Games Programming 2 Documentation

Chun Ting Matthew Wan

S1625394

*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*.

*Chun Ting Matthew Wan*

**Table of Contents**

1. **Overview of Main Methods, Structures and Classes Page 3**

**1.1 TheGame Page 3**

**1.1.1 Variables Page 3**

**1.1.2 checkCollision() Page 3**

**1.1.3 processInput() Page 4**

**1.1.4 gameLoop() Page 4**

**1.1.5 initSystems() Page 4**

**1.1.6 updateTransform() Page 5**

**1.1.7 drawGame() Page 5**

**1.2 Shaders (Fragment and Vertex) Page 5**

**1.3 Mesh Page 5**

**1.3.1 Variables Page 5**

**1.3.2 shperePos() and radius() Page 6**

**1.3.3 updateTheSphere() Page 6**

**1.3.4 initModel() Page 6**

**1.3.5 Draw() Page 6**

**1.4 ModelTexture Page 7**

**1.4.1 init() Page 7**

**1.4.2 bind() Page 8**

**1.5 Transform Page 8**

**1.5.1 Variables Page 8**

**1.5.2 GetModel() Page 8**

**1.6 Camera Page 9**

**1.6.1 Variables Page 9**

**1.6.2 GetViewProjection Page 9**

**2.0 Sources Page 10**

**2.1 Fog Page 10**

**2.2 Lighting Page 10**

1. **Overview of Main Methods, Structure and Classes**
   1. **TheGame** 
      1. **Variables**

Floats to hold the X, Y and Z coordinates of the monkey head model

A float to hold a value for the counter

A Boolean to check for collisions

GameState “\_gameState” A new game state

Mesh “mesh1” a new mesh for candy cane

Mesh “mesh2” a new mesh for monkey head

Mesh “mesh3” a new mesh for peanut

ModelTexture “texture” a new texture for red and white texture

ModelTexture “texture2” a new texture for rock texture

ModelTexture “texture3” a new texture for peanut texture

Shader “shader” a new shader

Camera “myCamera” a new camera

**Methods used in TheGame:**

* + 1. **checkCollision()**

This method is used to calculate the collision using the spheres data. It passes in the following arguments which is then used for the calculation.

bool TheGame::checkCollision(glm::vec3 m1Pos, float m1Rad, glm::vec3 m2Pos, float m2Rad)

The following formula is then used to calculate the distance between the centre of both spheres and if spheres collide it will return true and send a debug log to the console.

float distanceSq = ((m2Pos.x - m1Pos.x) \* (m2Pos.x - m1Pos.x) + (m2Pos.y - m1Pos.y) \* (m2Pos.y - m1Pos.y) + (m2Pos.z - m1Pos.z) \* (m2Pos.z - m1Pos.z));

if (distanceSq < (m1Rad \* m2Rad) \* (m1Rad \* m2Rad))

{

cout << "Collision";

return true;

}

return false;

* + 1. **processInput()**

This method gets the keyboard input of certain keys so that when they are pressed the monkey head will move. It moves the monkey head by getting its transform position and moving it by so many units. The keyboard input to move the monkey head is W, A, S and D.

if (event.key.keysym.sym == SDLK\_d)

{

CoordX -= 0.3;

}

* + 1. **gameLoop()**

This method uses a while loop to check if the game state is not in Exit and will keep looping through the methods that are placed in this while loops.

while (\_gameState != GameState::EXIT)

{

processInput();

drawGame();

updateTransform();

checkCollision(\*mesh2.shperepos(), \*mesh2.radius(), \*mesh1.shperepos(), \*mesh1.radius());

}

* + 1. **initSystems()**

This method is called once at the start when the game is run. The display of the game is initialised.

\_gameDisplay.initDisplay();

It initialises the camera which sets the position, field of view, aspect ratio and the near and far clipping plane.

myCamera.initCamera(glm::vec3(0, 0, -40), 70.0f, (float)\_gameDisplay.getWidth()/\_gameDisplay.getHeight(), 0.01f, 1000.0f);

Each of the model objects, texture and shader is also loaded.

mesh1.loadModel("..\\res\\CandyCane.obj");

texture.init("..\\res\\redandwhite.jpg");

shader.init("..\\res\\shader"); //new shader

Fog is also enabled which is later used in the drawGame method.

glEnable(GL\_FOG);

* + 1. **updateTransform()**

This sets the position, rotation and the scale for each of the models. It also increases the counter float every update.

transform.SetPos(glm::vec3(0, 0, 0)); //position of object

transform.SetRot(glm::vec3(0.0, 0.0, sin(counter))); //rotation of object

transform.SetScale(glm::vec3(0.1, 0.1, 0.1)); //scale of object

* + 1. **drawGame()**

This clears the display and sets the colour of it. It then binds the shader and texture to each of the meshes that are drawn. It also swaps the buffers each time a new one is ready.

It also draws the fog into the game. It creates an array to store the colour of the fog.

