Handmade lite

Manual

**Copyright Notice**

Please note, that this project is not open source and therefore has no license affiliated with it. However, feel free to use the engine to build your own game or learn from the code. *Handmade Lite* is an educational tool, designed to be used as such.

Please be respectful and do not distribute the engine or its code as your own work. You may alter or tweak the engine code as a learning experience and you may also ‘borrow’ some code and functionality and use it to create your own game engine or game.

****

**Table of Contents**

**Introduction**

There are plenty of game engines out there today, and game developers and programmers do not need to be bombarded with yet another offering. Especially since so many of the current engines are free to use, to a certain degree, and robust enough to handle almost anything. So why make another one?

After creating *Handmade*, I realised the need to build something a little more lightweight, a stripped-down version of the original. Something that also serves as an educational tool, but focuses only on 2D without relying too much on third party libraries and consisting of a massive codebase. With that in mind, *Handmade Lite* was conceived, a cholesterol-free, 2D-only educational game engine, with less the saturated fat!

Compared to *Handmade*, *Lite* has less clutter and gets down straight to the basics. Of course, you can use *Handmade* to create 2D games, but then you are bogged down with changing camera projections and dealing with a larger codebase that focuses on areas pertaining to 3D graphics and shaders.

*Lite* is built purely around the *SDL* library, and while the original *Handmade* also makes use of this framework, *Lite* only uses *SDL*, not only for rendering and input, but also for the audio, text, and networking, all managed by the *SDL* extension libraries *SDL\_mixer*, *SDL\_ttf*, *SDL\_net*, respectively.

Again, the intention here is not to compete with the likes of *Unity*, but to continue offering an educational tool for game developers, programmers, and more importantly, students, such that they can create small 2D games while learning the foundations of programming, *C++* coding and *object-oriented programming*.

While the game engine is not open-source, the source code is an open book, free for all to view and learn from.

So, with love and absolute passion for programming, from one programmer to another, I give you *Handmade Lite*.

**Designed to teach. Made from scratch. Built by hand.**

**Getting Started**

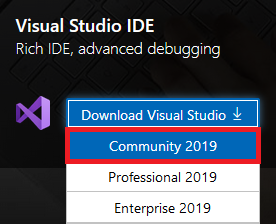
The first thing you will need is the *Visual* Studio *Integrated Development Environment* (IDE) in order to open, build and run the project. Visual Studio is …

<Talk a little more about Visual Studio>

**Downloading *Visual Studio***

You can get it free over here: [https://www.visualstudio.com](https://www.visualstudio.com/)

The latest version, at time of writing, is *Visual Studio 2019*, and I recommend that you download the *Community* edition, because it comes with enough tools for you to get started with, and it’s free!

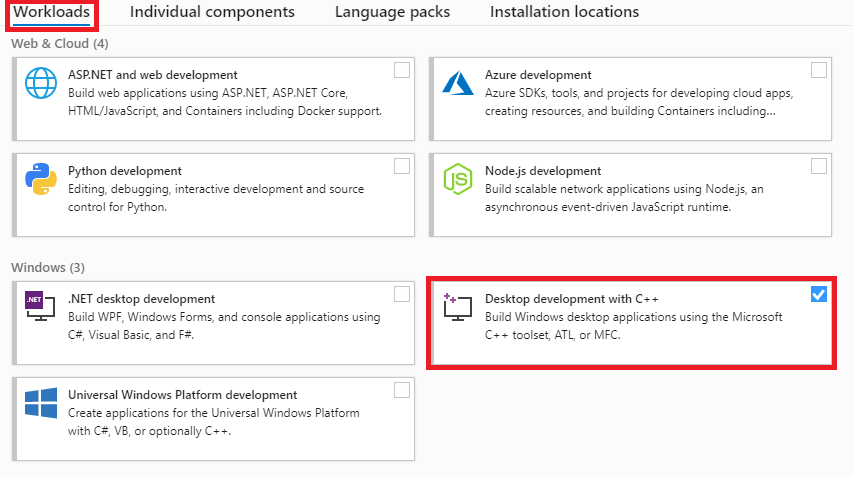


**Using the *Visual Studio Installer***

Once the *Visual Studio Installer* has downloaded, run it. Under the *Workloads* tab, select the **Desktop development** **with C++**. The components that will be installed are sufficient for what we will be doing with *Handmade Lite*. If you would like to manually choose which components to install, click on the **Individual components** tab and tick the boxes next to the components you wish to install.

