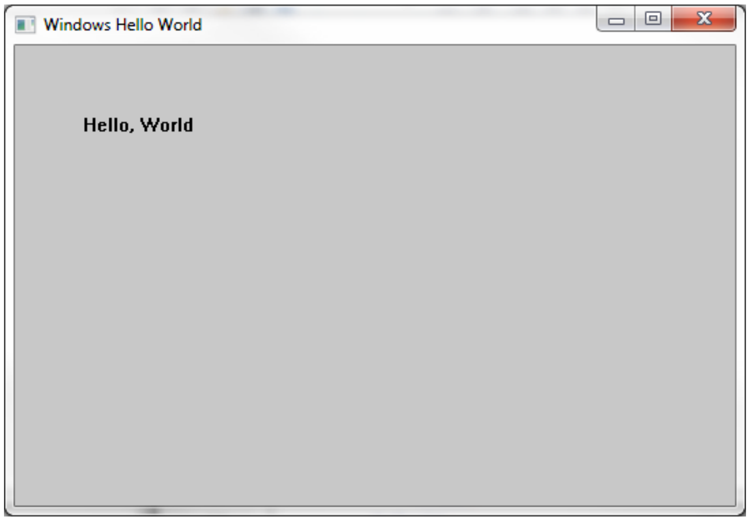
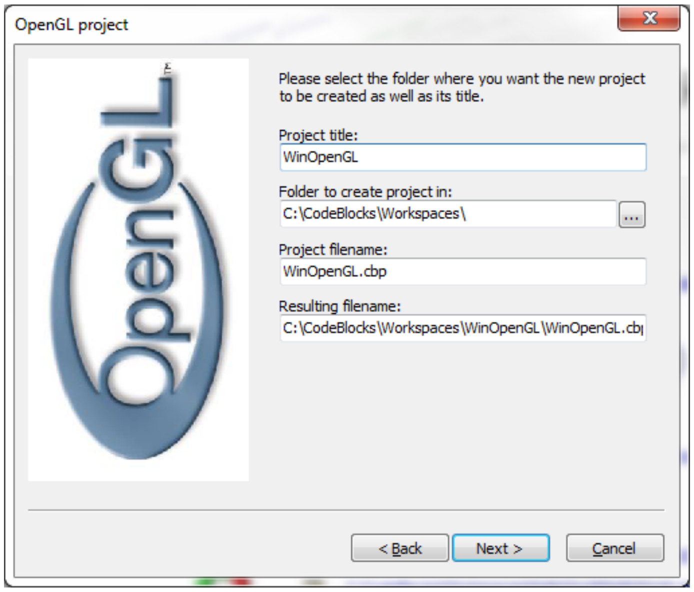


Figure 21 - Win32 (Hello, World) text

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"



OpenGL project

Please select the folder where you want the new project to be created as well as its title.

Project title: WinOpenGL

Folder to create project in: C:\CodeBlocks\Workspaces\

Project filename: WinOpenGL.cbp

Resulting filename: C:\CodeBlocks\Workspaces\WinOpenGL\WinOpenGL.cb

Figure 22 - 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 23 we will display a white rectangle shown in XXX>

TODO: Explain OpenGL specific code.

8. Build and Run.

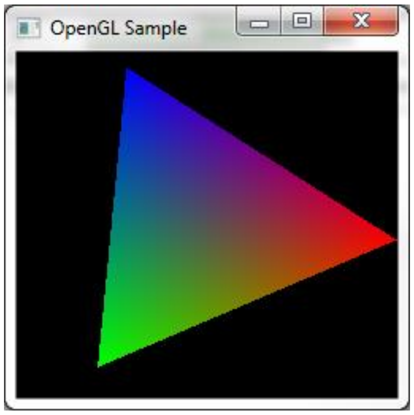


Figure 23 – Original OpenGL example

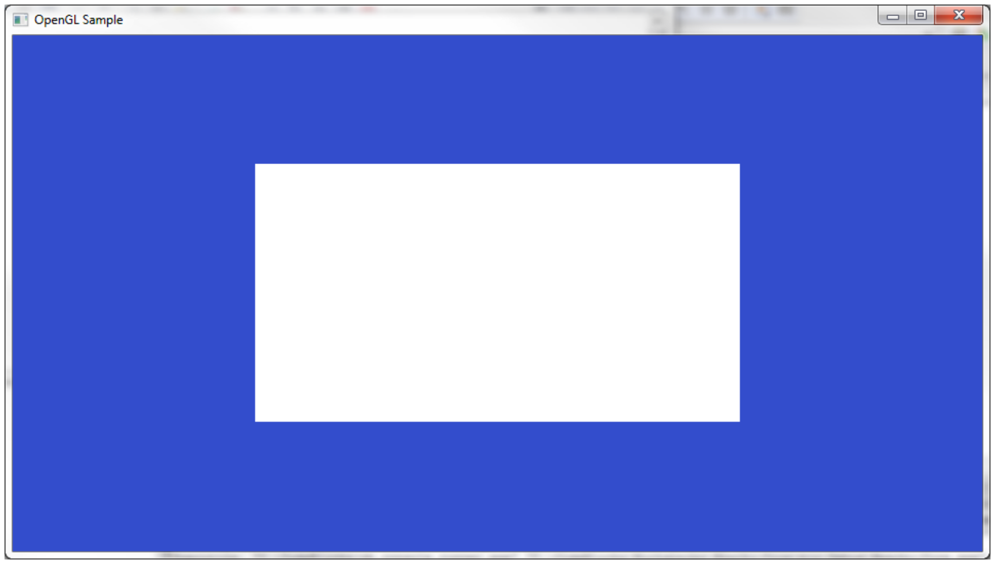


Figure 24- Modified OpenGL Example

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.

