M3I622944

“The Monkey Solar System”

Graphics Programming Coursework

Julio Prado Muñoz

S1840230

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.

Julio Prado JPRADO200@caledonian.ac.uk

Contents

Figures

1 Introduction

The aim of this report is to cover the process of development and implementation of a 3D visual effect using GLSL shaders. The effect is deployed along with other two shader programs worked during the lab sessions: a Explosion shader and a Reflection shader. These three visual effects are displayed within a skybox on the scene at the same time.

I would like to emphasize that all the work included in this coursework has been carried out by me and no external sources or models has been checked so possible better solutions could have been already approached.

1.1 Objective

The new shaders created attemps to draw a virtual solar system from a single object i.e. using a mesh containing a monkey to fully recreate a Sun with planets.

Figure 1: Solar System Schema

The picture shown on the left reflects the idea behind the model to create. There must be a Sun able to light the planets in its surrounding area. Furthermore, the planets have to go around the Sun and keep theirself in orbit. To add more realism to the model, planets should go in different wise.

To simplify the code, the rotation movement of the planets and the Sun will follow the same pattern specified by the rotation parameters set on the mesh object before drawing.

Of course, the whole system has to "float" in the space and move itself through the scene without losing the consistency of the model.

1.2 Mathematical Approach

To achieve the results shown in the section above it is clear that we should performace some transformations in the 3D space, particularly mathematical rotation movements.

First, let us remember what a rotation is. Rotation is the turning of a figure or object around a fixed point, or in the case of the 3D espace, around a fixed straight. This philosophy leads us to somehow see the Sun as a static point included in a rotation straight.

Therefore, the planets behave like objects which rotate around a transversal straight that passes through the Sun.

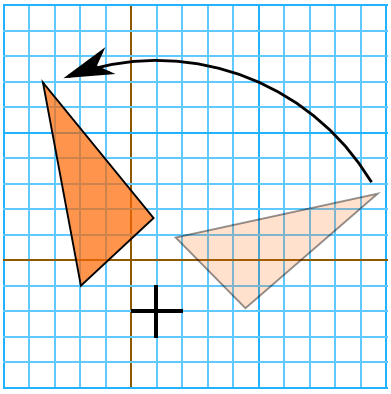

Figure 2: Rotation Example

Tipically, the rotation matrix associated with an angle θ and a vector $u = (u_x,u_y,u_z)$ is defined as follows:

$$R = \begin{bmatrix} \cos\theta + u_x^2 (1 - \cos\theta) & u_x u_y (1 - \cos\theta) - u_z \sin\theta & u_x u_z (1 - \cos\theta) + u_y \sin\theta \\ u_y u_x (1 - \cos\theta) + u_z \sin\theta & \cos\theta + u_y^2 (1 - \cos\theta) & u_y u_z (1 - \cos\theta) - u_x \sin\theta \\ u_z u_x (1 - \cos\theta) - u_y \sin\theta & u_z u_y (1 - \cos\theta) + u_x \sin\theta & \cos\theta + u_z^2 (1 - \cos\theta) \end{bmatrix}.$$

Figure 3: Rotation Matrix Expression

Once the matrix is created, the equation that retrieves the rotated vector *X'* from an orginal one *X* and a static point in the straight *A* is $X = A + R(X - A)$. We are going to name this equation as *"Rotation Equation"*.The problem now is how to find the rotation angle, the rotation vector, a fixed point and finally the rotation matrix.

Originally let us define the action plane on which the Sun is going to move. From now on, let us assume that the planets are going to move around the plane generated by the axis *x (1,0,0)* and *y (0,1,0)*. By setting this feature, it is clear that the transversal vector that is crossing the plane will always be *(0,0,1)* so we can build the vector $u = (u_x,u_y,u_z)$ as $u = (0,0,1)$. Although the reason why I am going to set the angle θ as 45º will be explained later, the rotation matrix around the

$$\begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 4: Result Rotation Matrix

vector *u = (0,0,1)* and the angle 45º stays as shown above. Note that *cos 45º = sen 45º* = $\sqrt{2}/2$.

