**Conor. T. Johnston**

**BSc (Hons) Computer Games (Software Development)**

**Module: Games Programming 3**

**Matric No: S1219419**

I confirm that the code contained in the file (other than provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award.

Signed: Conor Johnston

**Documentation on Code:**

The code in the project works by creating a class of the major objects that are utilized throughout the game and instantiating those objects into the main .cpp folder, which in this case is named Studentmain.cpp.

To start off the classes I will take a brief moment to mention that the following classes are not used in the actual game themselves but are present in the coursework folder both .h and .cpp files: cColours, cCube, CPyramid, cShapes, cSkybox and cXboxController.

Using Studentmain.cpp as the sounding board I will go through the file linearly and describe the coding used and their corresponding classes as they appear.

Firstly the defining of win32 as lean and mean as well as defining it as extra lean. This refers to reducing the header file size by removing the least commonly used API’s. Progressing down the next statement of importance is the GLX\_GLXEXT\_LEGACY, this is so the game used the localised version of glx not the systems.

Moving on from the defines we reach the #imports, these statements are the way in which the classes that are made are connected to the main .cpp file. Without these statements we cannot use the function nor create instances of these classes to make game objects.

There are also #includes of things like <window.h> and <iostream> these are includes that allow us to use system process like the iostream allows us to use the input output stream of the machine to detect any devices that are connected to the machine that send signals that could be used by the game.

A major part of the game is the GameConstants class, this is pivotal in a way like no other class not even the main .cpp, in this class the libraries are brought into the game. That means we can use coding to create and play audio load textures and use games math language. But most importantly it loads in OpenGl which is the open source language used for this games development. It also defines the major variables that the game will use such as the player position and energy levels, these variables are global therefore it allows access to other classes to change them without the need for parameter passing.

This brings us to the start of the main coding of the game, the creation of the window. We go on to making variables that hold the windows height, width and bpp, we use these values later to pass into an instance of the cWNDManager whose purpose is to create the window which the game will be shown on.

cWNDManager as the name suggests is the class that deals with the window, it can construct a window with no parameters specified or all the required parameters specified. It also attaches instances of the cSoundMgr, cFontMgr, cInputMgr and WindowOGL to the window.

The WindowOGL is the class containing the matrices for the view, such as the projection and view matrices.

The cSoundMgr is the managing class for sounds, this class works closely with the cSound class. The cSound class being the class that takes the information provided to it and loads the desired sound into the game, play it, stop it or delete it. The sound manager allows there to be more than one sound in the game by holding the information forwarded to it from the cSound class, and likewise send commands to delete certain sounds.

The cFontMgr’s purpose is near enough the same as the sound manager except this class is the font manager, it works closely with the cFont class whose purpose is to load in fonts from the filenames passed into it as well as determining their size and position on-screen. The font manager regulates this information by passing the information into the font class, it also allows there to be more than one font in the game as well which would not be the case with just the font class. Like the sound manager the font manager also controls the commands to delete fonts as well as add them to the game.

The cInputMgr class purpose is to monitor what keys are being pressed this includes the mouse buttons and adding that information to the appropriate buffer.

Back to the main .cpp file there is an instantiation of the previous four classes , which the previous three are then attatched to the cWNDManager object known as pgnWNDMgr.

This object goes on to create a window with exceptions to be thrown in the event of an error or NULL reference. Proceeding to the next part of the code is the referencing of the cTexture class whose job is to use opengl to map the textures in a way that they will fit their models. Several of these textures are created for the different points they are used throughout the game, each with their appropriate parameters and file path to the images they are going to texture.

The next section talks about the instantiation of the Gamepad class which is one of the more interesting classes. *(This class was developed with the help of a tutorial referenced at the end of the document)* This class uses Xinput in its include this is for the Xbox controller, it also contains functions to detect the movement of the analogue sticks (only the left was used in the game) and activate the motors to vibrate as well as the triggers. What was found with this class was that the float values entered into it had to be between 1.0 and 0, over or under was not affected any greater or less than these two, in-between however was used to stabilise the commands linked to them. With 0.5 values it would take a half pull of the trigger instead of a tap to activate it. There are functions to check the state of the controller which makes use of the iostream in the main cpp.

The next line mentions the cStarfield, this class is to generate the space setting of the game, it contains the usual constructors but randomises the spawning of the images textured to it through the use of random functions.

cMaterial is the next class we come across which deals with the material of the game objects, in this case since It is a star wars game it dealt with the in game Death Star, allowing the texture to be illuminated as though it had its own light source.

cLight is straight after, which as the name suggests is a class dedicated to constructing sources of light with values for their type of light generation, colour, and position of light on the scene of the game. There is also a GL created ambient light which allows us to see objects a certain way even if the previous lights that are made for the game are not switched on.

Next the scene fonts are added, this is done by adding their file paths into an array gamefonts then passing the information into the font manager while assigning the font a name to be used in the scene. The same happens to the game sounds with the array being name gamesound.