Once you are satisfied, click on Install.

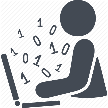
The *Visual Studio Installer* can help you manage multiple *Visual Studio* version installations. So, if you want to also install earlier version of the IDE, you can and whenever they need updating, you can do so within the installer. You can also later on return here to install the *Professional* or *Enterprise* editions if you so wish.



**Building the project**

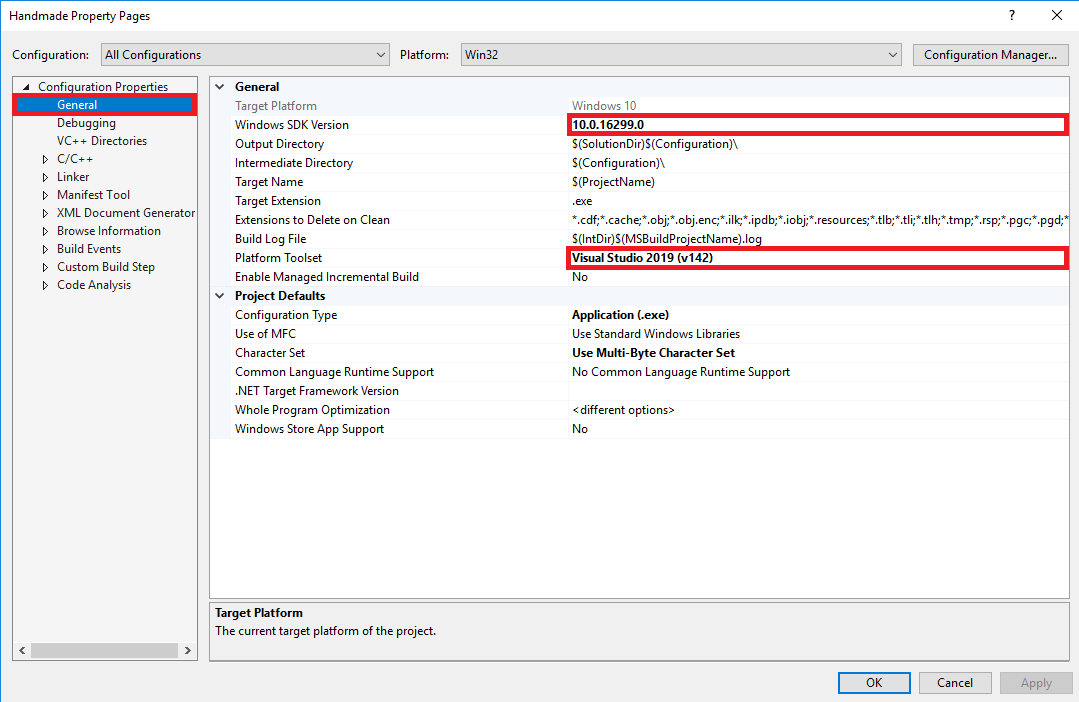
Because *Handmade Lite* was last edited in *Visual Studio 2019,* the project will by default compile and run perfectly from inside that IDE. However, if you have an earlier version of *Visual Studio*, such as *2017* or *2015*, you can still load and run the project.

Once you open the solution file, your *Visual Studio* version of choice will fire up the solution and project. However, you may have a problem when building the project because the incorrect compiler is currently set.



