Games Programming 2

Coursework Document

William Donald

S1632091

*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*.

Contents

[1.0 Main Game 3](#_Toc532835290)

[1.1 Variables 3](#_Toc532835291)

[1.2 Methods 4](#_Toc532835292)

[2.0 Mesh 8](#_Toc532835293)

[2.1 Variables 8](#_Toc532835294)

[2.2 Methods 9](#_Toc532835295)

[3.0 Transform 12](#_Toc532835296)

[3.1 Variables 12](#_Toc532835297)

[3.2 Methods 12](#_Toc532835298)

# 1.0 Main Game

## 1.1 Variables

Display \_gameDisplay;

GameState \_gameState;

Mesh meshShip;

Mesh meshRing;

Mesh meshRing2;

GameCamera myCamera;

Texture texture;

Texture texture2;

Texture texture3;

Shader shader;

int score;

int r1Direct;

int r2Direct;

Transform transform2 = Transform(glm::vec3(-30, -10, 90),

glm::vec3(90, 180, 0),

glm::vec3(0.2, 0.2, 0.2));

Transform transform3 = Transform(glm::vec3(30, -10, 90),

glm::vec3(90, 270, 0),

glm::vec3(0.03, 0.03, 0.03));

The \_gameDisplay variable will create the screen and the \_gameState variable will stare the current state of the game. The three Mesh variables meshShip, meshRing and meshRing2 will be used to create the three objects that will be in the game. myCamera is the camera that will be used to see the scene. The three textures texture, texture2 and texture3 will be used to store the textures for the models and apply them to their required model. The shader will be used to run the graphics pipeline to the GPU. The integer score will be used to keep track of the players score which will accumulate through each collision between the ship object and the two ring objects. The two integers r1Direct and r2Direct will be used when moving the rings along the X-axis. The Transforms transform2 and transform3 will be used to move, rotate and scale the two ring objects.

## 1.2 Methods

MainGame();

MainGame::MainGame()

{

\_gameState = GameState::PLAY;

Display\* \_gameDisplay = new Display();

Mesh\* meshRing();

Mesh\* meshShip();

Mesh\* meshRing2();

Texture\* texture();

Shader\* shader();

}

The MainGame constructor method which provides the initial value to \_gameState setting it to equal the state Play. MainGame also initialises \_gameDisplay, meshRing, meshring2, meshShip, texture and shader.

void run();

void MainGame::run()

{

score = 0;

r1Direct = 10;

r2Direct = -10;

transform.setTransform();

initSystems();

gameLoop();

}

The run method initialises score r1Direct and r2Direct setting them to 0, 10 and -10 respectively. transform.setTransform, initSystems and gameLoop are also called by run.

bool checkCollision(glm::vec3 m1Pos, float m1Rad, glm::vec3 m2Pos, float m2Rad, int ringNo);

bool MainGame::checkCollision(glm::vec3 m1Pos, float m1Rad, glm::vec3 m2Pos, float m2Rad, int ringNo)

{

float distSq = (

(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 (distSq < (m1Rad \* m2Rad) \* (m1Rad \* m2Rad))

{

if (ringNo == 1)

{

score++;

cout << "Score: " << score << endl;

transform2 = Transform(glm::vec3(transform2.GetPos()->x + r1Direct, -10,

500), glm::vec3(90, 180, 0), glm::vec3(0.2, 0.2, 0.2));

}

if (ringNo == 2)

{…}

return true;

}

return false;

}

The checkCollisions method is used to determine whether or not two objects have collided with each other. This method takes in the positions and radii of the two objects that are being checked as well as a ring number to distinguish between the two rings being checked. The float distSq is used to determine the distance between the two objects being checked utilising the formula Distance d. distSq is then checked to see if it is less than (radius1 multiplied by radius2)2 . If so then the two objects have collided. And then the ringNo is checked if it is 1 then the ship has collided with meshRing and if ringNo is 2 then the ship collided with meshRing2. The code following the check of ringNo is similar; the score is increased by 1 for meshRing and 2 for meshRing 2, a message to console is displayed and the ring the ship collided with is then moved to a new position. This method returns true if a collision occurred otherwise it returns false.

void initSystems();

void MainGame::initSystems()

{

\_gameDisplay.initDisplay();

meshShip.loadModel("..\\res\\Ship.obj");

texture.init("..\\res\\Ship.png");

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

…

…

myCamera.initCamera(glm::vec3(0, 0, -5), 70.0f,

(float)\_gameDisplay.getWidth()/\_gameDisplay.getHeight(), 0.01f, 1000.0f);