It is important to higlight for future use that without setting the angle the matrix would look as follows:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\operatorname{sen}\theta & 0 \\ \operatorname{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 5: Rotation Matrix depending on the Angle

Another mathematical issue to be remembered which is going to be relevant when calculating the light intensity on the planets is the dot product of two vectors. The dot product of two vectors is an scalar value resulted from the equation $u \cdot v = |u| \cdot |v| \cdot \cos\theta$. The most important thing that we extract from this operation is the sign of the result. Below are shown the different signs obtained depending on the position of the vectors.

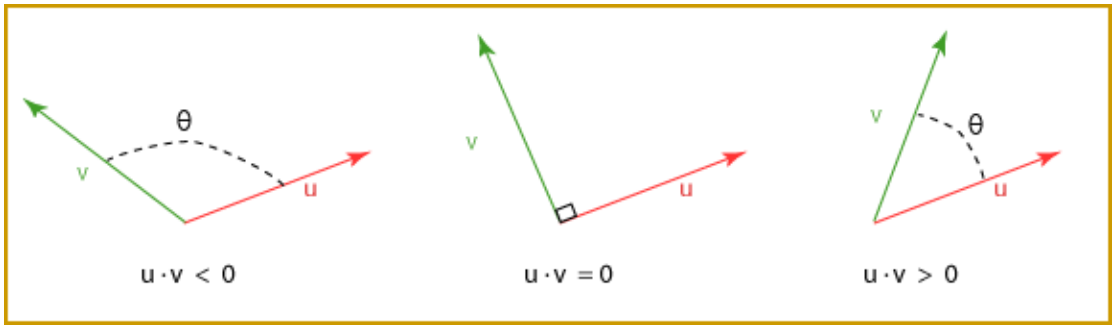

Figure 6: Sign Interpretation of the Dot Product

Hence, assuming that the light emitted from the sun is represented by the vector *u* and the normal vector of a point in a planet is represented by the vector *v*, we can associate the ilumination of the point by comparing the result of the dot product.

2 Implementation

As you may know by now, the Monkey Solar System model is the result of the pipeline and work produced by three shaders: a vertex shader, a geometry shader and a fragment shader. It is obvious that in order to display multiple objects (planets) and distribute them in the correct way from a single *mesh* object the presence of the geometry shader becomes essential.

The names of these shaders on the program are *shaderSystem.vert*, *shaderSystem.geom* and *shaderSystem.frag* respectively. Also, as the name of this report suggests, the object loaded on the associated *mesh* instance is a low poly monkey head.

2.1 Vertex Shader – *shaderSystem.vert*

The vertex shader is the first stage of the pipeline. In this stage we are going to set the position of the passed monkey and send the normals and texture coordinates to the next stage (in principle, we are rendering a textured model although the functionality of this feature will be shown in the fragment shader).

The code for the uniform and input variables is presented on the left. The input variables contain information about the position, normal and texture coordinate of each vertex.

```
//Version Number
#version 400

//The layout qualifers
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec2 VertexTexCoord;
layout (location = 2) in vec3 VertexNormal;

//Out variable

out VS_OUT {
    vec2 texCoords;
    vec3 normal;
} vs_out;

//Uniform variables


uniform mat4 model;
uniform mat4 projection;
uniform mat4 view;
```

Figure 7: Input, Output and Uniform Variables Vertex Shader

The uniform variables contain the model, view and projection matrix which will be used to update the local object coordinates to the one on clip space. Note that to achieve this we need to first pass them to world space by using the model matrix, then pass the result to the view space by using the view matrix and finally obtain the clip space by applying the projection one.

Within the input and uniform variables we can also find an output struct variable *VS_OUT* comprising two vector variables associated with the texture coordinate and the normal coordinate called *vs_out*.

Down below we can see the code for the main function. Not performing any unusual operation or transformation, this function sets the value of the *texCoords* of the *vs_out* as the one passed through the *VertexTexCoord* input variable and updates the normal by multiplying it by the known transpose of the inverse of the model matrix. This last result is normalized and set to the *normal* variable of the struct.

```
void main()
{
    //Assigning the texture coordinates and normal

    vs_out.texCoords = VertexTexCoord;
    vs_out.normal = normalize(mat3(transpose(inverse(model))) * VertexNormal);

    // Sets the position of the current vertex
    gl_Position = projection * view * model * vec4(VertexPosition, 1.0);

}
```

Figure 8: Vertex Shader Main

Finally, it is time to define the position that the vertex will hold on the scene, the gl_Position. Following the steps metioned before, this position is the result of multiplying the *VertexPosition* input with the uniform matrices.