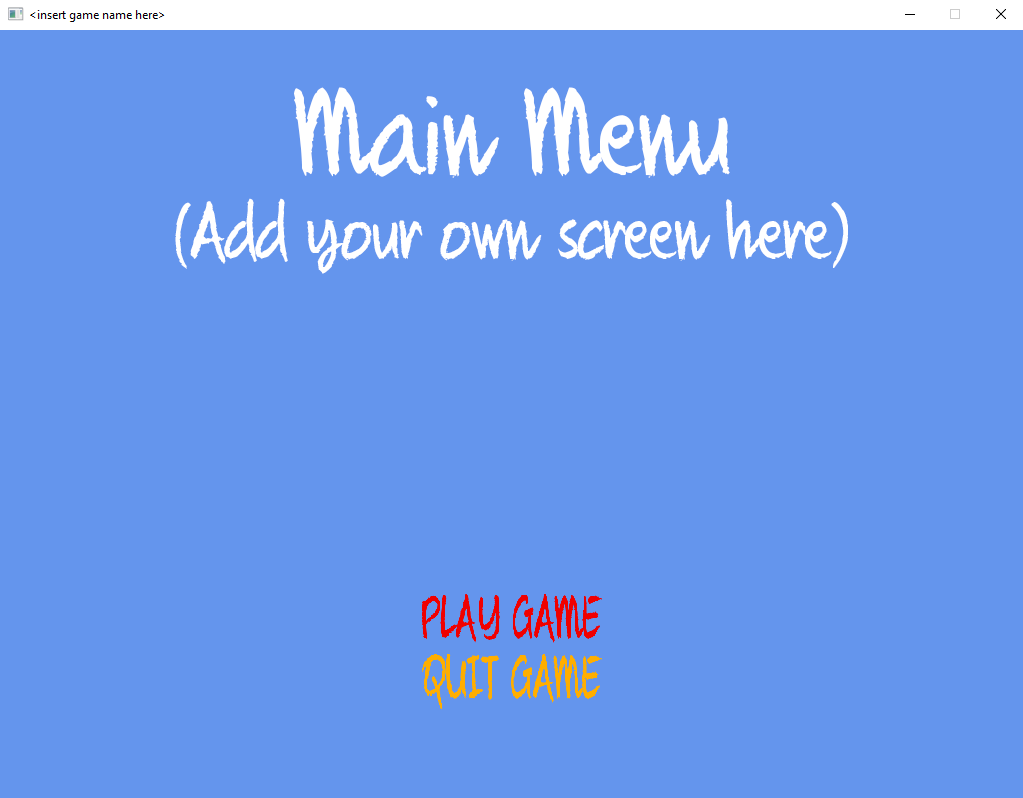
**This can be changed as follows:**

* Right-click the *Handmade* project and select **Properties**, or just hit **Alt-Enter**
* On the **General** tab, click the drop-down list next to the **Platform Toolset** option and select the compiler you have installed
* You may also need to select a different SDK, which you can do by clicking the drop-down list next to the **Windows SDK Version** option
* Select **OK**



**Running the project**

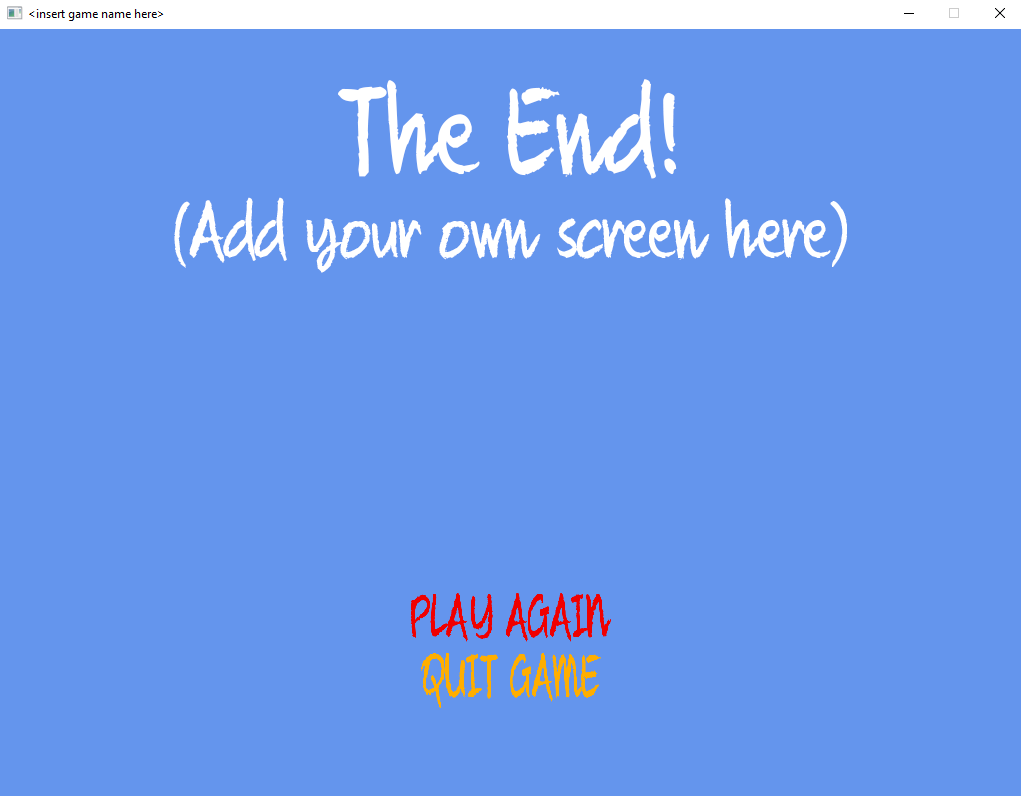
Once you have successfully setup the compiler and built the project, you should be able to run it with **F5**. The very first thing you are presented with is the default game window with a very basic menu. How exciting! You are now in the *main menu screen*.