}

The initSystems Method initialises the \_gameDisplay the three models and myCamera. \_gameDisplay.initDisplay is called to initialise the game display. The three models are initialised through the same process in the case of the ship meshShip.loadModel sets the mesh to a model that is loaded from file then the texture relating to the ship loads the appropriate texture from file and finally a new shader is initialised. This is repeated for meshRing using texture2 and MeshRing2 using texture3. myCamera.initCamera is then called to initialise the camera.

void gameLoop();

void MainGame::gameLoop()

{

while (\_gameState != GameState::EXIT)

{

processInput(); drawGame();

checkCollision(\*meshShip.SpherePos(), \*meshShip.sphereRadius(),

\*meshRing.SpherePos(), \*meshRing.sphereRadius(), 1);

checkCollision(\*meshShip.SpherePos(), \*meshShip.sphereRadius(), \*meshRing2.SpherePos(), \*meshRing2.sphereRadius(), 2);

if (transform2.GetPos()-> z < -10)

{

transform2 = Transform(glm::vec3(transform2.GetPos()->x + r1Direct, -10,

500), glm::vec3(90, 180, 0), glm::vec3(0.2, 0.2, 0.2));

}

else

{

transform2 = Transform(glm::vec3(transform2.GetPos()->x, -10, transform2.GetPos()->z - 0.5), glm::vec3(90, 180, 0), glm::vec3(0.2, 0.2, 0.2));

}

…

if (transform2.GetPos()->x > 30)

{

r1Direct = -10;

}

if (transform2.GetPos()->x < -30)

{

r1Direct = 10;

}

…

…

}

}

The gameLoop method runs each frame until the \_gameState is EXIT. This method calls the procesInput and drawGame methods each loop allowing for constant user input as well as allowing the display to be updated each frame. The checkCollision methods checking if ship collides with either of the ring objects is also called from this method. The remaining if statements are used to manipulate the positions of the ring objects. The first set of if statements checks if the Z position of transform2 or transform3 becomes less than -10 in which case they are moved further back in the Z-axis and slightly in the X-axis by adding the value of either r1Direct for transform2 or r2Direct for transform3. If those if statements conditions are not met then the two transforms are moved along the Z-axis by -0.5 for transform2 and -1 for transform3. The final set of if statements are used to keep the rings within the X- axis boundary of -30 and 30 by inverting r1Direct and r2Direct when they reach the boundary.

void processInput();

void MainGame::processInput()

{

SDL\_Event evnt;

while (SDL\_PollEvent(&evnt))

{

switch (evnt.type)

{

case SDL\_QUIT:

\_gameState = GameState::EXIT;

break;

case SDL\_KEYDOWN:

switch (evnt.key.keysym.sym)

{

case SDLK\_LEFT:

transform.transformObj(-0.5, 1);

break;

case SDLK\_RIGHT:

transform.transformObj(0.5, 3);

break;

}

break;

case SDL\_KEYUP:

switch (evnt.key.keysym.sym)

{

case SDLK\_LEFT:

transform.resetObjRot(0.5);

break;

case SDLK\_RIGHT:

transform.resetObjRot(-0.5);

break;

}

break;

}

}

}

The processInput method determines if the player inputs through the keyboard and checks if the player closes the game. SDL events are used to check if the player quits the game or presses the left or right arrow keys. If the event is quit then the game state is changed to EXIT. If the SDL\_KEYDOWN event is called then if the arrow key was pressed then transform.transformObj is called with values that are passed in will depending on if it was left or right arrow that was pressed. The SDL\_KEYUP event is called then the transform.resetObj is called with its passed in valued depending on the arrow key pressed.

void drawGame();

void MainGame::drawGame()

{

\_gameDisplay.clearDisplay(0.0f, 0.0f, 0.0f, 1.0f);

shader.Bind();

shader.Update(transform2, myCamera);

texture2.Bind(0);

meshRing.draw();

meshRing.UpdateSphere(\*transform2.GetPos() , 4.0f );

…

…

glEnableClientState(GL\_COLOR\_ARRAY);

glEnd();

\_gameDisplay.swapBuffer();

}

The draw game method is used to draw the models to the screen each time it is called. Firstly the display is cleared in order for the new models to be drawn in the place of the old ones. Then the shader is bound and then updated with the relevant transform and myCamera. The texture of the current model is then bound and the mesh is then drawn and the collision sphere for that model is updated. This process is done for each model that has to be drawn. Lastly glEnableClientState, glEnd and \_gameDisplay.swapBuffer are called.