2.2 Geometry Shader – *shaderSystem.geom*

The geometry shader is certainly the most important and relevant shader in The Monkey Solar System. It is the key stage from which all the planets and movements will be drawn and tracked. Let us start to analyze it by the input, output and uniform variables.

Tipically, in a geometry shader we define the geometry shape taken from the vertex shader and the one that is going to be passed to the fragment shader. These shapes are basically resumed in points, lines and triangles. Since the objective of our work is to duplicate the elements and not performing any other type of material transformation, we are taking single triangles and sending triangles to the next stage too (*triangle_strip*). However, how many new triangles is it creating? If we check the maximum number of new vertices that are going to be rendered, it is set to 12 (which means four new triangles as 12/3 = 4). One of this triangles will go to the Sun and each one of the rest three triangles will go to different planets. This means we are going to have one Sun and three planets orbiting around it. The shader is made to increase the number of vertices and hence the number of plantes as the user pleases.

```
//Version number
#version 330 core

//Layout qualfier
layout (triangles) in;
layout (triangle_strip, max_vertices = 12) out;

//Variable from provious stage

in VS_OUT {
    vec2 texCoords;
    vec3 normal;
} VertexIn[];

//Out variable

out VertexData {
    vec2 texCoord1;
    vec3 normal1;
    vec3 lightDir;
    vec3 pos;
} VertexOut;

//Uniforms

uniform mat4 translation;
uniform vec4 direction;
uniform vec4 direction_invert;
uniform vec4 point;
```

Figure 9: Input, Output and Uniform Variables Geometry Shader

Coming from the vertex shader we can see the *VS_OUT* struct variable here called *VertexIn[]*. It is worth noting that *VertexIn[]* is actually a vector which contains three *VS_OUT*, one for each vertex of the input triangle.

The output variable *VertexOut* will be send for each of the new vertices and it is a *VertexData* type comprising four vectors: the texture coordinates, the normal, the influence vector of light on it and the actual vertex position.

Nevertheless, the uniform variables might not be very intuitive. The matrix *translation* will always set as the resulted rotation matrix calculated with the vector $u = (0,0,1)$ and the angle $\theta = 45^{o}$ (See Figure 4). The vector *direction* is a coplanar vector of the plane formed by the axis *x* and *y* which is used to render the clockwise rotation planets. It is a vector that defines a direction to add to a determined Sun position in order to set the location of a new planet. This variable changes its value throughout the program by applying a rotation (See Figure 5) with a specific angle and the $u = (0,0,1)$ vector i.e. *direction* rotates around $u = (0,0,1)$. Just like *direction* does, *direction_invert* has the same properties but it defines the anti-clockwise rotation planets. It is always updated by a rotation matrix with the same angle as *direction* but with the opposite sign. The las uniform is a vector called *position* which hold the current position of the *mesh*: you may now imagine that just by knowing the Sun location and a direction vector from it, we can start drawing planets.

Let us take a look at the first part of the *main* function. First, we copy the value of the uniform *direction* and *direction_invert* to new local variables in order to update them. A counter variable is also set with value 1. Using an ordinary loop, we define the position of the Sun. Not applying any transformation, the new triangle has the same vertex position as the original one because we just copy the gl_Position from the vertex shader. In addition, we complete the values of the *VertexOut* variable by copying the texture coordinates in *VertexIn[].texCoords*, and filling the rest of the values with default null vectors. The reason why the rest of the values are default is because the Sun is not meant to perform any special operation in the fragment shader but the texture rendering. This texture will be in practice a hot Sun surface. To finish this first part, we finally terminate the first triangle by calling *EndPrimitive()* function.

```
void main()
{


    vec4 aux_direction = direction;
    vec4 aux_direction_invert = direction_invert;
    int counter = 1;

    for(int i=0; i < gl_in.length(); i++){

        VertexOut.texCoord1 = VertexIn[i].texCoords;
        VertexOut.normal1 = vec3(0.0,0.0,0.0);
        VertexOut.lightDir = vec3(0.0,0.0,0.0);
        VertexOut.pos = vec3(0.0,0.0,0.0);
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }

     EndPrimitive();
```

Figure 10: First Main Part Geometry Shader

The next step is rendering the planets. All this procedure takes place in a loop with three iterations. However, as it was said before, we can increase the number of planets to render just by adding multiples of three to *max_vertices* and iterations to this loop.

