HANDMADE GAME ENGINE

Overview

Overview

- This small game engine has been built to be used as a teaching tool, with the added benefit of creating graphical applications and games
- It is intended to be used to create a simple **2D** game while learning intermediate and advanced C++ concepts
- The engine will allow you to create small sprite-based games, with the ability to add in **2D** animations, integrate audio, display text and handle basic collisions
- It can be customized and setup in various ways to create bespoke games or graphics applications

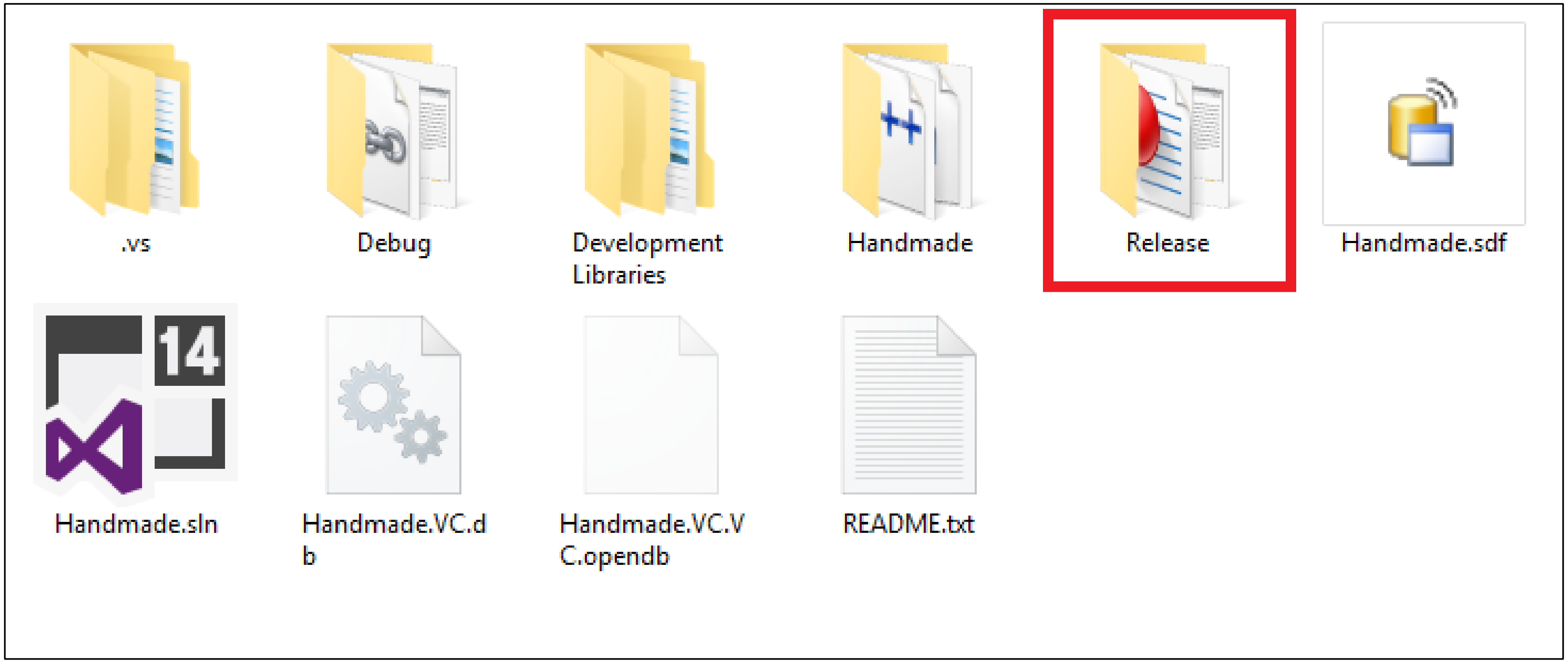
Overview

- The game engine runs from within a *Visual Studio 2015* project file.
- To run the application, load the project and build/run the program
- The game engine also makes use of external libraries and files which you do not have to worry about too much to get a game running
- The starting point of the application is from within the ***main.cpp file***.

Overview

- For each game project made with *Handmade*, there are **2** *Debug* and **2** *Release* folders
- The outer *Debug* and *Release* folders are situated in the project's main root directory. These folders contain the game's **EXE** file and subsequent **DLL** library files
- The **EXE** file, when executed, is dependent on a few **DLL**s to run, so make sure you have all the correct library files placed in the *Debug* or *Release* folder