The background screen you are presented with is merely a placeholder for the actual screen you would insert here. You have two choices – to *play* or *quit* the ‘game’. Quitting will bring you back to *Visual Studio*’s editor and selecting to *play* will enter the *play state*, which comes with yet another exciting background screen. Once again, this background image can be replaced with whatever image you desire later on.



From here, press **M** to return to the main menu or **Q** to exit the play screen and enter the *end state*.

****

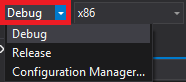
The end state is where the game will come to when the player either wins or loses. From here, you can choose to *play again* or *quit* the game altogether.

For now, don’t worry too much about the default screens, we will return to them and demonstrate how to load up your own. We will also demonstrate how the *game states* work and how to manage them, as well as create your own!

***Debug* and *Release* mode**

The engine is set up to run in both *Debug* and *Release* mode. The only difference is that in *Debug* mode a *console window* runs on the side displaying different types of feedback. In *Release* mode, you only have the game window, relieving the engine of the extra overhead that comes with the console window.

Ideally, while building and testing your game, you will want to remain in *Debug* mode and only build a *Release* version once you are ready to ship. Feel free to flip between the two modes at any time by simply selecting the option on the toolbar.



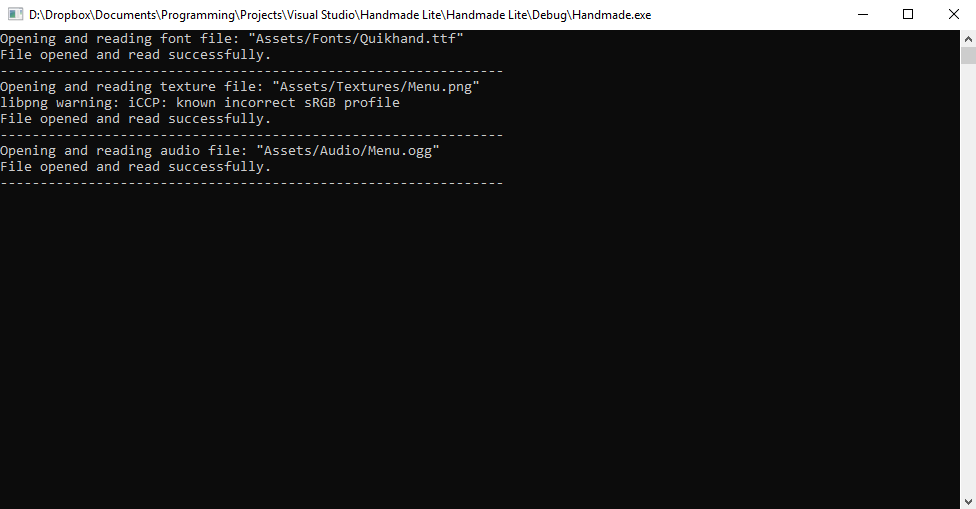
***32-bit* and *64-bit***

Currently, the game engine is built using 32-bit (x86) libraries, making it a 32-bit application. I have not yet managed to create a 64-bit version, but feel free to do that yourself if you want to. To select between the two, simply click the drop-down box next to the *Debug/Release* box and select your type.