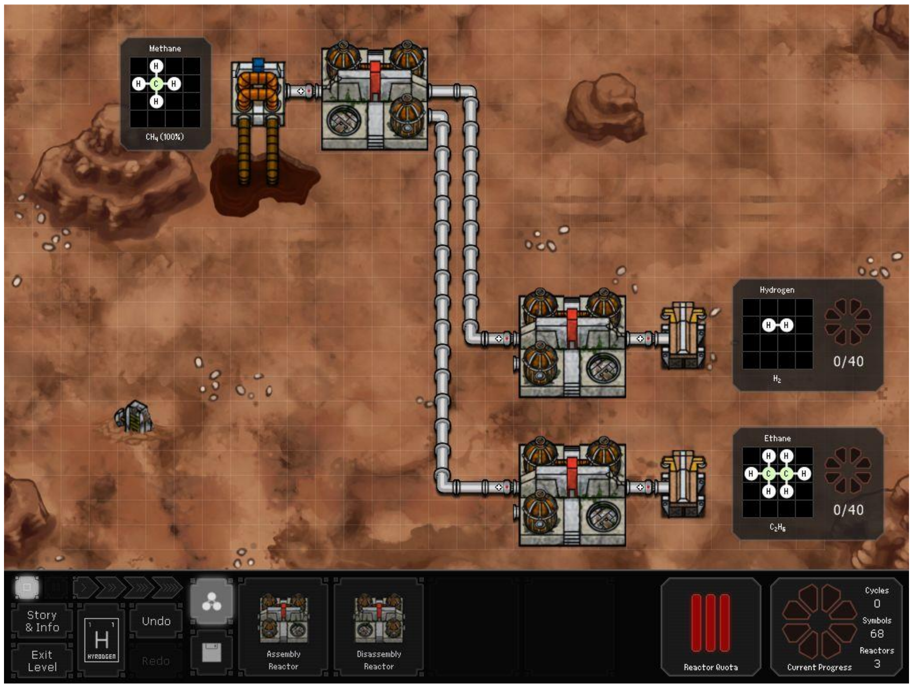


Figure 25 - Space Chem game (made with SDL)

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

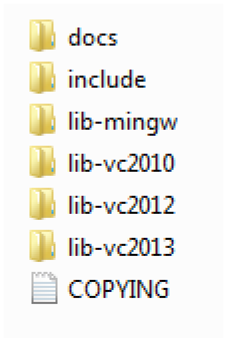
We will cover this tool in more detail when we get to the place in our project where we want to convert Sparky Game Engine to run a demo game on the browser. At this point in time there is no reason to believe that it will not be able to support GLFW with OpenGL.

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:



docs
include
lib-mingw
lib-vc2010
lib-vc2012
lib-vc2013
COPYING

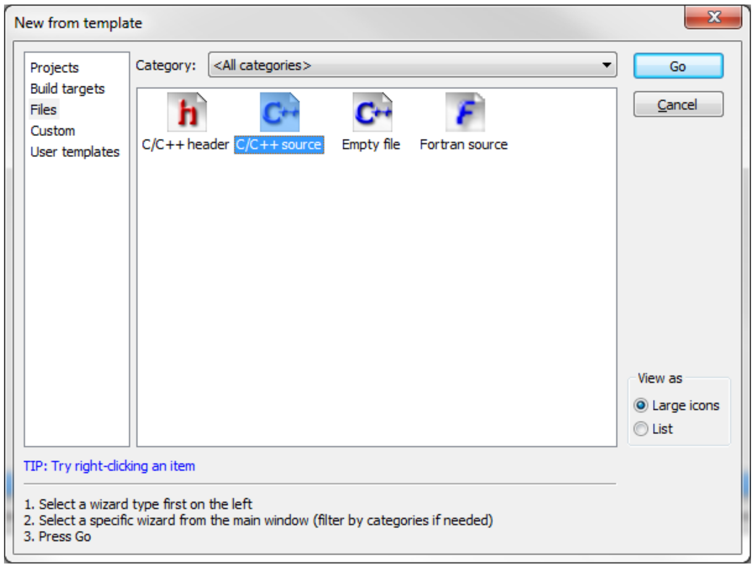
Figure 26 - Folders in GLFW

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 . . .