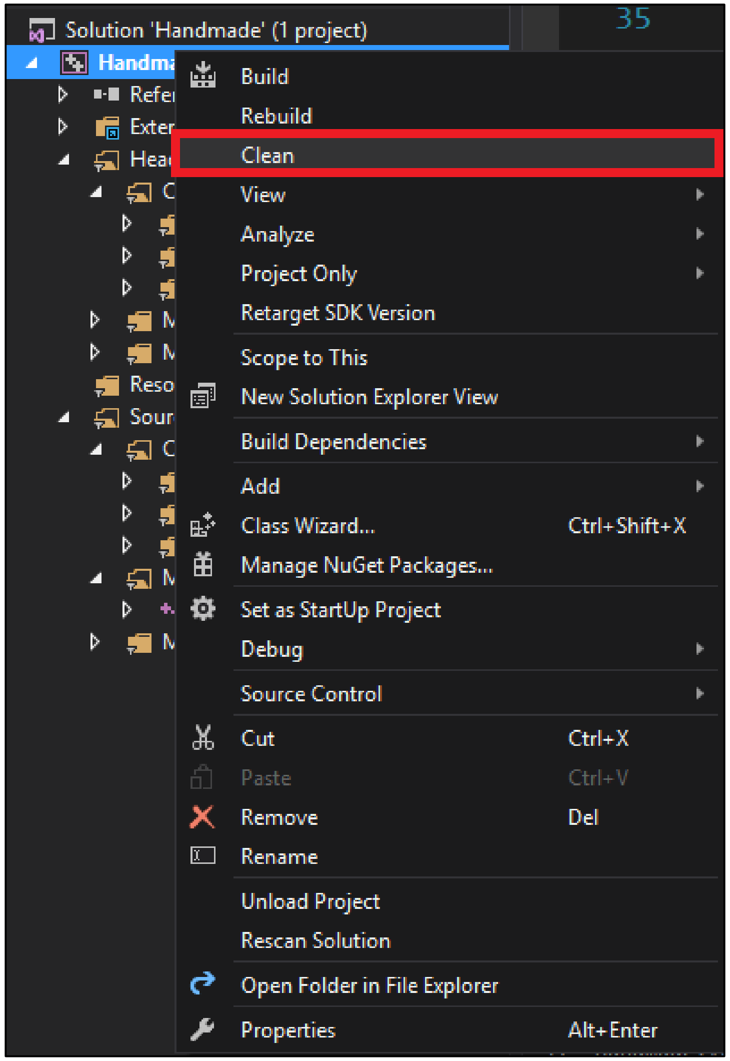
Overview


.vs
Debug
Development Libraries
Handmade
Release
Handmade.sdf
Handmade.sln
Handmade.VC.db
Handmade.VC.VC.opendb
README.txt

Overview

- The inner *Debug* and *Release* folders are situated in the main project directory, the same place where all the source code is located
- These two folders contain all the build files created when a *Debug* or *Release* build is made
- You may wish to clear these directories from time to time to create a fresh new build
- Alternatively, you can also clean up these folders by using the ***Clean*** option in the project solution's context menu

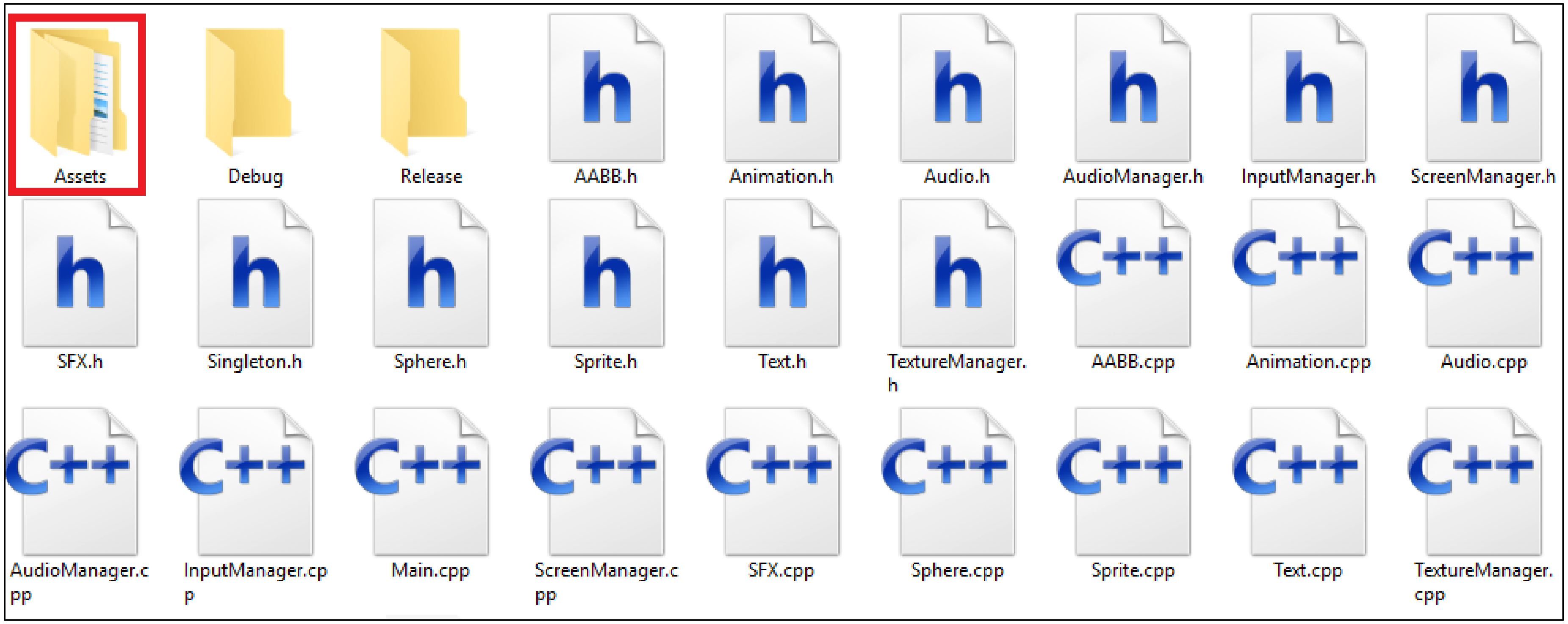
Overview


Solution 'Handmade' (1 project)
35
Handma
Refer
Exter
Head
Reso
Sour
Build
Rebuild
Clean
View
Analyze
Project Only
Retarget SDK Version
Scope to This
New Solution Explorer View
Build Dependencies
Add
Class Wizard... Ctrl+Shift+X
Manage NuGet Packages...
Set as StartUp Project
Debug
Source Control
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Unload Project
Rescan Solution
Open Folder in File Explorer
Properties Alt+Enter