**The console window**

The console window will display different feedback on successful tasks or dreadful issues that might occur when running the engine. This may include errors loading resource files, or certain parts of the framework not initializing properly.

For example, you may be loading an invalid file or one that was deleted, in which case it will tell you so. If any engine components did not load, it will display an error message. Always keep this terminal window close at hand, especially while debugging your application, as this will help identify any issues occurring in your code.

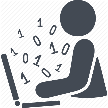


You may also wish to display some test output of your own while coding, and to do that, simply use the input/output stream. From within any of the source files, include the *iostream* header, like so:

#include <iostream>

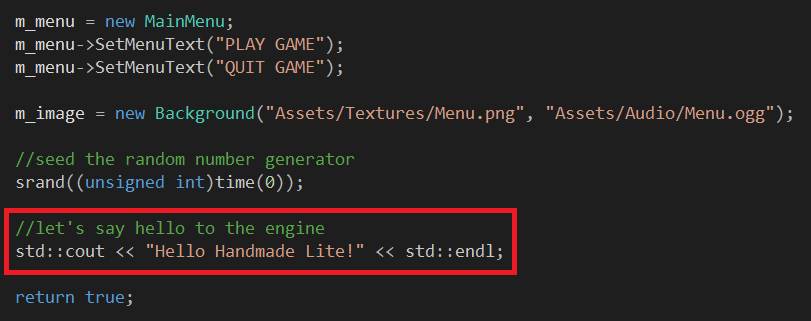
Now, anywhere in the code, write the message of your choice to be displayed in the console window:

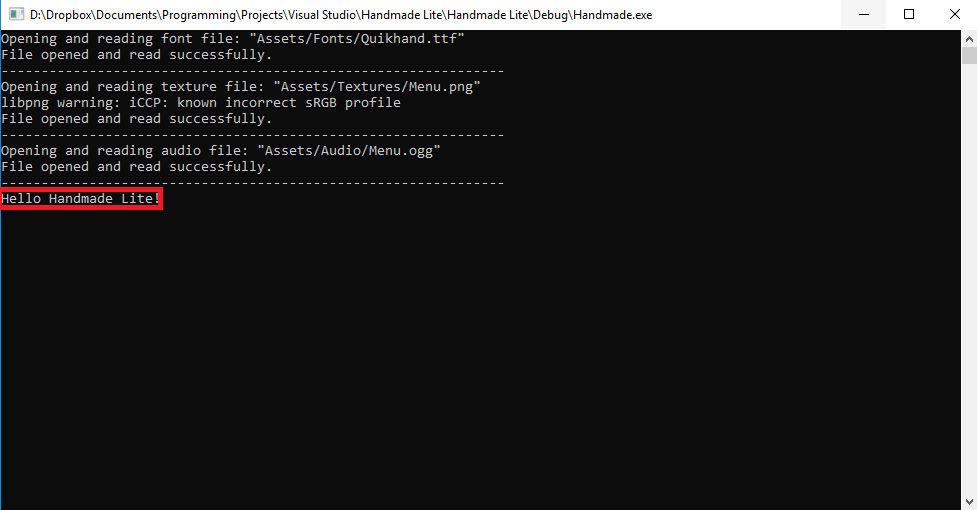
std::cout << "Hello!" << std::endl;



**Let’s try this ourselves and write a simple greeting to the screen.**

* Peruse through the codebase in your *Solution Explorer* until you find the **GameStates** sub-folder under the **Source Files** folder
* Open the folder and double-click the **MenuState.cpp**
* Within the code, include the *iostream* header file and scroll down until you find the **OnEnter()** function
* Right before the ***return true*** statement, insert your output message, something like *Hello Handmade Lite*
* Run the engine again and take a look at the terminal





Presto! Your message appears on the console window, right below the other initialization messages. Messages like this are pretty meaningless, but the point being made here is that it’s quite simple to display messages and the console window should now become your next best friend. It will undoubtedly help you resolve issues whilst debugging. You could use it to display the position of your player or the current state of your enemies.