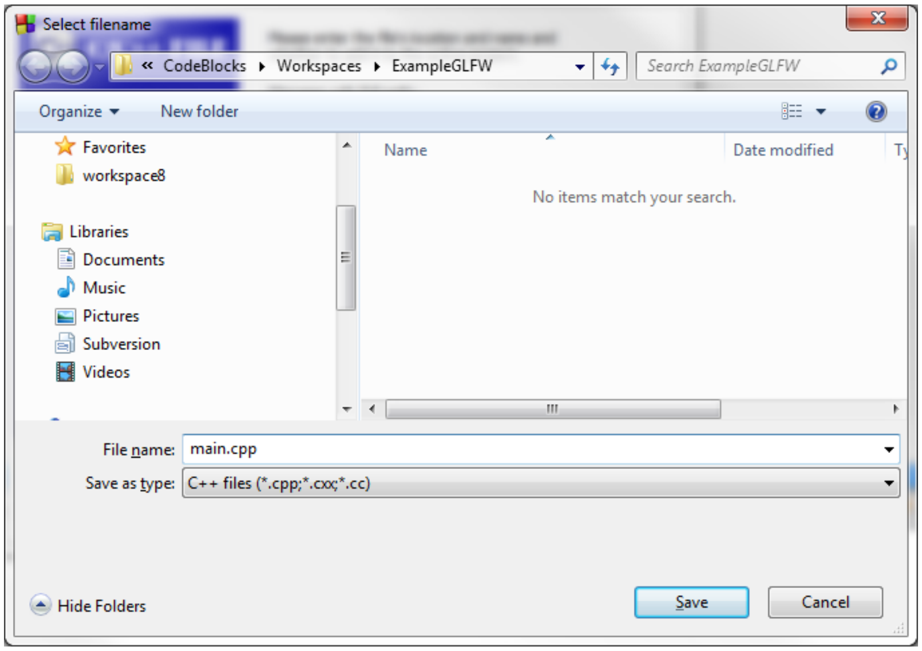


TIP: Try right-clicking an item

1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

Figure 27 - 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



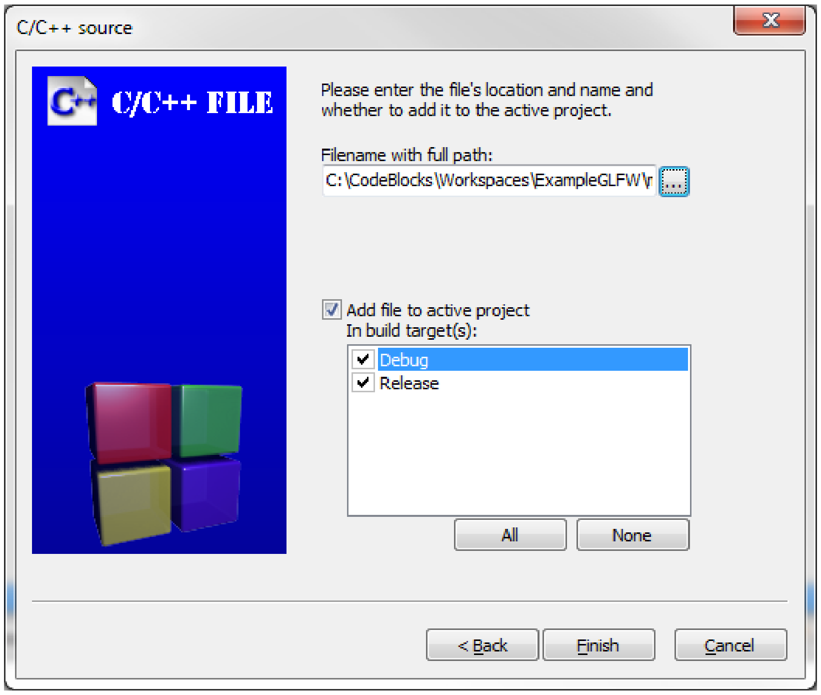
No items match your search.

File name: main.cpp

Save as type: C++ files (*.cpp;*.cxx;*.cc)

Figure 28 - 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


C/C++ source

Please enter the file's location and name and whether to add it to the active project.

Filename with full path: C:\CodeBlocks\Workspaces\ExampleGLFW\r

[✓] Add file to active project

In build target(s):

[✓] Debug

[✓] Release

Figure 29 - Creating main.cpp file

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

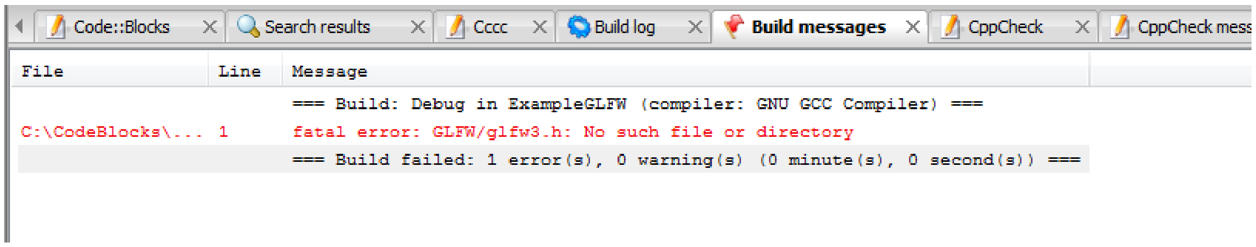
The editor will open with an empty C++ file.