# 2.0 Mesh

## 2.1 Variables

struct Vertex

{

glm::vec3 pos;

glm::vec2 texCoord; glm::vec3 normal;

};

pos is a Vector 3 that stores the positions of the vertices, tecCoord stores the texture coordinates for the vertices and normal stores the coordinates for the normals of the vertices.

struct Sphere

{

glm::vec3 pos;

float radius;

};

Pos is used to store the position of the sphere that is used for calculating the collisions between the objects. Radius uses to store the radius of the radius of the collision sphere.

class Mesh

{

enum

{

POSITION\_VERTEXBUFFER,

TEXCOORD\_VB,

NORMAL\_VB,

INDEX\_VB,

NUM\_BUFFERS

};

GLuint vertexArrayObject;

GLuint vertexArrayBuffers[NUM\_BUFFERS];

unsigned int drawCount;

Sphere meshSphere;

Transform transform;

};

The enumerator is used during the creation of the buffers. vertexArrayObject is used to store the object that will be drawn and vertexArrayBuffers is used to create an array of buffers. drawCount is used to determine how much of vertexArrayObject that will be drawn. meshSphere will be used to detect collisions and transform will be used to manipulate the position, rotation and scale of the ship model.

## 2.2 Methods

struct Vertex

{

Vertex(const glm::vec3& pos, const glm::vec2& texCoord)

{

this->pos = pos;

this->texCoord = texCoord;

this->normal = normal;

}

glm::vec3\* GetPos() { return &pos; }

glm::vec2\* GetTexCoord() { return &texCoord; }

glm::vec3\* GetNormal() { return &normal; }

};

The vertex constructor is used to set the values of pos, texCoord and normal of the vertex struct using the values passed in. the three getters return the values of pos, texCoord and normal of the vertices.

struct Sphere

{

Sphere();

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

{

this->pos = pos;

this->radius = radius;

}

inline glm::vec3\* GetPos() { return &pos; }

inline float\* GetRadius() { return &radius; }

inline void SetPos(glm::vec3& pos) { this->pos = pos; } inline void SetRadius(float& radius) { this->radius = radius; }

};

The sphere constructors are used to initialise each sphere with one setting the values of pos and radius to the values that are passed in. the getters are used to retrieve the values of pos and radius whereas the setters are used to change them when they need to be updated.

Mesh();

Mesh::Mesh()

{

drawCount = NULL;

}

The Mesh constructor sets the draw count to null when called.

~Mesh();

Mesh::~Mesh()

{

glDeleteVertexArrays(1, &vertexArrayObject);

}

The Mesh destructor deletes the arrays when it is called.

void draw();

void Mesh::draw()

{

glBindVertexArray(vertexArrayObject);

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

glBindVertexArray(0);

}

The draw method will draw the model to the screen when called. It will first bint the vertex array object then draw the elements required and lastly the vertex array object will be unbound.

void initModel(const IndexedModel& model);

void Mesh::initModel(const IndexedModel& model)

{

drawCount = model.indices.size();

glGenVertexArrays(1, &vertexArrayObject); glBindVertexArray(vertexArrayObject);

glGenBuffers(NUM\_BUFFERS, vertexArrayBuffers);

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

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

&model.positions[0], GL\_STATIC\_DRAW);

glEnableVertexAttribArray(0);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

…

…

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vertexArrayBuffers[INDEX\_VB]);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, model.indices.size() \*

sizeof(model.indices[0]), &model.indices[0], GL\_STATIC\_DRAW);

glBindVertexArray(0);

SpherePos(); sphereRadius();

}

The initModel method is used to initialise the mesh and the buffers that will be created. Firstly drawCount is set the amount of indices the model contains. Then a vertex array is created and stored in the vertexArrayObject then the VAO is bound. Three buffers are then generated using the same method but with altered values: position, texCoord and normal. The buffer is first bound followed by the data being moved to the GPU. The attributes are then filled lastly an array of attribute data is created. A buffer for the indices is bound and the data moved to the GPU. Once that has been completed the VAO is unbound. Spherepos and Sphere radius are then called to initialise the position and radius of the models collision sphere.

void loadModel(const std::string& filename);

void Mesh::loadModel(const std::string& filename)

{

IndexedModel model = OBJModel(filename).ToIndexedModel();

initModel(model);

}

The loadModel method creates an IndexedModel named model which is set to a model in the file using the filename passed into the method. The initModel method is then called with model being passed in.

glm::vec3\* SpherePos(){return this -> meshSphere.GetPos();} float\* sphereRadius() { return this -> meshSphere.GetRadius(); }

spherePos and sphereRadius are used to obtain the position and radius of the collision sphere by returning the output of meshShere.GetPos and meshSphere.GetRadius respectively.

void UpdateSphere(glm::vec3& pos, float radius);

void Mesh::UpdateSphere(glm::vec3 & pos, float radius)

{

meshSphere.SetPos(pos);

meshSphere.SetRadius(radius);

}

The UpdateSphere method is used to alter the position and radius of the collision sphere when it is called. meshSphere.SetPos and meshSphere.SetRadius are called to changes the values of position and radius for the collision sphere of the required model. The values pos and radius that are passed in are used for the alterations.