The first instance of the cCamera class which deals with the in game cameras is made, this is done by passing in the position, angle, up vector as well as the view and projection matrices into the class, the main cpp makes two of them to comply with the storyboards one object being birdsEye and the other theCamera.

cSphere class is then instantiated to create a sphere which will then be textured using one of the earlier made texture objects to make it appear like a Death Star. It will load the sphere and prep its texture and proceed to bind it to the sphere it creates, this class does not use the cModelLoader class’ binding methods.

The code proceeds to clear the buffers of the InputMgr and load models with the cModelLoader class, this class like the sound and font classes deals with loading in information from the games files and allowing it to be used in scene. It loads the specified file path of the declared object and it binds a texture to it. It also manages the cModel class like the sound manager does to the sound class, it allows for more than one model to be loaded into the game. cModel deals with getting the model file path from the model loader and using it to bring in the model while getting its dimensions and other related information to it, it can also be used to attach the sound and input managers to the models.

The main cpp then goes on to create the enemies and the player, cEnemy and cPlayer both are classes that inherit from cModel, they are a specialised form of model. The instances of cEnemy have a randomly generated movement and they are assigned a specific model to represent the enemy, finally it has constraints to stop it leaving the field of play. The instances of cPlayer are similar in this regard as well although on top of that the player class has collision detection built into it within its update function and its trigpulled function. In the update function it checks keypresses which if the spacebar is pressed it will create an instance of cLaser and fire a laser model on game scene, it also has instructions for the directional arrows. The important thing is that every time this update is called the collision detection is activated, checking if any laser objects are active and whether or not they intersect with an enemy or the field of play, if they do destroy the laser and the enemy, or if it’s the field of play just the laser. The trigpulled function was a copy of the collision detection and instantiation of a laser object; thePlayer.update() wouldn’t cover it for the laser firing due to the Gamepad code.

There are a few variables then declared for use in the main loop such as a variable to hold the strings for the font messages, validation counters and the tCount which is used to count the total elapsed time. The sound manager is also called to play the theme sound and we reach our final class’ access, albeit it’s a declaration of a vector for the lasers and enemy but it tardisWarsGame. This is like the GameConstants class and hosts two global declarations for the two vector instances of the laser and enemy classes, the main makes instances of them to use for a math calculation, in other words their movement.

Finally after all this information has been instantiated and its details called the main loop of the window, which operates while the window is running starts. The first piece of code of note is the camera switch. This calls the window objects camera function which will contain a Boolean value whose value can be changed on pressing either F1 or F2, depending on what that value is depends on what camera will be used for the windows field of view, and since this is called every cycle in the loop it can be switched back and forth.

The gamepad code is next for the player1 object whose update function is called to check if the controller is connected, if it is the debug log will receive a connected message. The next lines of code are a function to retrieve the state of the controller and series of if statements containing instructions on what to do in scene if the states are met. All values are set to 0.5 like states earlier on to reduce sensitivity in case some used remotes analogue sticks are ‘off kilter.’ It should be noted only the left analogue stick and left and right trigger are used so they are the only ones with coding. If the stick is up or down it will translate accordingly, left or right it will rotate accordingly. If the triggers are pulled the trigpulled function will be called to generate a laser model being fired with collision detection on it. All these functions turn the vibrate feature on momentarily before switching it off to signify the response to the users actions.

The sound switch is the next part in the main loop, which like the camera switch has a check function called soundtrack. Soundtrack checks a Boolean value that can be changed on pressing F3 or F4, depending on the state it will silence the theme or restart it (it only restarts if it was stopped.)

The starfield is called to be rendered next as well as the use of the use of the sun material. The scene lights are. The Death Star is rendered and the Enemy movements are calculated and moved. The player is rendered and then the Lasers movement is calculated and rendered.

The String variables are given values to be placed on the overly, the window sets the orthographic for 2D and the fonts and messages are pushed on-screen to make the overlay. The matrices are popped. The title of the game is shown with the amount of enemies and the current player rotation, the player energy is shown at the bottom of the screen, finally the game conditions are checked.

If the energy reaches 0, the theme will be stopped and the corresponding sound will be played in accordance with the storyboard, on the next press of the spacebar the window will close.

If the enemies are destroyed with energy left over, the theme will stop and the winning sound will play, the screen will then close.

If none of these conditions are met the buffers are swapped, tCount is updated with the elapsed time the input manager is cleared and the loop continues. When it stops the game is also set to end to prevent an error.

**References:**

<http://tf3dm.com/> (for models)

<https://lcmccauley.wordpress.com/2014/01/10/gamepadtutorial-part3/> (tutorial on how to make a gamepad.)

<http://stackoverflow.com/> (common error solutions and information on no architecture errors )

[www.cplusplus.com](http://www.cplusplus.com) (information on pointers and logic)

msdn.microsoft.com (information on key codes and Xinput commands for gamepad)