<more on this later!>

**The project folder structure**

As with any other game engine, before building your games, it is always a good idea to get familiar with the folder structure. This will help you later on when you are looking for asset files, source files and figuring out where to place your resources, or why certain files won’t load.

*Handmade Lite* is structured as simple as possible and the following folders and files are worthy of a mention.

**The *.vs* folder**

This folder is usually hidden and contains all the solution options for every *Visual Studio* editor used to open the project. Each time you open *Handmade Lite* with a different version of *Visual Studio*, a sub-folder is created within the *.vs* folder, containing the relevant files. Furthermore, this is also where the IPCH (pre-compiled headers) are stored, if that option is enabled. Beware as this can bloat the folder and before long you may end up with MB or GB of data!

<discuss more on IPCH>

**The *DevLib* folder**

All the *SDL* headers, library and runtime DLLs are stored here, so that when building/running the engine/game, the correct files are found. If you wish to link other third-party libraries to the engine, feel free to add the dependent files in this folder.

**The *Handmade* Project folder**

In here you will find all of your source code and the Handmade Lite Visual Studio project files. Any extra source code you add to the project will end up in here.

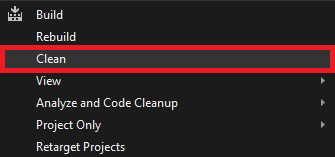
**The *Assets* folder**

This folder is stored in the *Handmade project folder*. Within this folder lie more sub-folders, each containing all the resources that will be used in your game, such as *sprites*, *textures*, *fonts*, *audio*, etc. As you build your game, you will add all of your assets into their respective sub-folders with the main *Assets* folder.

**The *Debug* and *Release* folders**

There are two sets of *Debug* and *Release* folders, one in the *root folder* of the project and one in the *Handmade project folder*. Whenever you build your game in *Debug* or *Release* mode, the respective folders will be created.

The inner *Debug* and *Release* folders, which you find in the project folder of Handmade Lite, contain all the build files for each build you create within the editor. These folders will, from time to time, begin to fill with clutter, so you may wish to clear these directories occasionally, and create a fresh new build. You may wish to completely delete these folders or, alternatively, you can also clean up these folders by using the *Clean* option from within the IDE. Simply right-click the Handmade project and select the Clean option from the context menu.



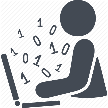
The outer *Debug* and *Release* folders, which are located in the root folder, where the **SLN** file is also located, contain the game’s **EXE** file, which is created every time you build in either *Debug* or *Release* mode. The **EXE** won’t run on its own, it does depend on various **DLL** files and the *Assets* folder. Without them, the game will not run properly.

**Building a *Distro* folder**

If you want to show off your game to your family or friends, you cannot always guarantee that they will have *Visual Studio* at their disposal, so how would you build and run the game?

The ideal solution (no pun intended) is to build a *distro*. This folder should contain everything to make the game run independently of the *Visual Studio* editor, so that anyone who wants to play the game, who does not have a copy of *Visual Studio* on their machines, does not need to first download and install the IDE.

As mentioned previously, when building your game in *Debug* or *Release* mode, a folder is generated in the root folder, containing an **EXE** file and some other database/log files. This is essentially your distro folder, but it requires some dependencies to run.

 **Let’s demonstrate how to produce a distro folder.**

* Build the project in either *Debug* or *Release* mode. The respective folder will appear in the root directory. Note however, that it is preferable to build in *Release* mode
* In the *Release* folder, you will now find a fresh, new **EXE** file and some **PDB** files (?) Remove all files except the **EXE**
* Go into the *DevLib* folder and open the *SDL* sub-folder. In there, you will find a *bin* sub-folder. Copy all the **DLL** files from there and place them in the *Release* folder
* Copy the entire *Assets* folder and paste it in the *Release* folder
* The **EXE** should now be self-contained and run independently. Rename the *Release* folder to your name of choice and use this to ‘ship’ your game.

