

Games Programming 2 Coursework

|  |  |
| --- | --- |
| **Module Name:** | Games Programming 2 |
| **Module Code:** | M3I626039 |
| **Module Leader:** | Bryan Young |
|  |  |
| **Student ID:** | S1920423 |
| **Student Name:** | Harry Calwell |

**By submitting this assignment, I agree to following statement:**

“Except where stated explicitly, all work in this report, project and accompanying source code, is my own original work and has not been submitted elsewhere in the fulfilment of the requirement of this or any other award”

Contents

[Camera & Transform 3](#_Toc27392680)

[Display 4](#_Toc27392681)

[Main Game 5](#_Toc27392682)

[Collision 7](#_Toc27392683)

[Mesh 8](#_Toc27392684)

[References 9](#_Toc27392685)

# Camera & Transform

The camera class consists of constant vector3 variables that set its position, field of view, aspect ratio, near clipping plane and far clipping plane.

Text

Description automatically generated

Figure 1-1: initCamera() method from Camera.h

The fov, aspectRatio, nearClippingPlane, and farClippingPlane variables are what are used to create the projection matrix which gives the perception of depth by appropriately scaling the values based on desired distance from the camera artefact.

Text

Description automatically generated

Figure 1-2: GetViewProjection() method from Camera.h

The projection matrix is returned based on the variables of the target it wants to look at which provides the user with the view in the game scene. This projection matrix works in tandem with the transform.h script which takes the 3D models physical properties and manipulates the geometric variables in relation to the projection matrix to give the final result of a 3D object with depth.

Text

Description automatically generated

Figure 1-3: GetModel() method from transform.h

The transform properties of the model are taken and returned as the relative position, rotation, and scale matrix.

Text

Description automatically generated

Figure 1-4: GetMVP() method from transform.h

Properties are then distorted by the camera.h projection matrix to give the 3D model an illusion of depth.

# Display

First the sdl Window is initialised to a null pointer and the desired screen width and height are assigned as floats to getters.

Text

Description automatically generated

Figure 2-1: Constructor and getters from Display.cpp

The createDisplay() method then initializes everything for the display. The “SetAttribute” variables declares how much memory we wish to assign to each display colour, red, green, and blue. Within the createDisplay() method there is also a SetAttribute for depth size which sets up a z-buffer to determine the render order in the scene. Then the OpenGL game window is created and returned to the sdlWindow address that was first initialized as a null pointer.

Text

Description automatically generated

Figure 2-2: Display is initialised

OpenGL context is then set based on the sdlWindow which allows the program to begin to render from the graphics card. When the OpenGL context is met, the function glewInit() completes the setup of the OpenGL display.

# MainGame

MainGame.cpp is responsible for executing the actual main scene in the game program using all the relevant methods and variables from the other scripts and headers created in the file. The header file for the main game contains all the methods and variables used by the MainGame.cpp script, which then runs the execution order as well as other commands that are used within the main scene of our program application.

Text

Description automatically generated

Figure -1: MainScene class from MainGame.h

Text

Description automatically generated

Figure 3-2: run() method from MainGame.cpp

# The run() method in the MainGame.cpp script then initiates the systems used in the game scene as well as the primary gameLoop() method which is responsible for calling the methods for drawing the meshes onto the screen, handling collision detection, and processing player input.

# Text Description automatically generated

Figure 3-3: initSystems() method from MainGame.cpp

# The meshes, textures, shaders, and sound effects are all loaded into the application from a local file using methods that are called from their relative .cpp scripts. Next the gameLoop() method is triggered which calls the drawTheMeshes() function found in the MainGame.cpp script.#

# Text Description automatically generated

Figure 3-4: Example of mesh being spawned from drawTheMeshes() method in MainGame.cpp

In the drawTheMeshes() method, the transform for each method is set, this sets the mesh’s position, rotation, and scale within the world scene. Then the shaders are set to the mesh and updated to match the projection matrix of the camera. The texture for the mesh is binded to it and then the spawn() method is triggered which draws the mesh into the game scene. The \_gameDisplay.switchBuffer() method handles the game display being sent to the GPU using buffers.

# Collision

Collision detection comes from the Mesh.cpp script. A mesh sphere is drawn around the original mesh and is updated to track the mesh’s position and size.  
Text

Description automatically generated  
Text

Description automatically generated

Figure -1: loadMesh() and updateSphereData() method from Mesh.cpp

By declaring the sphere meshes of each 3D model within the MainGame.cpp script, we can then set up the criteria for checking if collision occurs and produce our desired output.

Text

Description automatically generated

Figure 4-2: Collision detection method from MainGame.cpp

From MainGame.cpp

The method for collision detection is continuously checked by the main gameLoop() method. Collision is detected when the squared distance between the two sphere meshes is within a shared radius. When collision is returned true under this qualifier then the sound effect plays, and the command screen produces the output text.

# Mesh

The scripts for controlling the meshes that are presented on screen utilize buffers. Buffers are linear allocations of memories that are used for a number of purposes. The mesh.cpp script takes the data from the given 3D model and binds it to the relevant OpenGL context as a buffer.

Diagram

Description automatically generated

Figure -1: Buffer diagram (Young, 2021).

Our properties that are set as buffers are stored in a vertex array object to be processed and sent to the GPU to allow for visualization.

Text

Description automatically generated

Figure 5-2: Buffers being used in Mesh.cpp

Here the properties of the model such as positions, texCoordinates, normal and indices are binded as data in an array buffer. The data is then moved to the GPU, giving a starting address (pointer) of the data and then an end address on the GPU. This is how the processed model is displayed on the screen through the GPU as a mesh using OpenGL.

Text

Description automatically generated with medium confidence

Figure 5-3: Method for spawning mesh into game scene from Mesh.cpp script

The spawn() method then finalizes the process by drawing all the elements onto the screen. This method is called from the MainGame.cpp script when it is required for the meshes to be drawn into the game scene.

# References

YOUNG, BRYAN, 2021. OpenGL Graphics Pipeline & Buffers [lecture online]. 11 October. Available from: [https://eu.bbcollab.com/collab/ui/session/playback](https://eu.bbcollab.com/collab/ui/session/playback%20)