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.



```
File                Line   Message
                           === Build: Debug in ExampleGLFW (compiler: GNU GCC Compiler) ===
C:\CodeBlocks\...   1      fatal error: GLFW/glfw3.h: No such file or directory
                           === Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Figure 30 - 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 26.

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

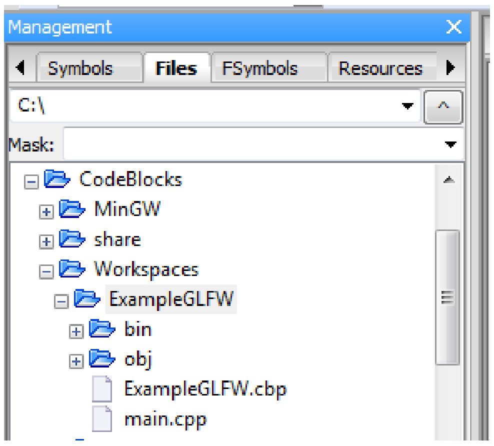


Figure 31 - Creating a folder

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

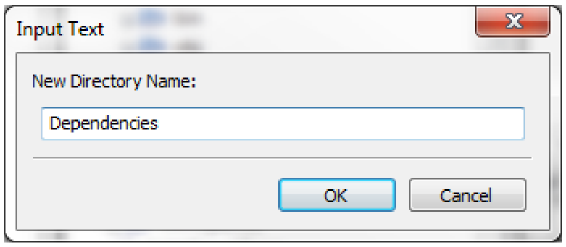


Figure 32 - Creating "Dependencies" directory

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

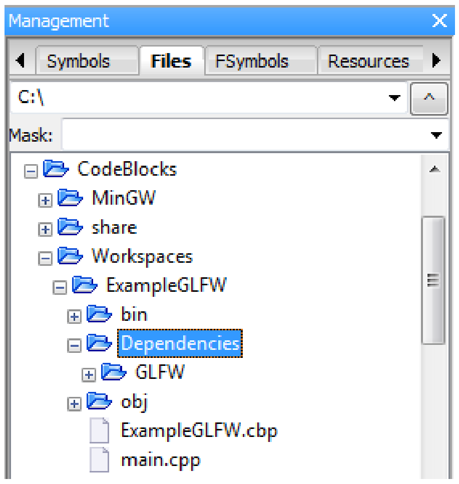


Figure 33 - Creating directory for include files

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

16. Select Project → Build options… menu option

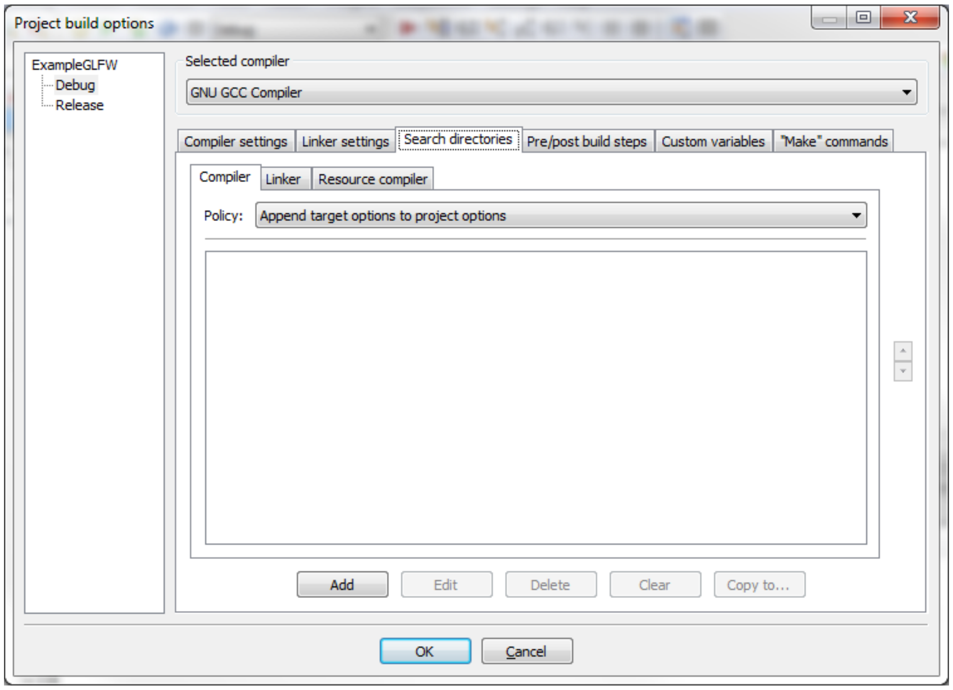
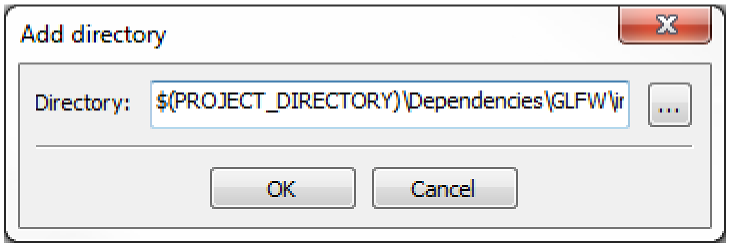


Figure 34 - Adding include to the search directories

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

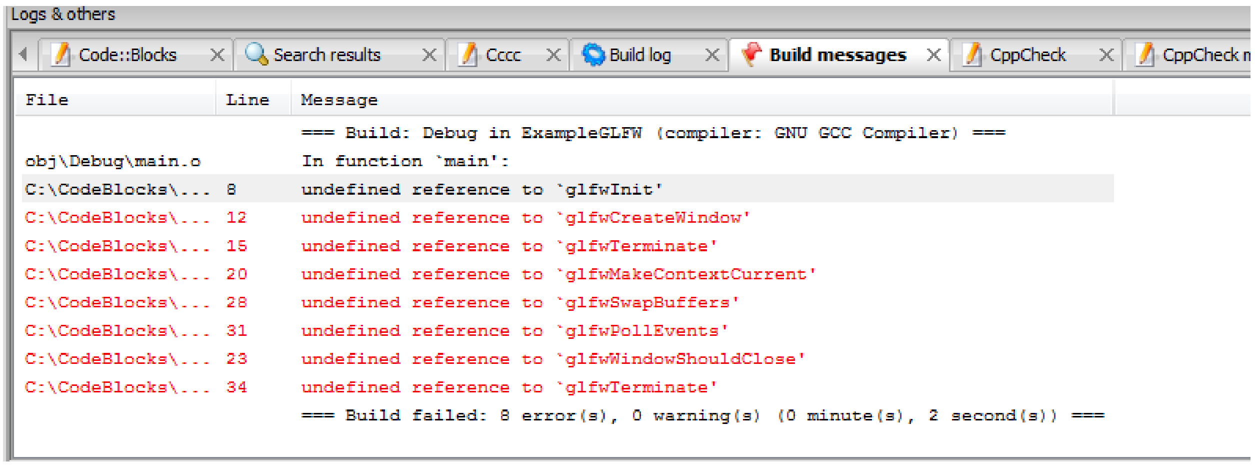


Directory: $(PROJECT_DIRECTORY)\Dependencies\GLFW\ir

Figure 35 - 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.