Overview

- There is also an *Assets* folder located in the main project directory
- This folder contains 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
- ***Note : Make sure you copy this folder into your Distro folder when you are ready to publish and distribute your game***

Overview


Assets
Debug
Release
AABB.h
Animation.h
Audio.h
AudioManager.h
InputManager.h
ScreenManager.h
SFX.h
Singleton.h
Sphere.h
Sprite.h
Text.h
TextureManager.h
AABB.cpp
Animation.cpp
Audio.cpp
AudioManager.cpp
InputManager.cpp
Main.cpp
ScreenManager.cpp
SFX.cpp
Sphere.cpp
Sprite.cpp
Text.cpp
TextureManager.cpp

The *main.cpp* File

- The main entry point for the game is the *main.cpp* file. When you build and run your game, everything begins at this point
- Within this file, you can set the initial start-up properties, such as the screen resolution and the name of the game window :

```
if (!(TheGame::Instance()->Initialize("My Awesome Game", 1024, 768)))
{
    return 0;
}
```

- You can set the game to *fullscreen* mode as well by adding a *boolean* flag argument at the end of the game initialization function call :

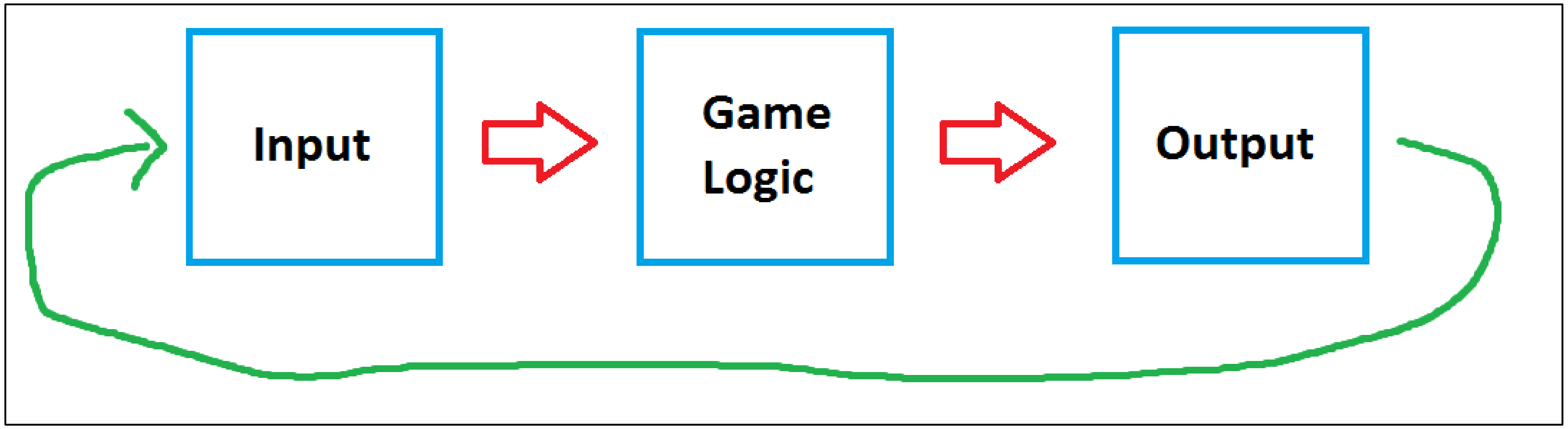
```
TheGame::Instance()->Initialize("My Awesome Game", 1024, 768, true)
```

The Game Class

- The *Game.cpp* source file hosts the main game loop that will run until the game is triggered to end
- The main loop will clear the screen, read basic input and refresh the frame buffers
- When the user decides to quit, the main game loop will exit and perform all clean-up tasks for all the sub-systems that were initialized before
- Within the main game loop, various ***game states*** are updated and rendered accordingly

