Anna Irene Margaritis S1905566

Computer Games (Software Development), Games Programming 2 coursework documentation

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

*Signature*.

Appendix

[1 Main & Game 3](#_Toc91245958)

[2 ScreenDisplay 3](#_Toc91245959)

[3 Mesh 4](#_Toc91245960)

[4 Shading 4](#_Toc91245961)

[5 Texturing 5](#_Toc91245962)

[6 Transform 5](#_Toc91245963)

[7 Camera 5](#_Toc91245964)

[8 Audio 5](#_Toc91245965)

# Main & Game

The link to the finished project can be found here: <https://github.com/Shirehii/GP2-CWK>

Main.cpp contains the single function that starts the entire program. Its primary purpose is to call Game’s constructor and StartGame() function and keep it running until the user closes the program.

  
*Figure 1: Code snippet showing the contents of Main.cpp*

The Game script contains the functions necessary for booting up and keeping the game running. StartGame() will call two other functions, InitializeSystems() and GameLoop(). Of those two, the former will run only once and will initialize the game’s display and camera, as well as load the game’s models, shaders, textures, and audio files. The latter will run a while loop based on the current GameState. If that GameState is ‘PLAYING’, then the while loop will not stop, effectively keeping the game running infinitely (or until the user manually closes it by pressing the escape key). When the while loop is running, the game will play background music, detect any collisions made between objects and play a sound effect if a collision is detected, as well as run two functions: ProcessUserInputs() and UpdateDisplay().

ProcessUserInputs() will simply get any events and process them using GLFW’s function glfwPollEvents(). For example, if the escape key is pressed then the GameState will change to ‘QUITTING’, ending the function with a ‘return;’ and stopping the while loop contained in GameLoop(), mentioned in the previous paragraph. Additionally, there’s functionality for registering any arrow key presses, which will call movement functions from the Camera struct. The camera can be zoomed in by pressing the plus and minus keys, which will call the ZoomCamera() function. Camera rotation can also be called from here by using the Camera’s MouseControls() method.

*Figure 2: Code snippet showing part of the ProcessUserInputs() function*

UpdateDisplay() will make changes to one of the game’s display buffers, starting with clearing it, and then activating the shader program using UseShader(). After textures are applied, each model is then drawn on the screen using the Mesh class’ Display() function, which can take a ‘counter’ variable as a parameter for positions, rotations and/or scale. The ‘counter’ variable counts up every time UpdateDisplay() is called. After everything else has ran, the buffers are swapped.

# ScreenDisplay

The ScreenDisplay class contains methods related to the game’s display, such as creating the game window or swapping buffers.

The ScreenDisplay() constructor creates a null pointer for the program window, and sets the default values for the window’s height and width. The ~ScreenDisplay() destructor will delete the game window, then quit SDL entirely.

  
*Figure 3: The ScreenDisplay constructor and destructor*

InitializeDisplay() is called once in the Game class’ InitializeSystems() function. InitializeDisplay() starts off by initializing SDL, GLFW and glew, then setting up the double buffer. Z-buffering is enabled, as well as face culling in order to stop faces that are not pointing to the camera from being displayed. After a new GLFW window is created, another function is called, named CheckForErrors(), which checks the GLFW window and glew to determine if they’ve been initialized correctly. Lastly, the display’s background color is set.

  
*Figure 4: The functions that enable Z-buffering and face culling*

ChangeBuffer() is used in the Game class for swapping the buffers. ClearDisplay() resets the display by taking in four parameters, for the red, green, blue, and alpha values, then using them when calling glClearColor(). Afterwards, the color and depth buffers are cleared as well.

Additionally, the ScreenDisplay class contains GetWidth() and GetHeight(), which are getters for the game window’s width and height, mostly used for initializing the camera.

# Mesh

The Mesh header file contains a Vertex struct, a BoundingSphere struct and the Mesh class. The Vertex struct is composed of a vector3 holding the position, a vector2 holding the texture coordinates and a vector3 holding the normal, as well as getters for those variables. The BoundingSphere is used for collision detection, and is composed of a vector3 position and a radius, along with the getters for them. Additionally, it includes an UpdateSphereData() method, used to update the BoundingSphere’s position and radius.

The Mesh class itself consists of functions for loading and initializing a model, and displaying it, as well as an UpdateSphereData() method, called from the Game class during runtime. LoadModel() takes in a file path as a parameter and uses it to load a model from file, then calls InitializeModel(). That function initializes the model by generating a vertex array and storing it in a vertex array object before binding it. Then, buffers are generated based on the array of data and moved to the GPU. After specifying to OpenGL how many attributes are in the object, it is then specified how OpenGL should read that data. Afterwards, after telling OpenGL what type of data the buffer is and passing the data, it is then moved to the GPU again. Lastly, the vertex array object is unbound.