```
File              Line   Message
                         === Build: Debug in ExampleGLFW (compiler: GNU GCC Compiler) ===
obj\Debug\main.o         In function `main':
C:\CodeBlocks\... 8      undefined reference to `glfwInit'
C:\CodeBlocks\... 12     undefined reference to `glfwCreateWindow'
C:\CodeBlocks\... 15     undefined reference to `glfwTerminate'
C:\CodeBlocks\... 20     undefined reference to `glfwMakeContextCurrent'
C:\CodeBlocks\... 28     undefined reference to `glfwSwapBuffers'
C:\CodeBlocks\... 31     undefined reference to `glfwPollEvents'
C:\CodeBlocks\... 23     undefined reference to `glfwWindowShouldClose'
C:\CodeBlocks\... 34     undefined reference to `glfwTerminate'
                         === Build failed: 8 error(s), 0 warning(s) (0 minute(s), 2 second(s)) ===
```

Figure 36 - 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

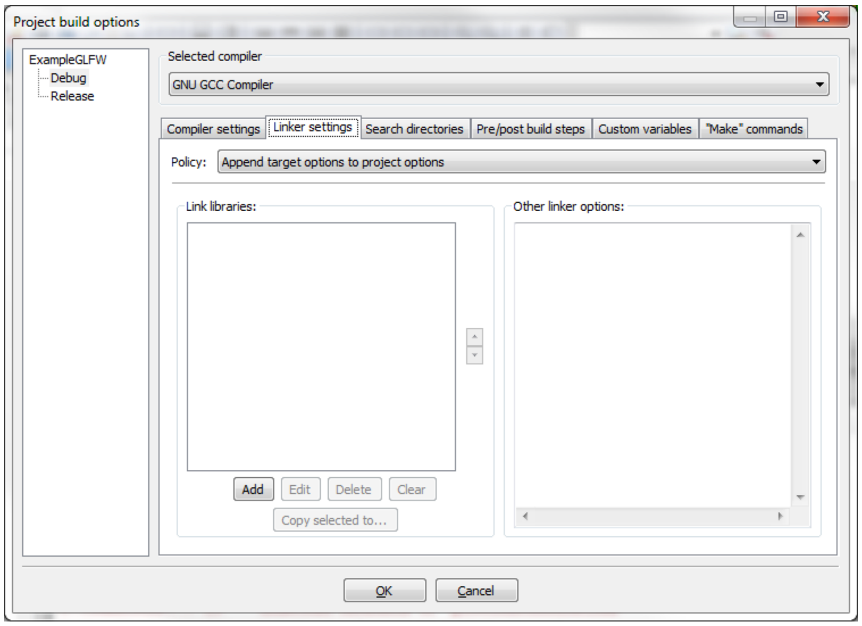
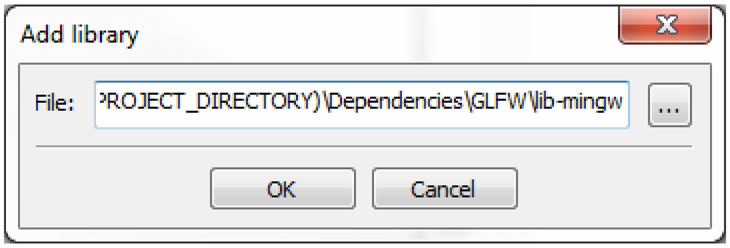


Figure 37 - Adding the GLFW link libraries

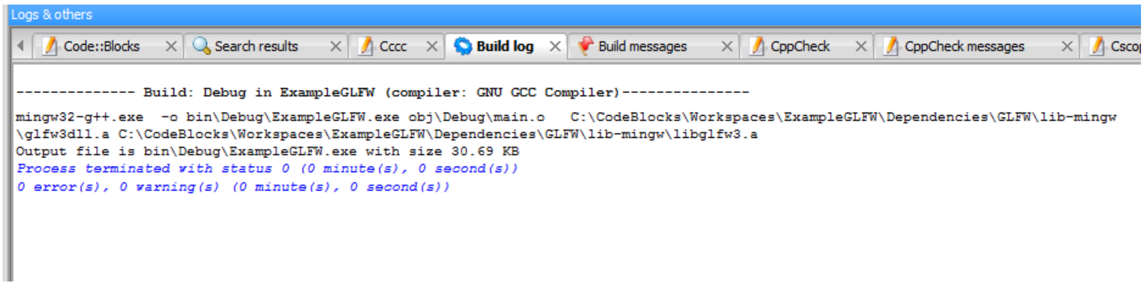
23. Click Add and enter
    - $(PROJECT_DIRECTORY)Dependencies\GLFW\lib-mingw\glfw3dll.a
    - $(PROJECT_DIRECTORY)Dependencies\GLFW\lib-mingw\libglfw3.a



File: `PROJECT_DIRECTORY)\Dependencies\GLFW\lib-mingw`

Figure 38 - Adding link libraries

24. Click OK, and Build again.