GLfloat fogColour[] = { 0.5f, 0.5f, 0.5f, 1 };

glFogfv(GL\_FOG\_COLOR, fogColour);

It then uses a linear function that starts the fog at 0 and climbs linear until it reaches the 1 at the end distance. A start and end distance from the camera are also set. The start distance is when the fog will begin and grows in a linear fashion until it reaches the end distance from the camera.

glFogi(GL\_FOG\_MODE, GL\_LINEAR);

glFogf(GL\_FOG\_START, 1.0f);

glFogf(GL\_FOG\_END, 5.0f);

* 1. **Shaders (Fragment and Vertex)**

This handles the lambertian lighting in the shader fragment and vertex. The -vec3 holds the lighting direction. In Lambertian lighting, the direction is negative as it makes it look like the lighting is shot out the camera. It is also clamped to prevent the lighting from being too bright are going negative.

gl\_FragColor = texture2D(diffuse, texCoord0)

\* clamp(dot(-vec3(0,0,1), normal0), 0.0, 1.0);

* 1. **Mesh**
     1. **Variables**

A float to hold the radius of the sphere

A vector 3 to hold the position of the sphere

A vector 3 to hold the positions of vertices

A sphere “meshSphere” a new sphere

**Methods used in Mesh:**

* + 1. **shperepos() and radius()**

These methods return the position and radius held in the sphere.

sphere(glm::vec3& pos, float & radius)

{

this->position = pos;

this->radius = radius;

}

* + 1. **updateTheSphere()**

This method updates the spheres data by using the setters created to set the position and the radius.

meshSphere.Setpos(pos);

meshSphere.Setradius(radius);

* + 1. **initModel()**

This method initialises all the buffers that are needed for the models. We use glBindBuffer to let openGL what type of data the buffer is in the parameters.

glBindBuffer(GL\_ARRAY\_BUFFER, vertexArrayBuffers[POSITION\_VERTEXBUFFER]);

We then use glBufferData move the buffer to the GPU and in the parameter, we put the type of buffer, size it will be and the target location to store the buffers.

glBufferData(GL\_ARRAY\_BUFFER, model.positions.size() \* sizeof(model.positions[0]), &model.positions[0], GL\_STATIC\_DRAW);

It also updates initialises a sphere when the model is initialises.

* + 1. **Draw()**

In the draw method we first bind the vertex array using

glBindVertexArray(vertexArrayObject);

We can draw the array using glDrawArrays and in the parameters we tell openGL that we want to draw using triangles, where we want to start and when we want to end.

glDrawElements(GL\_TRIANGLES, drawCount, GL\_UNSIGNED\_INT, 0);

We then have to unbind the vertex array using

glBindVertexArray(0);

* 1. **ModelTexture**

**Methods used in ModelTextures**

* + 1. **init()**

We store the width, height and number of components which we get from stbi.

int width, height, numComponents;

We also generate a texture and the address of the texture using

glGenTextures(1, &textureHandler);

Once we have the space for a texture, we then bind the texture using

glBindTexture(GL\_TEXTURE\_2D, textureHandler);

We then control the texture wrapping using glTexParameteri. GL\_TEXTURE\_WRAP\_S is the texture width whereas GL\_TEXTURE\_WRAP\_T is the texture height.

The GL\_REPEAT parameter will repeat the texture at the end of each time a texture is placed. E.g if the texture is ten pixels wide on the eleventh pixel it will repeat the texture.

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

We then use glTexParameterf which allows us to determine the power handles when reducing or increasing the texture. In this take we use GL\_LINEAR to which means the texture will linearly interpolate between the existing pixels.

GL\_TEXTURE\_MIN\_FILTER is when the texture uses less pixels than its actual resolution whereas GL\_TEXTURE\_MAG\_FILTER is when it uses more pixels than the actual texture resolution.

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

We then send the texture to the GPU with the following parameters.

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA, width, height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, imageData);

* + 1. **Bind()**

We can bind a texture using

glActiveTexture(GL\_TEXTURE0 + unit);

glBindTexture(GL\_TEXTURE\_2D, textureHandler);

It will set the active texture unit and then we can bind the texture to a unit.

We also use assert to make sure the unit is between 0 and 31 as openGL can only hold up to 31 textures.

assert(unit >= 0 && unit <= 31);

**1.5 Transform**

**1.5.1 Variables**

A vector 3 to hold the position

A vector 3 to hold the rotation

A vector 3 to hold the scaling

**1.5.2 GetModel()**

This method takes the position, rotation and scale and converts it into a matrix. It is also set as a constant as this method doesn’t change the position, rotation or scale.

In this method we generate a 4x4 matrix for the position and scale.

glm::mat4 posMat = glm::translate(pos);

glm::mat4 scaleMat = glm::scale(scale);

For the rotation matrix we have to create a separate matrix for each of the axis. This is because glm does not have a function for converting the angled rotation into a single matrix.

glm::mat4 rotX = glm::rotate(rot.x, glm::vec3(1.0, 0.0, 0.0));

glm::mat4 rotY = glm::rotate(rot.y, glm::vec3(0.0, 1.0, 0.0));

glm::mat4 rotZ = glm::rotate(rot.z, glm::vec3(0.0, 0.0, 1.0));

We then get a single rotation matrix by multiplying all the x, y and z matrix together.

glm::mat4 rotMat = rotX \* rotY \* rotZ;

Once we have the single rotation matrix we can then return the position matrix by multiplying the position, rotation and scaling matrix together.

return posMat \* rotMat \* scaleMat;

**1.6** **Camera**

**1.6.1 Variables**

A matrix for the projection matrix

A vector 3 for the position of the camera

A vector 3 for the forward direction

A vector 3 for the up direction

**1.6.2 GetViewProjection()**

In this method we get the view projection matrix by using glm::mat4 GetViewProjection(). Inside the parameter we put in the position we want to be looking at and the up direction. We then have to multiply this by the perspective matrix to return the view projection matrix.

inline glm::mat4 GetViewProjection() const

{

return projection \* glm::lookAt(pos, pos + forward, up);

}

1. **Sources**

**2.1 Fog**

The fog used was based on this fog tutorial available at this address:

<https://www.youtube.com/watch?v=WH7hXxiNG6g>

**2.2 Lighting**

The lighting was added based on this guide at this address:

<https://www.youtube.com/watch?v=NS980twY1ZE&t=1084s>