The planets are not textured so the *VertexOut.texCoord1* variable is set to null with a not valid value *(-1,-1)*. Also, as the planets are not rotated but only translated to a new position, normals can just be copied to *VertexOut.normal1* from the original ones.

Then, a differentiation is made between even and odd iterations in order to create clocwise and anti-clockwise planets. One way or another, the position of every vertex is set as the result of adding the *aux_direction* or *aux_direction_invert* vector multiplied by the counter to the original position (this allows the planets to get away from the Sun). Similarly, the light direction of every vertex is set as the *aux_direction* or *aux_direction_invert* vector (think of it as the direction of the light source which is the Sun). Unlike the previous vertices, this time we do set the *VertexOut.pos* to the value of the used position.

```
for(int i = 0; i < 3; i++){

    VertexOut.texCoord1 = vec2(-1,-1);
    VertexOut.normal1 = normalize(VertexIn[i].normal);

    if(mod(i,2) == 0){

        for(int j = 0; j < gl_in.length(); j++){


            VertexOut.lightDir = normalize(aux_direction.xyz);
            gl_Position = gl_in[j].gl_Position + aux_direction*counter;
            VertexOut.pos = gl_Position.xyz;

            EmitVertex();
        }

        EndPrimitive();

        aux_direction = point + translation *(aux_direction - point);
        aux_direction[3] = 0;

    }

    else{

        for(int j = 0; j < gl_in.length(); j++){


            VertexOut.lightDir = normalize(aux_direction_invert.xyz);
            gl_Position = gl_in[j].gl_Position + aux_direction_invert*counter;
            VertexOut.pos = gl_Position.xyz;

           EmitVertex();
        }

        EndPrimitive();

        aux_direction_invert = point + translation * (aux_direction_invert - point);
        aux_direction_invert[3] = 0;

    }

    counter++;

}
```

Figure 11: Second Main Part Geometry Shader

After emiting each vertex and later ending the triangle, it is time to update the direction vector. The reason why this vector is updated is to create an angle between the planets that share the same wise to give more reality to the model. To achieve that, the vector is introduced in the *Rotation Equation* and used later for the next triangle. This way, same wise planets will have an angle deviation of 45º. Finally, the last stage in the loop is to increase the counter in one unit.

2.3 Fragment Shader – shaderSimmetry.frag

Let us move to the last stage: the fragment shader. Looking at the variables of this shader we can find the *VertexData* struct variable (here called *Vertex*) from the previous stage. Of course, the top output variable at this shader is the *fragcolor* which defines the color of each fragment. Along with the variables above, we also define new uniforms: the *projection* matrix (seen in vertex shader) and sampler2D variable called *diffuse* tasked with the texture visualization.

```
//Version number
#version 400
//Layout Qualifer
layout( location = 0 ) out vec4 fragcolor;

in VertexData {
   vec2 texCoord1;
   vec3 normal1;
   vec3 lightDir;
   vec3 pos;
} Vertex;

//Uniform variables

uniform mat4 projection;

uniform sampler2D diffuse;
```