```
-------------- Build: Debug in ExampleGLFW (compiler: GNU GCC Compiler)---------------
mingw32-g++.exe  -o bin\Debug\ExampleGLFW.exe obj\Debug\main.o   C:\CodeBlocks\Workspaces\ExampleGLFW\Dependencies\GLFW\lib-mingw
\glfw3dll.a C:\CodeBlocks\Workspaces\ExampleGLFW\Dependencies\GLFW\lib-mingw\libglfw3.a
Output file is bin\Debug\ExampleGLFW.exe with size 30.69 KB
Process terminated with status 0 (0 minute(s), 0 second(s))
0 error(s), 0 warning(s) (0 minute(s), 0 second(s))
```

Figure 39 - 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.

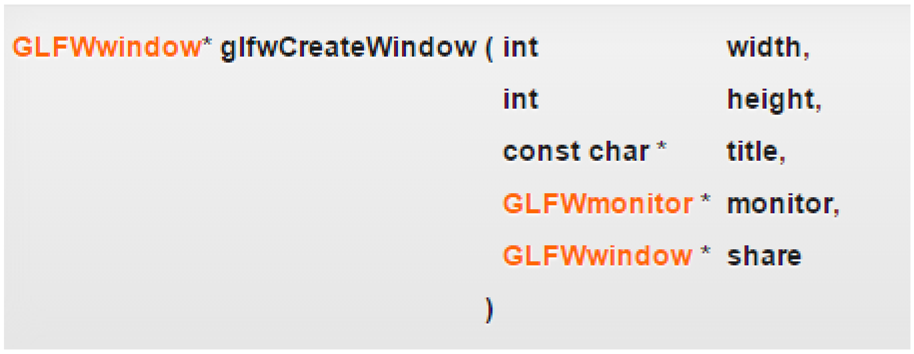
```
    /* Initialize the library */

    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 40 - 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.

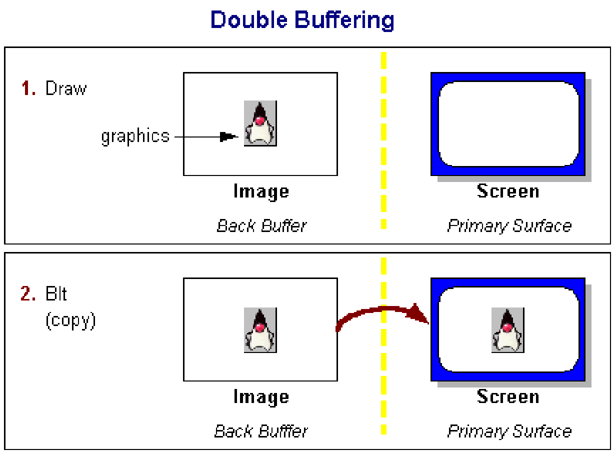


Figure 41 - Double buffering

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

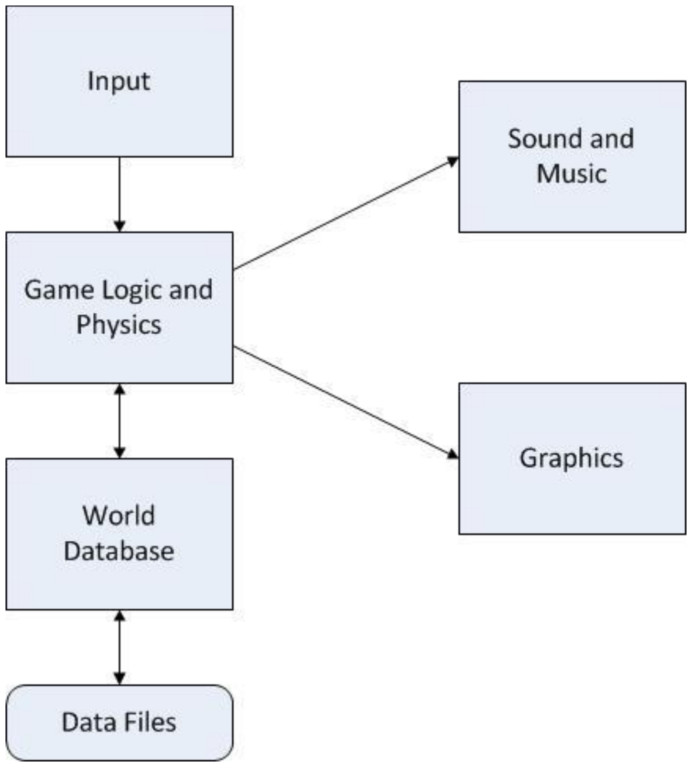
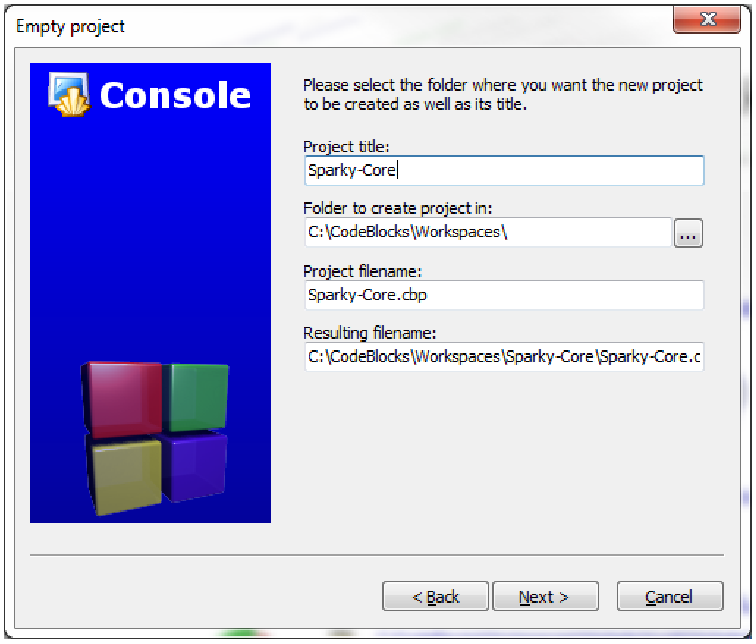


Figure 42 - Simple Game Architecture

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



Please select the folder where you want the new project to be created as well as its title.