After building a distro folder, you now have a means of distributing your game and running it, independent of *Visual Studio* – it should run straight out of the box. If you wish to show off your *Visual Studio* project as well, you can certainly do so, but you may also wish to remove any extra clutter that can easily be re-generated. You can safely delete the *.vs* folder, as well as all *Debug* and *Release* folders, as these are all reproduced when loading and re-building the project.

**Getting familiar with the engine**

**The *main.cpp* source file**

Everything starts up from the *main.cpp* file. There is a minimal amount of code in here, because we want to keep all of the main functionality of *Handmade Lite* within a robust collection of classes.

The first section of code will initialize the main core game engine and here you can set the name of the game, as you wish it to appear on the game window, as well as the game’s resolution.

if (!(TheGame::Instance()->Initialize("My Awesome Game", 1280, 720)))

{

return 0;

}

If you want to run the game in fullscreen mode, you just need to add an extra argument of **true** behind the other arguments, like so:

if (!(TheGame::Instance()->Initialize("My Awesome Game", 1280, 720, **true**)))

{

return 0;

}

After the game has been initialized, the first game state is created, which is the *Menu* state. This is the first game screen you see appear when you run the project.

TheGame::Instance()->AddState(new MenuState(nullptr));

Next, the game engine is ready to run. This is done by calling the *Run()* function, which appears as follows:

if (!TheGame::Instance()->Run())

{

return 0;

}

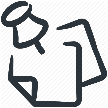
What the *Run()* function does is execute an internal loop as long as there are active game states to update and render. Once all game states have been exhausted, the function returns and the game engine closes down all sub-systems, as follows:

TheGame::Instance()->ShutDown();

The remainder of main.cpp should be pretty clear. The Game class and the game states will be discussed below.

**The *Game* class**

*Handmade Lite* runs using a *Finite State Machine*, which is managed from within the *Game* class. The state machine manages all game states, be that adding, updating, rendering and removing them. The *Run()* function is probably the most important part of the class and the main loop in here will continue to run until the game has been flagged to end. While active the loop checks for active game states and if there are any, it will update everything in that game state, followed by rendering all objects in the state. Time per frame is also calculated here, so that this may be used within game objects or other parts of the engine for calculations needing delta time.

 ***For a more technical description of how the Game class has been designed, please read the accompanying Technical Reference****.*

**The Game States**

As stated above, the engine runs using game states, and all states derive from a base *GameState* class. Therefore, to create your own game state, you will need to make it a child class of *GameState*. There already are three default game states built-in, as you have witnessed when running the project; each game screen is a game state.

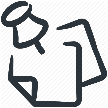
For each game state, there are four important routines that you will need to override when creating your own game state and they are:

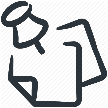
***OnEnter()*** – Here is the place to put all your start-up code, which can be anything like object initializations, resource loading or game object creation. This function will be automatically called when the said game state is added to the state machine.

***Update()*** – This function will update all your game objects and other components that the state in question contains. Here you can place specific gameplay code such as game object collision testing, distance checks, or trigger events. Before long, however, this routine can become overloaded with functionality, at which point you may want to break the tasks down into smaller functions of your own.

***Draw()*** – This function will render all of the required game objects and other components. Since this routine is set aside only for rendering, there is not much else that belongs in here.

***OnExit()*** – The opposite happens here to what occurs in *OnEnter(),* so for example all resources loaded before should be unloaded here. This function is for all shut-down tasks for the current state, such as freeing objects and resources from memory, or closing down specific objects.

 ***For a more technical description of how the GameState and subsequent game state child classes have been designed, please read the accompanying Technical Reference****.*

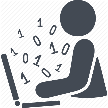
 ***We will create our own game state a little later on down the line, but let’s first cover a few other basics.***

**The *Menu* State**

The menu state is the first game state to be created and as such the first state to be executed when the project runs. As you saw previously, when running the ‘game’, you are presented with an initial screen that contains a very basic menu.