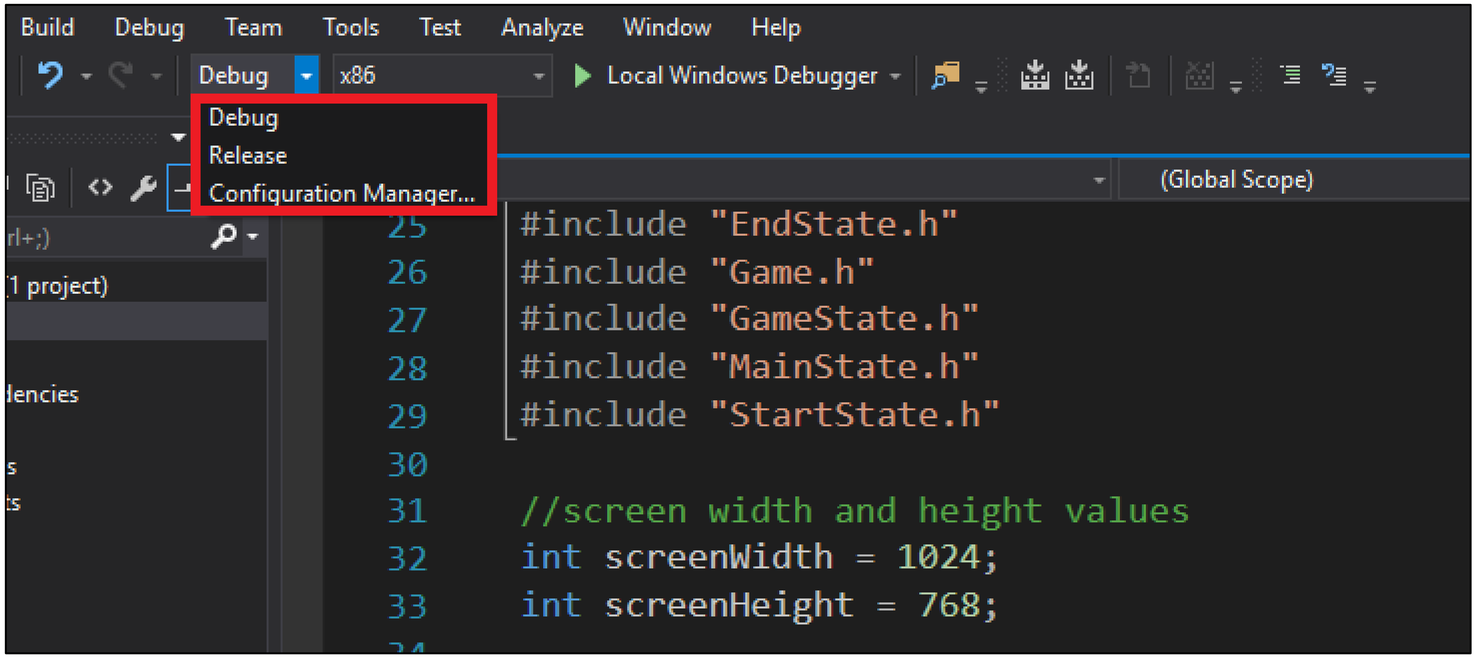
The Game Class

- Every game, at its core, will loop the same **3** main processes in a loop :



Debug / Release Mode

- The game can run in either *debug* or *release* mode, and either mode can set individually :



```
#include "EndState.h"
#include "Game.h"
#include "GameState.h"
#include "MainState.h"
#include "StartState.h"

//screen width and height values
int screenWidth = 1024;
int screenHeight = 768;
```

Debug / Release Mode

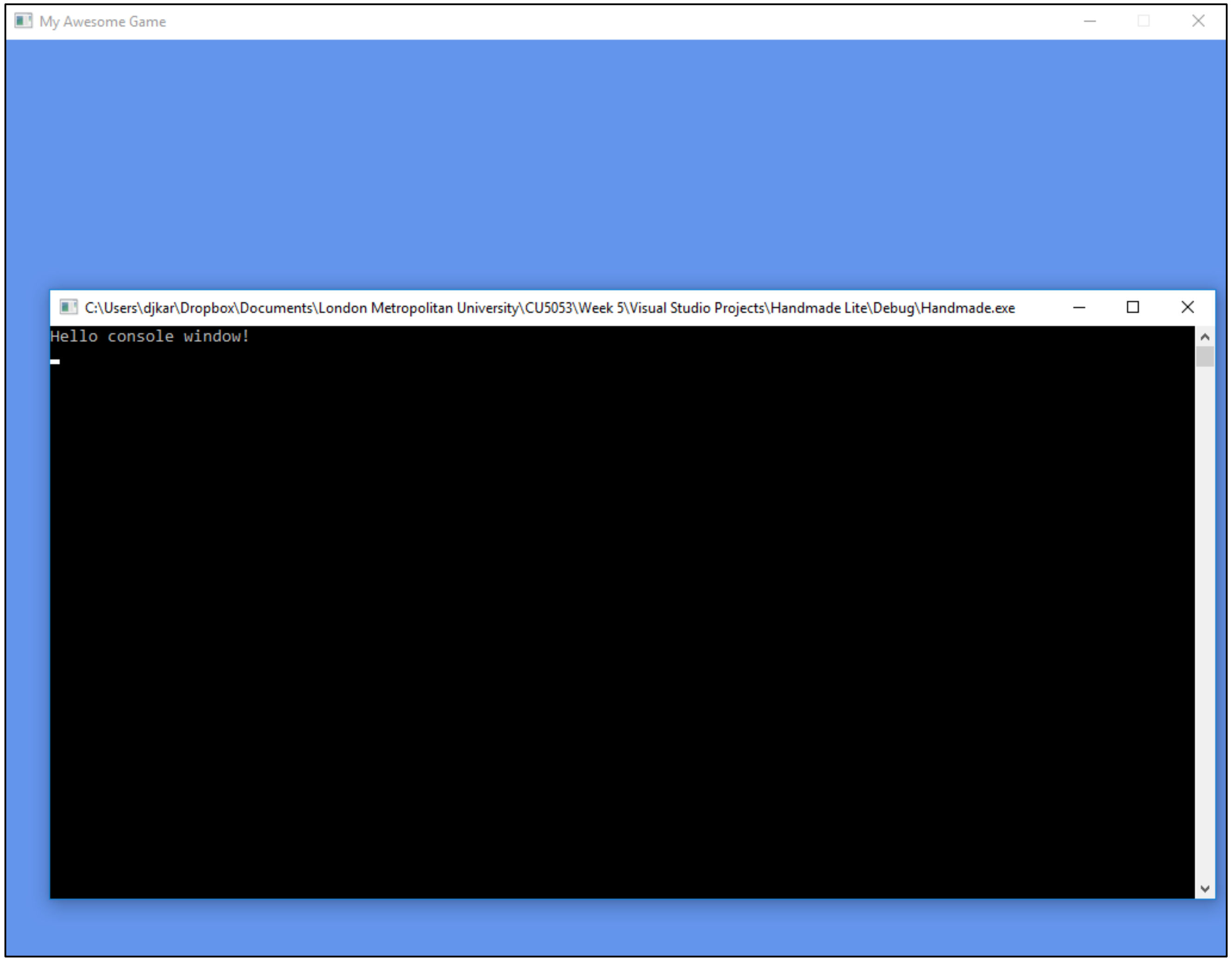
- In *debug* mode there is the added benefit of outputting debug data to the console window
- Make sure you include the following header file before outputting text to the console :

```
#include <iostream>
```