Project title: Sparky-Core

Folder to create project in: C:\CodeBlocks\Workspaces\

Project filename: Sparky-Core.cbp

Resulting filename: C:\CodeBlocks\Workspaces\Sparky-Core\Sparky-Core.c

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.

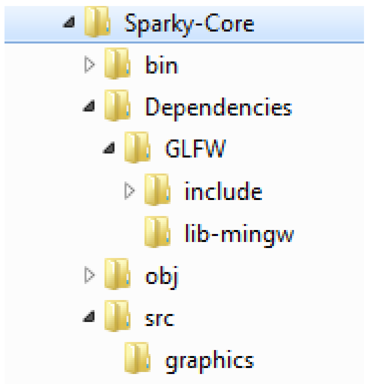


Figure 43 - Initial Sparky folder structure

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
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;
}
```

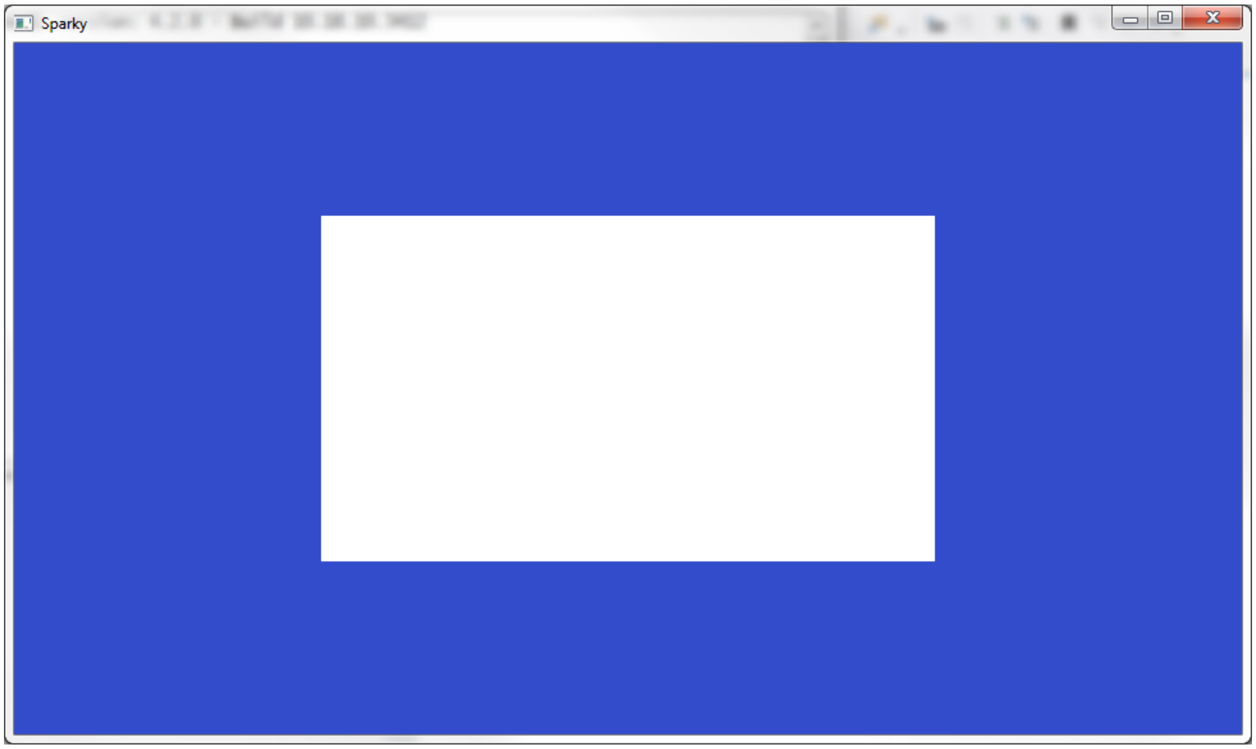


Figure 44- Episode 1 results

The final Workspace structure appears as shown below.

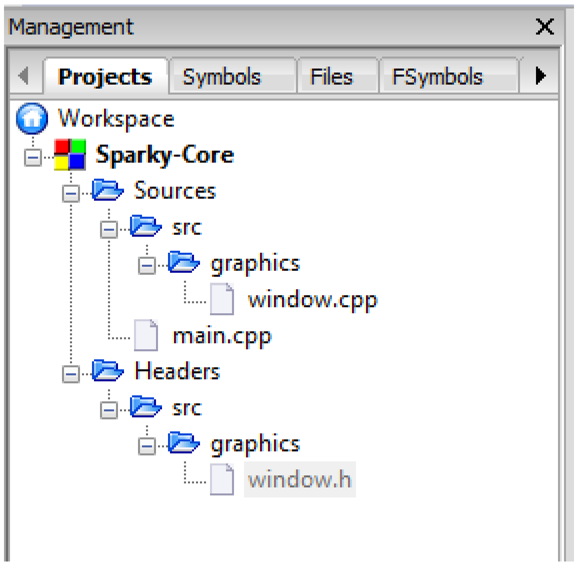


Figure 45 - Episode 1 Workplace structure

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

Episode 2 – The ???

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.

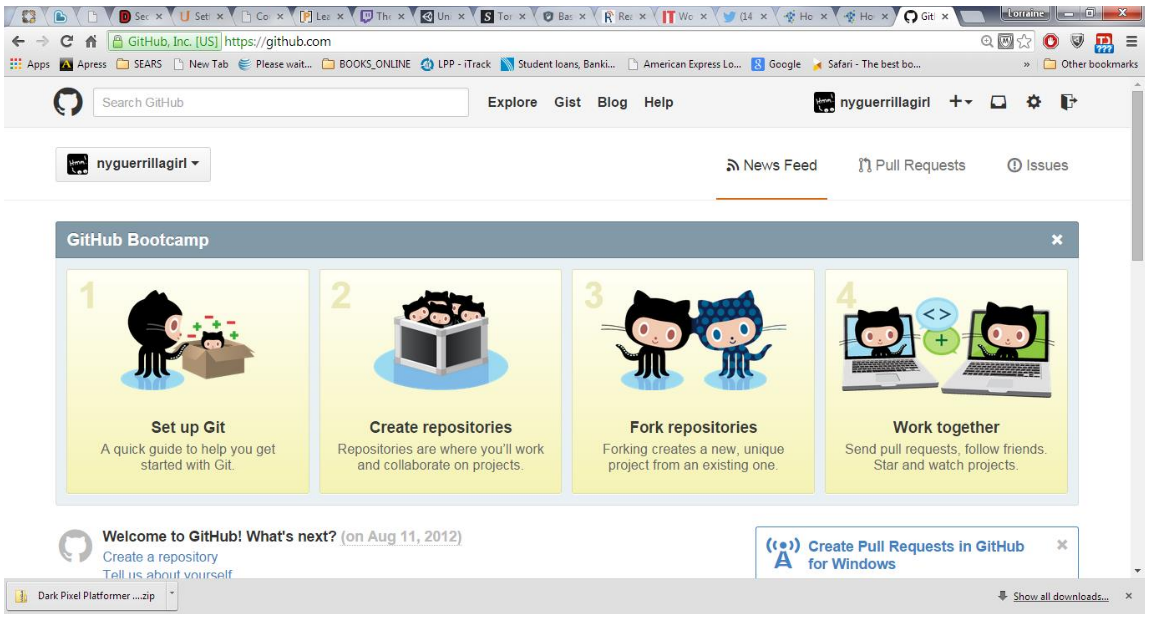
OpenGL Libraries

1. GLUT - https://www.opengl.org/resources/libraries/glut/
2. GLFW - http://www.glfw.org/
3. xx

Recommended Textbooks

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.



Set up Git