The *Menu* state is managed via the *MenuState* class, which derives from *GameState* and overrides all of the main functions detailed in the previous section. It also contains a *Background* and *MainMenu* game object, which represent the background image and menu system, respectively. The *Update()* function checks if one of the two menu items are selected and will run blocks of code based on if the **PLAY** or **QUIT** choice was selected. If **PLAY** was chosen, the *Play* state is created, and the *Menu* state is marked as inactive, ready for the transition. Alternatively, if the **QUIT** option is chosen the *Menu* state is again marked as inactive, without choosing any other game state, thereby ending the main game loop, and subsequently, the game itself.

The *OnEnter()* function creates the menu and adds the two menu items. It also creates the *Background* game object, by loading in an image and a sound to play.

 **Let’s experiment a bit with the Menu state**

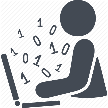
* Look for the *OnEnter()* function and add a menu item to the already existing list of menu items. How about something like **Highscores**?

m\_menu->SetMenuText("HIGHSCORES");

* Run the project and see your new menu in action.



In terms of the menu logic, we need to update a few things in the *Update()* function so that we can perform a basic action when the **HIGHSCORE** option is selected.

 **Do the following:**

* Go to the *MenuState.h* header file and add a **HIGHSCORES** token into the enum, between **PLAY** and **QUIT**. We need to make sure the order of the enums is exactly the same as the menu itself

enum MenuOption { PLAY, **HIGHSCORES**, QUIT };

* Now, in the *Update()* routine, add the following code between the **PLAY** and **QUIT** code blocks:

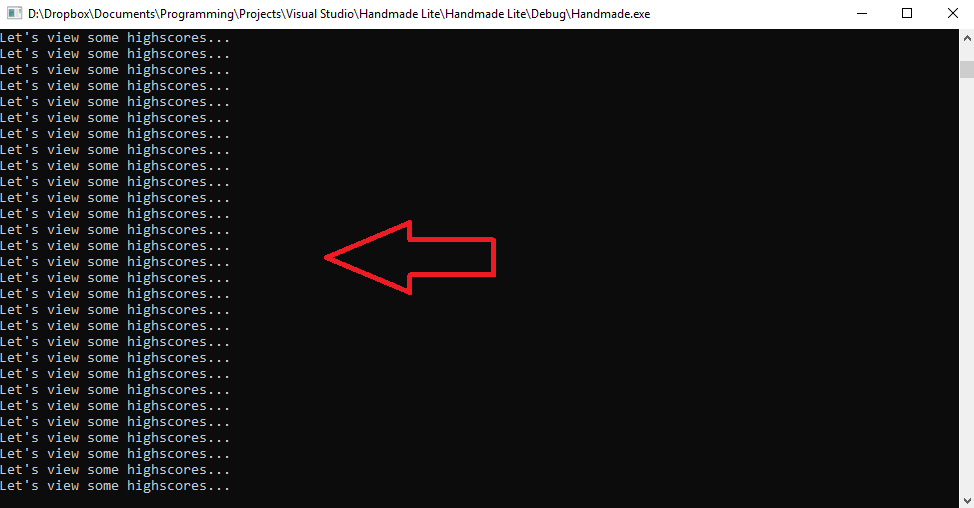
if (m\_menu->GetMenuOption() == HIGHSCORES)

{

std::cout << "Let's view some highscores..." << std::endl;

}

* You will need to include the *iostream* header file for *std::cout* to work. See above on how to do that.
* Run the application and choose the **HIGHSCORES** option.



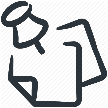
You will notice that the text is displayed multiple times and that is because we are calling it within the *Update()* function which is called many times per second.

 **Try this one out yourself:**

* Add a menu option to view a potential **TUTORIAL** for the game, thereby moving to a *Tutorial* state.
* Simply display ‘Welcome to the game tutorial’ on the console window, just as we did before.
* Don’t forget to add the enum option as well!

 **Try this one out yourself:**

* Add your own background image and music to the *Menu* state.

 ***For a more technical description of how the MainMenu and Background classes have been designed, please read the accompanying Technical Reference****.*

**The *Play* State**

Okay, so….