- Now, from anywhere in the game you can output text and messages to the console :

```
std::cout << "Hello console window!" << std::endl;
```

Debug / Release Mode



C:\Users\djkar\Dropbox\Documents\London Metropolitan University\CU5053\Week 5\Visual Studio Projects\Handmade Lite\Debug\Handmade.exe

```
Hello console window!
```

The Game States

- There are **3** main game states that run within the game, namely :
- ***Menu State***
- ***Play State***
- ***End State***
- Each game state will control, update and render a wide range of ***game objects***
- ***Note : If you are comfortable enough to do so, you may wish to add and manipulate you own game states***
- ***Note : We will work predominantly in the <u>PlayState</u>***

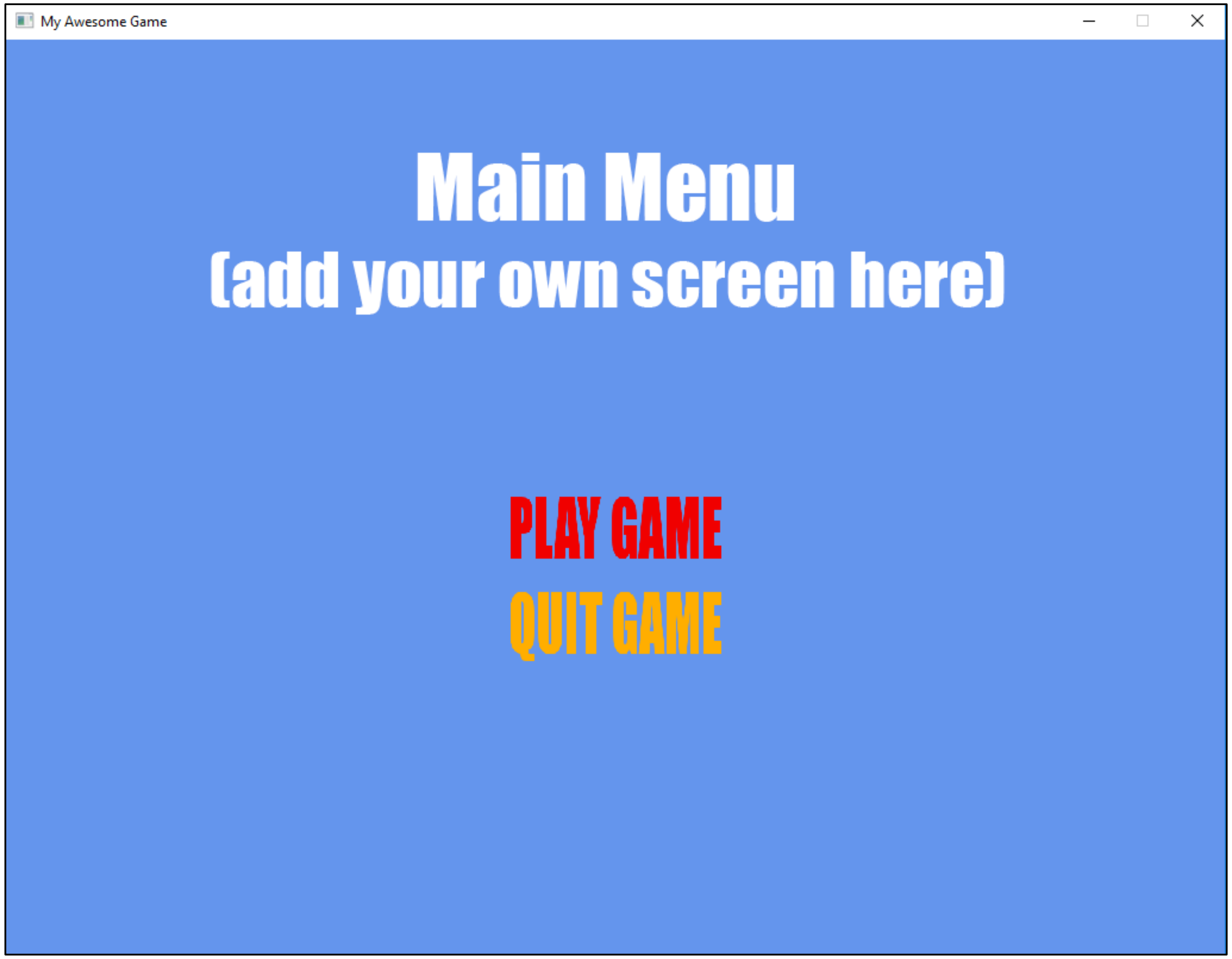
The Game States

- There are **4** main functions that run while any of the above game states are active :
- ***OnEnter()*** : All game objects and initialization for the game happens here
- ***Update()*** : All game objects are updated in this routine
- ***Draw()*** : All game objects for the main state are rendered here
- ***OnExit()*** : All shutdown tasks for the main game state occur here

The *Menu* State

- This is the state you encounter when you first run the application.
- It presents you with a choice to **PLAY** the game or **QUIT** entirely. Use the keyboard to manoeuvre through the main menu
- Everything in the *Menu* state is controlled in the *MenuState.cpp* file.