A quick guide to help you get started with Git.

Create repositories
Repositories are where you'll work and collaborate on projects.

Fork repositories
Forking creates a new, unique project from an existing one.

Work together
Send pull requests, follow friends. Star and watch projects.

Welcome to GitHub! What's next? (on Aug 11, 2012)
Create a repository
Tell us about yourself

Create Pull Requests in GitHub for Windows

Figure 46 - 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.

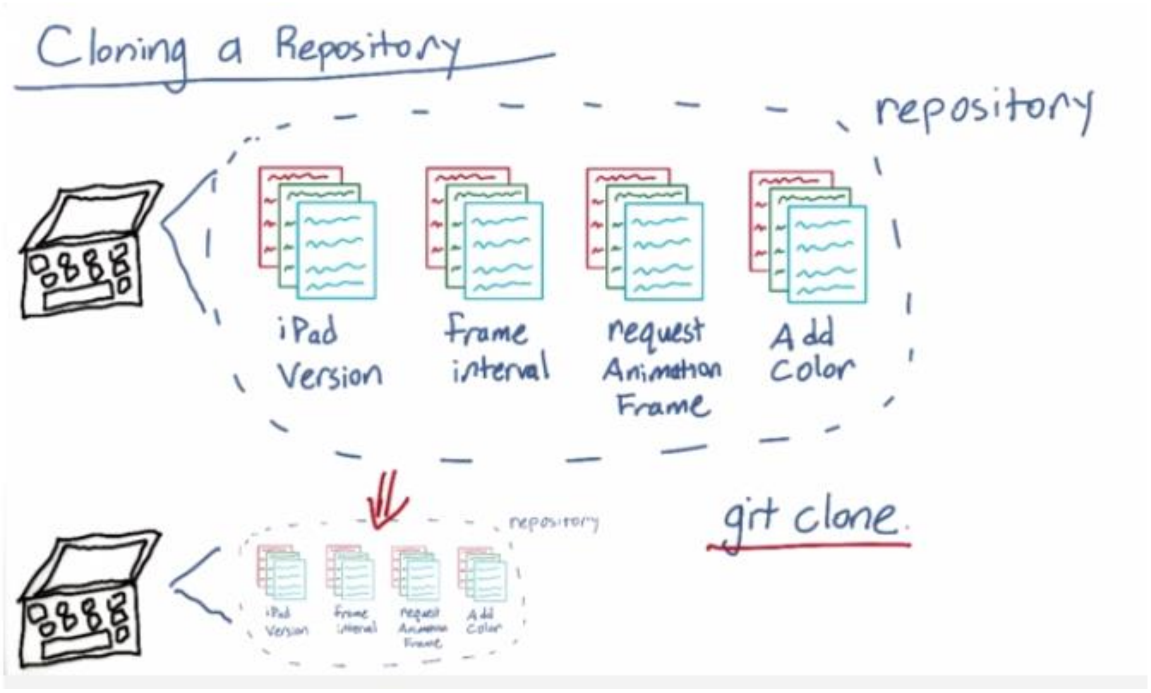


Figure 47 - Cloning a repository

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

Code::Blocks

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 48 - 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 running the application.

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[9]. 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.

GLFW is a cross-platform windowing toolkit providing support for keyboard, mouse and joystick functions for creating games.

[9] http://en.wikipedia.org/wiki/OpenGL