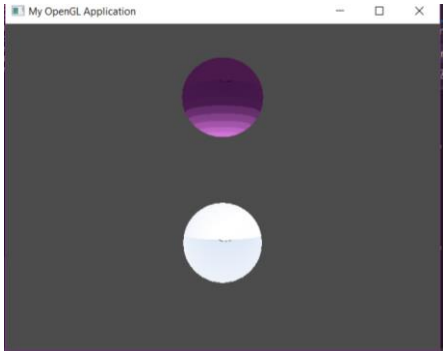
3D GRAPHICS AND ANIMATION COURSEWORK REPORT

Stuart Marples

F20GA
Stefano Padilla

H00156296
SM861

Scene graph



The scene consists of a FPS styled camera, a light and a spherical ball. In this scene the ball will bounce up and down continuously while changing speed and size throughout the animation. The user can move the light while also being able to control the camera to be able to move around the scene.

To create this scene, I exported a sphere object from blender and read that object into the code while also reading in the various shaders for the light and the model. I used the function readObj and readShader which goes through a file such as vs_model.glsl and reads in the data from the files and saves it to the appropriate variables for use later in the program.

```
string vs_text = readShader("vs_model.glsl");
string fs_text = readShader("fs_model.glsl");
```

The bouncing ball and light would remain the same size while loading a texture and material. For the texture I created a "sphere_map.ktx" file which loads an image of a sky and cloud to the sphere. Unfortunately, it is not very clear on either of the spheres due to the materials but if you look at the light you can see a black curve which is the base of the cloud and the sky colour on the bottom half of the light.

```
istringstream rawDataStream(rawData);
string dataLine;
int linesDone = 0;
while (std::getline(rawDataStream, dataLine)) {
    //reads in data for vertices
    if (dataLine.find("v ") != string::npos) {
        glm::vec3 vertex;

        int foundStart = dataLine.find(" ");  int foundEnd = dataLine.find(" ", foundStart+1);
        vertex.x = stof(dataLine.substr(foundStart, foundEnd - foundStart));
```

Model and Transformation

The only object that was used was a sphere that was created in blender. This sphere got imported into the code as the light source and the ball that would bounce. The only transformation that occurs throughout the program is when the ball is close to reaching the bottom of the bounce. At this point the ball slowly expands across its x axis, which is decreasing the xscale, and shrinks across its y axis, which is increasing the yscale, until it hits the bottom. Then it slowly expands the y axis and shrinks the x axis to take the ball back to its original size when it begins to move back up.

```
//if the ball is close to the ground, stretch and squash the ball
if (bheight < 15) {
    modelScale[0] = glm::vec3(xscale, yscale, 1.0f);
    yscale = yscale + 0.02f;
    xscale = xscale - 0.03f;
}
```

Lights and Materials

To create the material, I changed a lot of the values available in blender including items like specular light being changed to CookTorr and the diffuse to lambert. For the light I created some ambient, diffuse and specular lighting to light up the scene. I would find the direction and distance for the diffuse and specular light which is then utilised in the final calculation which figures out the colour of the light by using a long but not too complex mathematical function. The variables are then saved in the source code for the ability be to changed easily and allowed me to change the scene to how I wanted it to look more easily.

```
uniform vec4 ia;      // Ambient colour
uniform float ka;           // Ambient constant
uniform vec4 id;      // diffuse colour
uniform float kd;           // Diffuse constant
uniform vec4 is;      // specular colour
uniform float ks;           // specular constant
uniform float shininess;// shininess constant


void main(void)
{
// Diffuse
     vec4 lightDir = normalize(lightPosition - fs_in.fragPos);
     float diff = max(dot(normalize(fs_in.normals), lightDir),
0.0);
     vec4 diffuse = diff * id;

     // Specular
     vec4 viewDir = normalize(viewPosition - fs_in.fragPos);
     vec4 reflectDir = reflect(-lightDir,
normalize(fs_in.normals));
     float spec = pow(max(dot(viewDir, reflectDir), 0.0),
shininess);

     // Light
     color = vec4(ka * ia.rgb + kd * id.rgb * diffuse.rgb + ks *
is.rgb * spec, 1.0) * texture(tex, fs_in.tc);
```