The *Menu* State



The *Menu* State

- For instance if you want to add a menu option, simply add one in the *OnEnter()* function :

```
m_menu->SetMenuText("HOW TO PLAY");
m_menu->SetMenuText("DO SOMETHING");
```

- Make sure you also add the corresponding *enum* values in the header file that will reflect the menu choices :

```
enum MenuOption { PLAY, HOW_TO_PLAY, DO, QUIT };
```

The *Menu* State

- Now, in the *Update()* routine, we can use the enum values to respond when that particular menu item is selected :

```
if (m_menu->GetMenuOption() == HOW_TO_PLAY)
{
    //display instruction screen
}

if (m_menu->GetMenuOption() == DO)
{
    //do something special
}
```

The *Menu* State

- If you want to change the background image, or audio, simply load up the relevant files of your choice in the *OnEnter()* routine :

```
m_image = new Background("Assets\\Textures\\<image>", "Assets\\Audio\\<audio>");
```

- ***Note : All background images are usually stored in the <u>Textures</u> folder and all audio in the <u>Audio</u> folder. These are sub-folders within the <u>Assets</u> folder. You may of course label these folders and store your resources as and how you wish***

The *Menu* State

- Similarly you can also change the main menu font by loading any **TTF** font file of your choice in the *MainMenu* class :

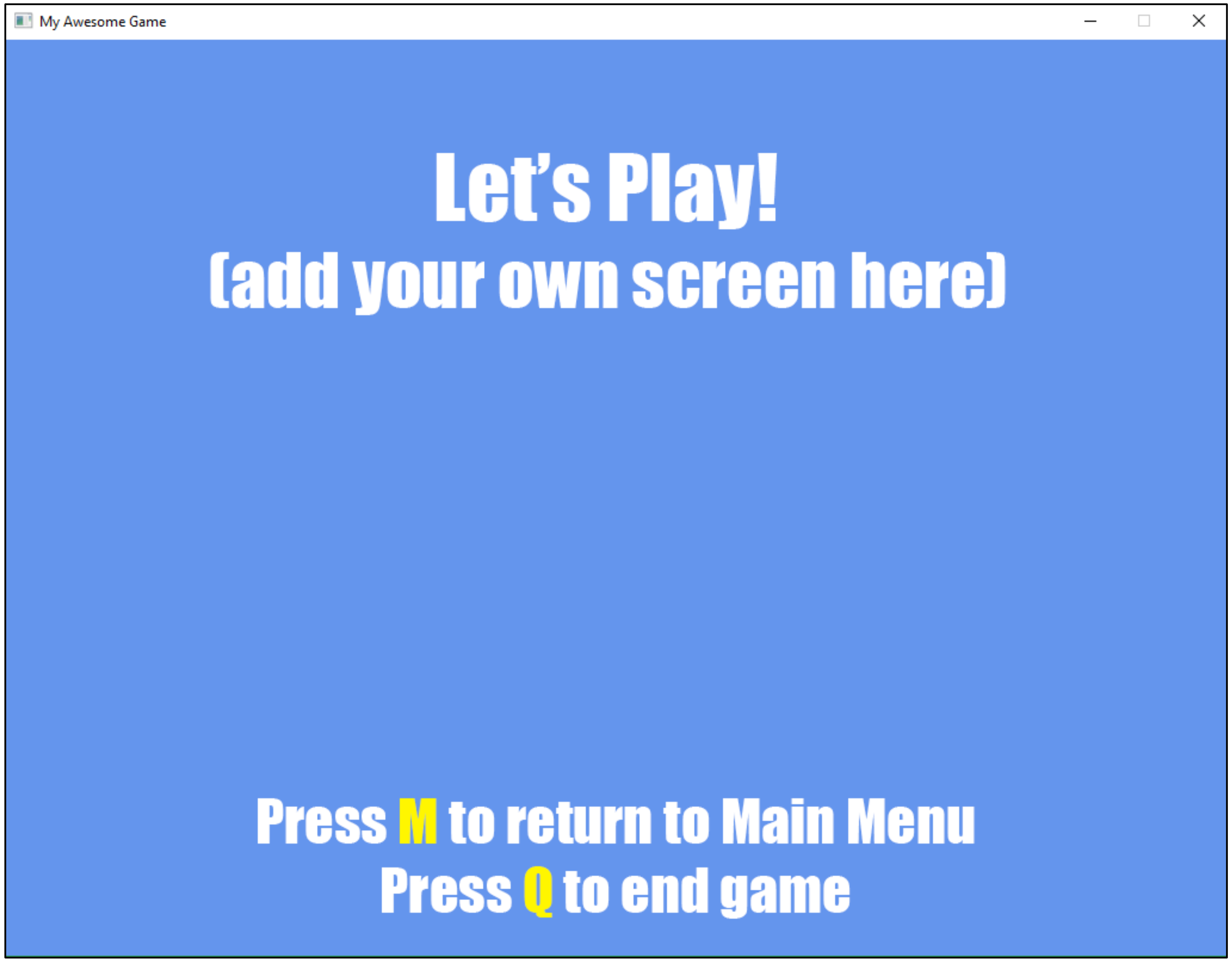
```
TheTexture::Instance()->LoadFontFromFile("Assets\\Fonts\\<font>", 100, "MENU_FONT");
```