Figure 12: Input, Output and Uniform Variables Fragment Shader

In the main function, we first check if the fragment corresponds to the Sun or a planet. If it is the Sun we simply assign the color with the texture. If not, rim and light intensity features will be added to the fragment. For the rim effect, it is first needed to undo one transformation in the position variable: we need to get the view space position coordinates by multiplying the inverse of the projection matrix by the *Vertex.pos*. Then, this position is negated and the dot product of the result with the normal vector is performed. In the end, the result of this operations constitutes the rim contribution and it is saved on the vdn variable.

```
void main()
{

    if(Vertex.texCoord1[0] == -1){   //It is not the sun



        vec3 n = Vertex.normal1;
        vec3 p = normalize(inverse(projection) * vec4(Vertex.pos,0)).xyz;

        vec3 v = - p;

        float vdn = 0.4 - max(dot(v, n), 0.0);        // the rim contribution

        float intensity;
        vec4 color;
        intensity = dot(Vertex.lightDir,  - Vertex.normal1);



        if (intensity > 0.0)
           color = vec4(0.2 + 0.8*intensity, 0.1 + 0.4*intensity, 0.1 + 0.4*intensity, 1.0);
        else
           color = vec4(0.2,0.1,0.1,1.0);

      fragcolor.a = 0.4;
      fragcolor.rgb = vec3(smoothstep(0.4, 0.8, vdn)) + vec3(color.x,color.y,color.z);



    }

    else {
       fragcolor = vec4(texture(diffuse, Vertex.texCoord1));
    }

}
```

Figure 13: Main Fragment Shader

The light intensity is calculated by applying the dot product between the light direction and again, the normal (but this time negated). This gives as a parammeter that will be used to update the light intensity along with the rim contribution (See figure 6).

2.4 The *setSystem* function

sWhen drawing the model, there is a function in the *MainGame* class used to set the values of all the uniforms: *setSystem(vec3 variable, float lapsus).*

```
void MainGame::setSystem(glm::vec3 variable, float lapsus)
{


	shaderSystem.setMat4("model", glm::mat4(transform.GetModel()));
	shaderSystem.setMat4("view", myCamera.GetView());
	shaderSystem.setMat4("projection", myCamera.GetProjection());

	shaderSystem.setVec4("point", glm::vec4(variable, 1));

	glm::mat4 translation;


	translation[0] = glm::vec4(sqrtf(2.0f) / 2.0f, -sqrtf(2.0f) / 2.0f, 0, 0.0f);
	translation[1] = glm::vec4(sqrtf(2.0f) / 2.0f, sqrtf(2.0f) / 2.0f, 0, 0.0f);
	translation[2] = glm::vec4(0.0f, 0.0f, 1.0f, 0.0f);
	translation[3] = glm::vec4(0.0f, 0.0f, 0.0f, 1.0f);


	shaderSystem.setMat4("translation", glm::mat4(translation));

	// Clock-wise

	glm::mat4 translation_offset;

	translation_offset[0] = glm::vec4(cos(lapsus), -sin(lapsus), 0, 0.0f);
	translation_offset[1] = glm::vec4(sin(lapsus), cos(lapsus), 0, 0.0f);
	translation_offset[2] = glm::vec4(0.0f, 0.0f, 1.0f, 0.0f);
	translation_offset[3] = glm::vec4(0.0f, 0.0f, 0.0f, 1.0f);

	glm::vec4 pos = glm::vec4(variable, 1) + translation_offset * (glm::vec4(3, 0, 0, 0) - glm::vec4(variable, 1));
	pos[3] = 0;

	shaderSystem.setVec4("direction", pos);

	// Anti clock-wise


	glm::mat4 translation_offset_invert;

	translation_offset_invert[0] = glm::vec4(cos(-lapsus), -sin(-lapsus), 0, 0.0f);
	translation_offset_invert[1] = glm::vec4(sin(-lapsus), cos(-lapsus), 0, 0.0f);
	translation_offset_invert[2] = glm::vec4(0.0f, 0.0f, 1.0f, 0.0f);
	translation_offset_invert[3] = glm::vec4(0.0f, 0.0f, 0.0f, 1.0f);

	pos = glm::vec4(variable, 1) + translation_offset_invert * (glm::vec4(3, 0, 0, 0) - glm::vec4(variable, 1));
	pos[3] = 0;

	shaderSystem.setVec4("direction_invert", pos);

}
```

Figure 14: SetSystem Funtion

This function set the values of the coordinate system matrices from the *transform* and *myCamera* instances, the *point* uniform as the *passed* variable (mesh position), the *translation* matrix as the Result Rotation Matrix (See figure 4) and the direction vectors by applying a rotation transformations to the vector *(3,0,0,0)* according to the value of the *lapsus* (counter variable).

3 References

- Learn OpenGL, https://learnopengl.com/
- Movimientos Rígidos, UGR https://www.ugr.es/~rcamino/docencia/geo1-03/g1tema9.pdf