Interactions

```
// calculate movement
GLfloat cameraSpeed = 1.0f * deltaTime;
if (keyStatus[GLFW_KEY_W]) cameraPosition += cameraSpeed * cameraFront;
if (keyStatus[GLFW_KEY_S]) cameraPosition -= cameraSpeed * cameraFront;
if (keyStatus[GLFW_KEY_A]) cameraPosition -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
if (keyStatus[GLFW_KEY_D]) cameraPosition += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;

// moving light displacement x y z
if (keyStatus[GLFW_KEY_LEFT])          lightDisp.x -= 0.05f;
if (keyStatus[GLFW_KEY_RIGHT])         lightDisp.x += 0.05f;
if (keyStatus[GLFW_KEY_UP])            lightDisp.y += 0.05f;
if (keyStatus[GLFW_KEY_DOWN])          lightDisp.y -= 0.05f;
if (keyStatus[GLFW_KEY_1])             lightDisp.z += 0.05f;
if (keyStatus[GLFW_KEY_2])             lightDisp.z -= 0.05f;
```

The user can interact with the scene by moving the camera by using the w, a, s and d keys to move and use the mouse to aim the camera. The user can also move the light in any direction they want by using the arrow keys to move along the x and y axis and the 1 and 2 keys to move across the z axis.

To move the mouse, I would read in the x and y values of where the mouse is and work out the difference between where the mouse is and where is just was to find the offset. With this we can apply a sensitivity to it and add this value to the yaw and the pitch to get the new values for the x and y axis. I then use cos and sin with the yaw and pitch to figure out the angles of the movement before finally moving the camera in the same direction as the user's mouse.

```
int mouseX = static_cast<int>(x);
int mouseY = static_cast<int>(y);

if (firstMouse){
    lastX = (GLfloat) mouseX; lastY = (GLfloat)mouseY; firstMouse = false;
}

GLfloat xoffset = mouseX - lastX;
GLfloat yoffset = lastY - mouseY; // Reversed
lastX = (GLfloat)mouseX; lastY = (GLfloat)mouseY;

GLfloat sensitivity = 0.05f;
xoffset *= sensitivity; yoffset *= sensitivity;

yaw += xoffset; pitch += yoffset;

// check for pitch out of bounds otherwise screen gets flipped
if (pitch > 89.0f) pitch = 89.0f;
if (pitch < -89.0f) pitch = -89.0f;

glm::vec3 front;
front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
front.y = sin(glm::radians(pitch));
front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
cameraFront = glm::normalize(front);
```

Animation

The animation that was utilised in this project was to make the ball bounce, while slowing at the top of its bounce and also squeezing and stretching at the bottom of the bounce. To create this, I added some global variables to control the height, speed, scaling and a Boolean value to determine if the ball is going up or down and then only edited the “update” function to add these effects to the ball.

```
//if ball is going up, move the ball upwards
if(down == false) {
    modelPositions[0] = glm::vec3(0.0f, by, 0.0f);
    //if the ball is close to the ground, stretch and squash the ball
    if (bheight < 15) {
        modelScale[0] = glm::vec3(xscale, yscale, 1.0f);
        yscale = yscale + 0.02f;
        xscale = xscale - 0.03f;
    }
    by = by + bspeed;
    bheight++;
    if (bheight > (maxHeight * 0.1)) {
        bspeed = bspeed - 0.0002f;
    }
}
```

If the ball was travelling upwards I would increase the y axis of the ball. Once it starts travelling upwards I would start to decrease the speed of the ascent until it came to a gentle stop. Making the ball go back down was the same method but in reverse so the ball would slowly speed up while falling until it hit its minimum point. When it gets close to the bottom the ball would stretch out across the x axis while also squeezing at the y axis to appear more realistic.