

Coursework

Games Programming 2

|  |  |
| --- | --- |
| **Module Name:** | Games Programming 2 |
| **Module Code:** | M3I625656 |
| **Module Leader:** | Bryan Young |
|  |  |
| **Student ID:** | S1714052 |
| **Student Name:** | Connor McSheffery |

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

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

Contents

[GameAudio Class 3](#_Toc29319750)

[GameCamera Class 4](#_Toc29319751)

[GameDisplay Class 5](#_Toc29319752)

[GameMain Class 6](#_Toc29319753)

[MeshModel Class 8](#_Toc29319754)

[ObjectLoader Class 9](#_Toc29319755)

[GameShader Class 10](#_Toc29319756)

[GameTexture Class 11](#_Toc29319757)

[Transform Class 12](#_Toc29319758)

# GameAudio Class

The Audio class deals with the sounds featured within the game, such as the background music and the collision sound effect.

The constructor opens an OpenAL device and then stores all the data within it. The destructor deletes all this data.

**isBigEndian() & convertToInt()**

To handle the raw data isBigEndian() and convertToInt() is used. These take the raw data and allow containment of them. isBigEndian() stores the data, and convertToInt() changes a char array to an int array so that the OpenAL can use these as it uses ints as parameters.

**loadWAV(const char\* fn, int& chan, int& samplerate, int& bps, int& size)**

This loads a WAV audio file so that it can be played using openAL.

**loadSound(const char\* filename)**

This method takes in a filename and then finds the file matching the name and will then decipher what type of data is being found and then stores this data to be used when playing sounds.

**playSound(unsigned int id, glm::vec3& pos)**

This method takes in the ID of the audio source that is to be played and a vector3 position which will be used to place the sound object so that the audio can be moved.

The unsigned int ID refers to the data found from the initial loadSound() method which will then be played in the position given.

The position given is used mainly for situational sound effects, such as when the stone knocks into the boat in my game.

**stopSound(unsigned int id)**

The stopSound() method is very self-explanatory. It takes in the sound that is to have its audio cut off. The program then stops the audio from continuing to play.

**setlistener(glm::vec3& pos, glm::vec3& camLookAt)**

This method takes in a vector3 which is used as the position to place the listener for the scene which is the camera within the scene, and if the camera was to move around, then the second vector3 would be the forward vector of the camera where it is looking at.

# GameCamera Class

This class deals with all the camera properties such as the initialization and the finding the properties of it.

**initCamera(const glm::vec3& pos, float fov, float aspect, float nearClip, float farClip)**

This method initializes the camera with the initial position of the camera, the field-of-view of the camera. The aspect of the window, which is the width / height of the screen. The method also takes in the near and far clip of the camera so it knows how far to render out.

**getPos()**

This method returns the position of the camera within the game for use.

**GetViewProjection()**

This method returns the view projection of the camera within the game.

# GameDisplay Class

This class deals with the general display of the screen.

**initDisplay()**

This method initializes the display of the game using SDL and GL. This method sets the attributes used for colours, and buffers. It also creates the window used and validates everything is working.

**swapBuffer()**

This method briefly swaps the OpenGL buffers for a window.

**clearDisplay(float r, float g, float b, float a)**

This method sets the background colour of the game based on the rgba values that are handed into the method.

**getWidth()**

**getHeight()**

These are getters that return the Width and Height.

# GameMain Class

This class deals with most of the game itself. It draws the game, takes in input and runs the game loop.

The constructor creates the display, meshes and the audio device within the game, whilst also setting the gamestate to Play.

**run()**

This method calls the initSystems() method to initialize everything and then calls the gameLoop() function to run the game.

**initSystems()**

This initializes everything within the game such as the calling the initialize function within the display class and also loads the sounds and models used within the game. The camera is also set up and placed into the game.

**gameLoop()**

This method calls each other method within the game for it to run. The runtime is found by using chrono::high\_resolution\_clock::now() which is stored into a new variable each frame and is subtracted from the previous frame time to find the frame time. This frame time is added to a float variable which then stores the whole games runtime which is used later as the score for the player.

The method then calls the processInput() method to check for input, then the drawGame() to update what is on the screen and then will call the collision() method three times to check if the player has hit any of the three rocks within the game.

The background music is then set to play for ambience.

**processInput()**

This method uses switch events to check for gamestates and for input. First, the method checks if the gamestate is to be set to EXIT and if it is, then the game will quit out.

The method then checks for user inputs, if the user presses the ‘a’ key or the left arrow key, then a variable called ‘left’ will be set to true which is used for moving the player left. If the user presses ‘d’ or the right key then the right variable will be set to true which is used for moving the player right.

It also checks if the player presses the ‘escape’ key and if so then the game will quit out.

**collision(glm::vec3 m1Pos, float m1Rad, glm::vec3 m2Pos, float m2Rad)**

This method deals with the collision detection within the game. The m1Pos and m1Rad stores the player’s position and radius, the m2Pos and m2Rad stores the position and radius of the rock being checked at that time.

A float variable is found using a formula given which finds the distance between the player and the rock.

If the float variable is a value less than the radius of the player plus the radius of the rock then a collision is detected, a sound effect is played to show the player it has been hit and then a lose Boolean value is set to true.

**playAudio(unsigned int Source, glm::vec3 pos)**

The playAudio() method plays the audio sources by calling the playSound() method within the audio class.