# 3.0 Transform

## 3.1 Variables

glm::vec3 pos;

glm::vec3 rot;

glm::vec3 scale;

int currentDirection;

float counter;

float decreaseCounter;

bool canGoBack;

The three vector3 variables are used to store the position, rotation and scale of the object. The integer currentDirection is used to identify the direction the ship is moving. The floats counter and decrease counter are used for the movement of the ship and the rotation while moving. The Boolean canGoBack is used when moveing the ship away from the border.

## 3.2 Methods

Transform(const glm::vec3& pos = glm::vec3(), const glm::vec3& rot = glm::vec3(), const glm::vec3& scale = glm::vec3(1.0f, 1.0f, 1.0f))

{

this->pos = pos;

this->rot = rot;

this->scale = scale;

}

The transform constructor is used to set the values of pos, rot and scale using the values that are passed into the method.

inline glm::mat4 GetModel() const

{

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

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

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));

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

return posMat \* rotMat \* scaleMat;

}

The GetModel method is used to create a 4 dimensional matrix that will be used to move models in the scene. The movement matrix is created by returning the result of three matrices multiplied together. These matrices are a position matrix, a rotation matrix and a scalar matrix.

inline glm::vec3\* GetPos() { return &pos; }

inline glm::vec3\* GetRot() { return &rot; }

inline glm::vec3\* GetScale() { return &scale; }

inline void SetPos(glm::vec3& pos) { this->pos = pos; }

inline void SetRot(glm::vec3& rot) { this->rot = rot; }

inline void SetScale(glm::vec3& scale) { this->scale = scale; }

The three getter methods return the values of the transforms position, rotation and scale whereas the three setters are used to alter the three values with the required new values.

void transformObj(float inputPos, int direction);

void Transform::transformObj(float inputPos, int direction)

{

if (direction != currentDirection)

{

decreaseCounter = counter;

currentDirection = direction;

counter = 0;

canGoBack = true;

}

else

{

canGoBack = false;

}

if (transform.pos.x > -25 && transform.pos.x < 25)

{

if (counter > 0)

{

transform.SetPos(glm::vec3(inputPos \* 25 \* decreaseCounter, -15.0, 32.5));

decreaseCounter -= 0.1;

}

}

else if(canGoBack)

{

transform.SetPos(glm::vec3(transform.pos.x - inputPos, -15.0, 32.5));

}

if (transform.rot.z >-0.5 && transform.rot.z < 0.5)

{

transform.SetRot(glm::vec3(0.0, 0.0, inputPos\*counter));

counter = counter + 0.1;

}

transform.SetScale(glm::vec3(sinf(0.5), sinf(0.5), sinf(0.5)));

}

The transformObj method is used to move and rotate the ship object. The first if statement is used to determine if the object is will be moving in the same direction. If not then; decreaseCounter will be set to the current value of counter, currentDirection will be set to direction, counter will be set to zero and canGoBack will be set to true. If the direction is the same as currentDirection then canGoBack will be set to false. The second if statement is used to constrain the ship to between the X-axis values of -25 and 25. If the ship is within these bounds, then a new if counter will check if counter is greater than 0 if so then the ship will be moved in the X-axis by inputPos multiplied by 25 multiplied by decreaseCounter and then 0.1 will be subtracted from decreaseCounter. If the ship is out of bounds then a check will occur to see if canGoBack is true, if so the ship will be moved by – inputPos. The next if statement will check if the objects Z-axis rotation is within the bounds of -0.5 and 0.5, if this is the case then the ship will be rotates in the Z-axis by inputPos multiplied by counter. Counter is then increased by 0.1. finally the scale of the ship is set.

void resetObjRot(float inputPos);

void Transform::resetObjRot(float inputPos)

{

if (transform.rot.z != 0)

{

transform.SetRot(glm::vec3(0.0, 0.0, 0.0));

}

}

The resetObjRot method resets the rotation of the object if the rotation of the ship in the

Z-axis does not equal zero.

void setTransform();

void Transform::setTransform()

{

transform.SetPos(glm::vec3(0.0, -15.0, 32.5));

transform.SetRot(glm::vec3(0.0, 0.0, 0.0));

transform.SetScale(glm::vec3(sinf(0.5), sinf(0.5), sinf(0.5)));

}

The setTransform method is used to initialise the ship object to its required starting position, rotation and scale.