- ***Note : Fonts are usually stored in the <u>Fonts</u> sub-folder of <u>Assets</u>.***
- ***Note : We will cover images, audio and fonts in more detail in the coming slides***

The *Play* State

- Once you hit **PLAY** in the main menu, you will automatically enter the *Play* state, where the main game will run
- All code that controls the *Play* state resides in the *PlayState.cpp*, and we will spend much time within the *Update()* and *Draw()* member functions
- To change the background image and audio, simply load the file of your choice in the *OnEnter()* routine, just as we did in the *Menu* state
- Right now, the play state will stay running until either the **M** or **Q** key is pressed, which will return to the menu or end the game, respectively

The *Play* State



My Awesome Game

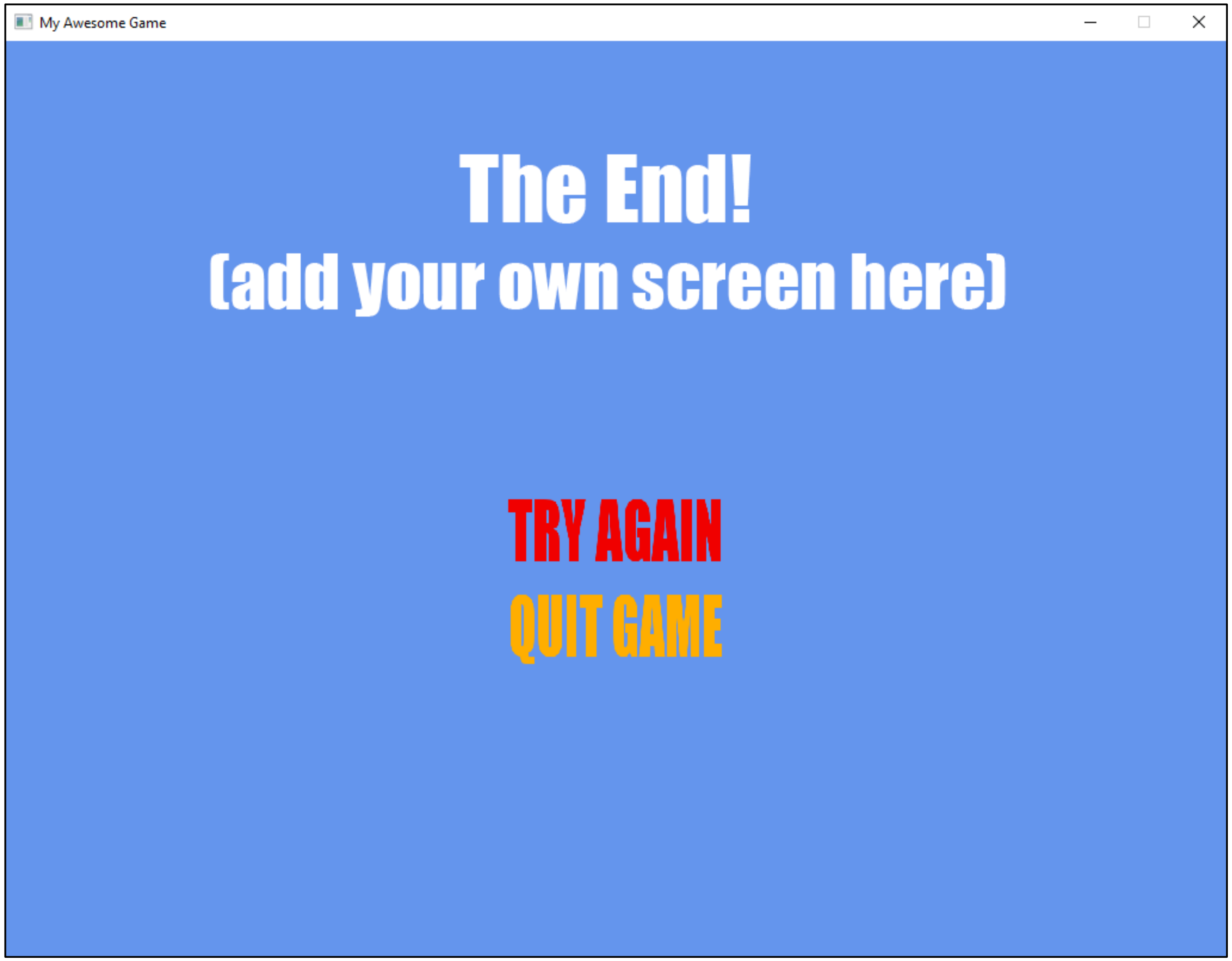
Let's Play!
(add your own screen here)

Press M to return to Main Menu
Press Q to end game

The *End* State

- The *End* state is reached once the game ends, or the user presses the **Q** key in the *Play* state
- This state also has a menu with options to *try again* or *quit* the game entirely.
- Similar to the *Menu* state, you can change the background image, audio and the menu items, if you wish to add more functionality later on