With the model ready to be used, it can be displayed using the Display() method, which binds the vertex array, displays it using glDrawElements() and then unbinds it. It also updates the model transform’s values to simulate movement, as well as updating the bounding sphere’s position to ensure collision detection remains accurate.

   
*Figure 5: The Mesh class’ UpdateTransformValues() function, which is called in Display()*

# Shading

The Shading class handles anything related to shaders. It contains a constructor, destructor and functions for various purposes, such as creating, initializing and loading shaders. The constructor is empty, but the destructor when called will detach and delete the shaders from the program, before deleting the program itself.

InitializeShader() is called once from the Game class to prepare the shading program for use. This is done by first creating the shader program and then calling the CreateNewShader() function to create a new shader from file. Right after that, a for loop is ran to attach any created shaders to the shader program, before binding the attribute locations. The program is linked to create executables that will run on the GPU shaders and then validated. After checking for errors in linking and validation, the uniform is set up with the shader program. With the shading program initialized, it can be used simply by calling UseShader().

   
*Figure 6: Code snippet showing part of the InitializeShader() function*

# Texturing

The Texturing class contains functions for texturing the meshes used in the game. It does so through two methods, InitializeTexture() and UseTexture().

InitializeTexture() loads an image from file and stores that data into a variable called textureData, before checking if it’s null, to confirm that it loaded properly. Then, the texture is generated and stored in an array called textureHandler and is bound to edit its values. Its parameters are set to control the texture wrapping and linear filtering. After that, the texture is ready to be sent to the GPU, and the textureData is freed from the CPU. A counter called amountOfTextures counts up to assist with generating future textures and assigning them to their proper slot in the textureHandler array.

With the texture ready, it can be used through the function UseTexture(). It takes in an integer as a parameter to specify to the program which of the prepared textures have to be used in this case. After checking the number to ensure it falls within the specified index range, it binds the texture while specifying its type to be GL\_TEXTURE\_2D.

   
*Figure 7: The Texturing class’ UseTexture() function*

# Transform

Transform is a struct containing three vector3 variables representing a position, a rotation and a scale. These three variables can be accessed and used through the getters and setters available for them. The Transform constructor takes in three parameters as well, to represent the three transform variables, and sets them.

Additionally, GetModel() returns a matrix composed of a position matrix, a rotation matrix and a scale matrix multiplied, and is used in the Shading class’ UpdateTransform() function. The position, rotation and scale matrices are translated from the position, rotation and scale vector3 variables contained in the Transform struct.

# Camera

The Camera header file contains the struct for the game’s camera. The default constructor is empty, and camera initialization happens through the InitializeCamera() function. It takes in a position, a field of view, the aspect ratio, as well as near clip and far clip values as parameters. The function then sets the camera’s position, its forward and up vectors, and its projection matrix based on the parameters. The GetViewProjection() inline creates a projection matrix using glm::lookAt().

For camera movement, the functions MoveCameraHorizontally() and MoveCameraVertically() can be called from the Game class, and will move the camera one unit in any of the four directions. The values can be changed within the game’s code to move more units with each step. Similarly, the ZoomCamera() method moves the camera in the z-axis to simulate zooming in and out. Camera rotation is handled in the MouseControls() function, which additionally hides, unhides and centers the cursor as needed.

   
*Figure 8: Code snippet showing part of the MouseControls() function*

# Audio

The Audio class, consisting of both a header and a .cpp file, contains the functionality necessary for loading and playing audio in the game. The Audio constructor sets the default values for the audio rate, the channels, the buffers and also sets the audio format to ‘AUDIO\_S16SYS’. With the values set, it opens the audio mixer with those parameters before checking to make sure that it initialized properly. The destructor simply quits SDL.

There’s two functions, AddNewSound() and AddNewBackgroundMusic(), used for loading new sound effects and background music from a file, respectively. The former loads the file and stores the sound effect before adding it to the vector that contains all sound effects. The latter loads the music from file and directly stores it into a Mix\_Music variable.

As with adding new audio files, there’s two functions for using the loaded audio tracks. PlaySound() is used for sound effects, and takes in an integer to specify which of the loaded sound effects should be played. If the integer is not out of range, then it plays the sound effect through use of the Mix\_PlayChannel() function. Similarly, PlayBackgroundMusic() uses checks if using is not currently playing, in which case it will use Mix\_PlayMusic() to play the audio file added from AddNewBackgroundMusic.

   
*Figure 9: The Audio class’ PlaySound() and PlayBackgroundMusic() functions*