**drawGame()**

This method displays everything on the screen. The clearDisplay() method is first called from the display class which sets the background to blue as it is to show the boat is on water.

The shader is loaded as is the texture for the boat.

The player/boat is then drawn based on the movement variables at the time of called. If both movement variables are set to false then the boat will remain in the same spot as the previous frame, if the left variable is set to true then the boat will be moved to the left based on the previous position it was at. If the right variable is true then the same as the left variable will run however it will move to the right rather than left.

There is also validation within these so that the boat cannot move too far away.

If the game has just started then the rock obstacles will be placed above the screen and set to specific x-placements so they will not collide with each other. After they have been drawn, the method will check if the rock has passed the player and if so, then the rock will reset above the screen but set to a random x-placement.

The method then checks if the lose variable was set to true within the collision() method and if so, the player’s score is sent to the command log and the game will quit out.

# MeshModel Class

This class deals with the models that are featured within the game.

The constructor initializes the vertexArrayObject by setting the drawCount variable, which stores how much of the vertexArrayObject is wanted to be drawn, to NULL.

The deconstructor deletes the arrays that were created for the model.

**draw()**

This method binds the vertexArray and then draws the model using the glDrawElements function.

**init(Vertex\* vertices, unsigned int numVertices, unsigned int\* indices, unsigned int numIndices)**

This method creates the model base by inserting the indices and vertices needed to create it.

**loadModel(const std::string& filename)**

This method takes in a filename which will store the location to the model that is to be created and the initModel() method will then create that model.

**initModel(const IndexedModel& model)**

This method sets us the VertexArrayObject by generating a vertex array, binding it, creating the buffers and moving the buffers to the GPU. And then finally unbinding the VertexArrayObject.

**updateSphereData(glm::vec3 pos, float radius)**

This method sets the position and radius of the model using the values passed into it.

**getSpherePos()**

**getSphereRadius()**

These methods are getters that return the position and the radius of the model respectively.

# ObjectLoader Class

The constructor takes a filename, opens the file found with this name and reads in all the uvs, normals and vertices from it and stores it.

**CalcNormals()**

This method takes the normals and indices from the file and calculates the final normals from it and stores them.

**ToIndexedModel()**

This method checks for uvs and normals from the model and if there are not any then they are initialised. The method then creates models to generate normals upon it and separate the texture coordinates from it. If the model has no normals then the CalcNormals() function is called to give the model normals.

**FindLastVertexIndex(const std::vector<OBJIndex\*>& indexLookup, const OBJIndex\* currentIndex, const IndexedModel& result)**

**CreateOBJFace(const std::string& line)**

This method adds the UVs and then normals to the indices of the object to create the face of it.

**ParseOBJIndex(const std::string& token, bool\* hasUVs, bool\* hasNormals) & ParseOBJVec3(const std::string& line) & ParseOBJVec2(const std::string& line)**

These methods are very similar as they take in data and turn them into new objects such as and Index, vector3 value and vector2 value respectively.

**CompareOBJIndexPtr(const OBJIndex\* a, const OBJIndex\* b)**

This method returns a boolean of true if the vertexIndex of a is less than the vertexIndex of b, and false if the vertexIndex of b is greater than the vertexIndex of a.

**FindNextChar(unsigned int start, const char\* str, unsigned int length, char token)**

This method returns an integer value which shows the place of the character in the token variable in the str set of characters.

**ParseOBJIndexValue(const std::string& token, unsigned int start, unsigned int end) & ParseOBJFloatValue(const std::string& token, unsigned int start, unsigned int end)**

These methods take string values and convert them into Index and float values respectively.

**SplitString(const std::string &s, char delim)**

This method splits a string value by the delim character.

# GameShader Class

This class deals with the shader that is used to create the objects within the game.

The constructor creates a shader program, vertex shader and fragment shader. It then adds all of the shaders to the program and binds the attributes to it. It will then create executables and check for validity and errors.

The deconstructor detaches the shaders, deletes the shaders and then deletes the program.

**Bind()**

This method installs the program for rendering.

**Update(const Transform& transform, const Camera& camera)**

The Update() method creates a uniform matrix to be used within the shader.

**CreateShader(const std::string& text, unsigned int type)**

This method sends the source code to openGL which will compile the code from it and return a shader which will be used within the game.

**LoadShader(const std::string& filename)**

This method takes the filename of the shader, opens it and then loads the shader from inside the file.

**CheckShaderError(GLuint shader, GLuint flag, bool isProgram, const std::string& errorMessage)**

This method checks if there are any faults with the shader created and if so will send the error message to the console terminal.

# GameTexture Class

This class deals with the texturing of the objects within the game.

The constructor loads the image from a filename given in the method, and stores all the data from it, such as the width, height, data type, etc.

The deconstructor deletes the texture handler within the game.

**Bind(unsigned int unit)**

This method sets the active texture to be used and then binds it to the unit.

# Transform Class

The constructor sets the position, rotation and scale of the transform object.

**GetModel()**

This method creates a matrix based on the position, rotation and scale of the transform object by multiplying all the separate matrices together.

**GetPos() & GetRot() & GetScale()**

These methods are getters which return the position, rotation and scale of the transform object respectively.

**SetPos(glm::vec3& pos) & SetRot(glm::vec3& rot) & SetScale(glm::vec3& scale)**

These methods are setters which change the position, rotation and scale of the transform object respectively.