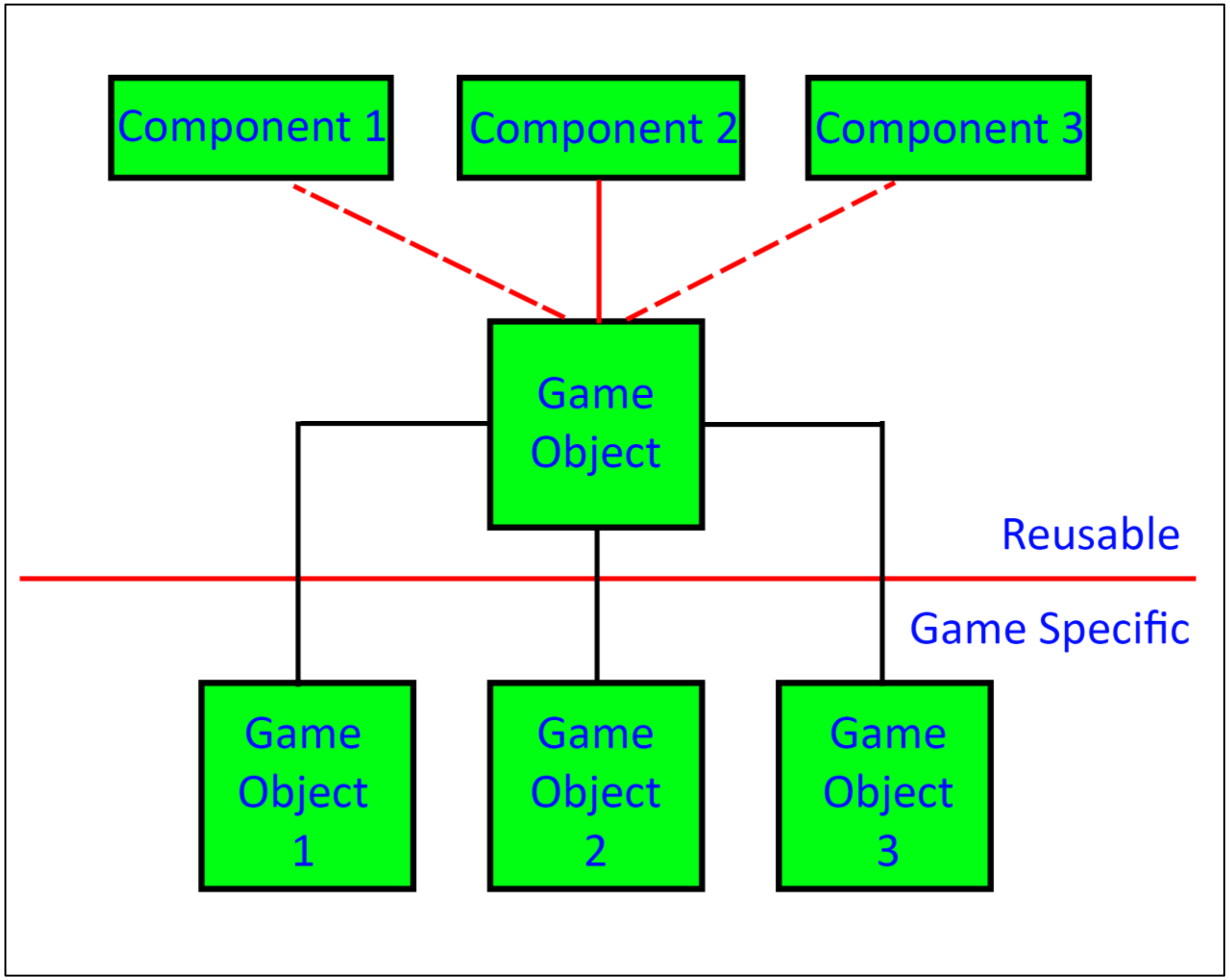
The *End* State


My Awesome Game

The End!
(add your own screen here)

TRY AGAIN
QUIT GAME

The Game Object



The Game Object

- To create our game object and make use of it, we can do the following :

```
class MyGameObject : public GameObject
{

public:

    MyGameObject();
    virtual ~MyGameObject();

    virtual void Update();
    virtual bool Draw();

private:

     //private declarations and components

};
```

The Game Object

- There are **2** main functions that need to be overridden inside each game object we create :
- ***Update()*** : All code related to updating the object, such as its position or rotation, as well as reading keyboard or mouse input goes here
- ***Draw()*** : This routine is reserved for rendering the game object in the scene

The Game Object

- Take a look at the pre-built *Background* class, which derives from *GameObject* and overrides the *Draw()* function to render the actual background image :

```
bool Background::Draw()
{
    m_image.Draw();
    return true;
}
```

- Because every game state has a background, it needs to be created in each game state, so that we can actually make use of it

The Game Object

- To do that we can either declare and instantiate the game object on the stack, like so :

```
Background background;
```

- Or we could create a pointer to the game object and instantiate it on the heap in the *Play* state's *OnEnter()* function :

```
Background* background;
background = new Background();
```

- We could also add the game object to the pre-defined game object *vector* that already exists in the *Play* state

```
m_gameObjects.push_back(new Background());
```

The Game Object

- To actually render the image on screen, we need to call the game object's *Draw()* function and the best place to do this would be in the *Play* state's own *Draw()* routine :

```
bool PlayState::Draw()
{
    //render the background (stack)
    background.Draw();

    //render the background (heap)
    background->Draw();
}
```

- ***Note : If you add the game object to the main game object vector, then it will automatically be called***

The Game Object

- Similarly the *Menu* and *End* state have a *MainMenu* game object which encapsulates the menu for each state
- The main menu is updated and rendered each frame and the corresponding *Update()* and *Draw()* calls reside in the states' *Update()* and *Draw()* member functions
- ***Note : Remember to always manually destroy your game object if you use a pointer to instantiate it on the heap!***
- ***Note : We will be creating many of our own game objects in due course!***

Summary

- For the rest of the slides (and module), we will spend most of our time within the *Play* state
- The following slides will cover more in-depth *Handmade* demonstrations on :
- ***Sprites, Animations & Text***
- ***Audio, Input & Collision***