Simple 3D representation of planets with OpenGL

By: Maxime Pichet

INFO-H502

Presented to Gauthier Lafruit

December 19th 2023

Contents

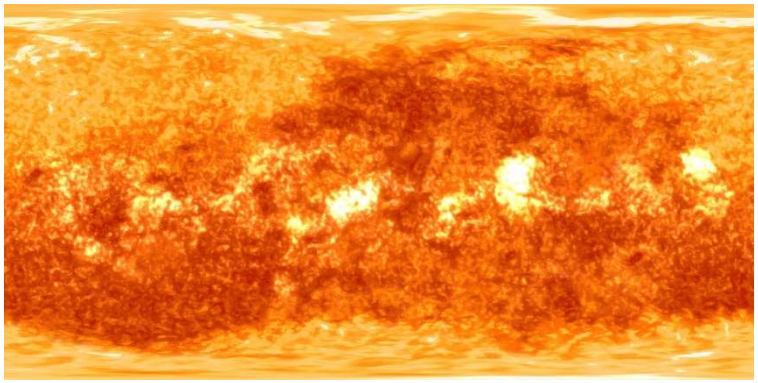
Project description

This project was made using OpenGL and C++. It consists of 4 planets, notably the sun, the earth, a reflective sphere and a refractive one. The sun rotates on itself as the earth rotates on itself and around the sun. The reflective sphere also rotates around the sun. In sum, it is a 3d model of some planets in space.

Sun

The center piece of the scene is the Sun. The sun was made using a basic sphere that was modeled in blender and then imported in the program as an object file. In order to add some texture to this sphere, a picture of the sun was mipmapped onto the sphere.

Picture 1. Texture of the sun



This picture was loaded using the stb library (see Annex). The texture data was then passed to the vertex shader using the buffer.

```
in vec3 position;
in vec2 tex_coord;
in vec3 normal;

out vec2 f_tex_coord;
out vec3 FragPos;
out vec3 Normal;

uniform mat4 M;
uniform mat4 itM;
uniform mat4 V;
uniform mat4 P;

void main() {
    FragPos = vec3(M * vec4(position, 1.0));
    gl_Position = P * V * vec4(FragPos, 1.0);

    // Transform normals correctly
    Normal = normalize(vec3(itM * vec4(normal, 1.0)));

    // Pass texture coordinates to fragment shader
    f_tex_coord = tex_coord;
}
```

Figure 1. vertex shader for the Sun

We can see here that the tex_coord which are the coordinates for the texture, were passed to the fragment shader.

```
in vec2 f_tex_coord;
in vec3 FragPos;
in vec3 Normal;

out vec4 FragColor;

uniform sampler2D textureSampler;
uniform vec3 u_light_pos;
uniform float u_light_intensity;
uniform vec3 u_light_color;


    vec3 lightDir = normalize(u_light_pos - FragPos);
    float diffusion = max(0.0, dot(Normal, lightDir)) * u_light_intensity;
    vec3 diffuse = u_light_color * diffusion * 0.9;

    vec4 textureColor = texture(textureSampler, f_tex_coord);


    vec3 finalColor = textureColor.rgb * diffuse;

    FragColor = vec4(finalColor, 1.0);
}
```

Figure 2. Sun fragment shader

In the fragment shader, I decided to only apply diffuse lighting as all points of the sphere should emit the same amount of light. Using the texture() function of glsl, we can sample the texel of the texture and apply it at the f_tex_coord or the pixel of the texture.

```
shaderSun.use();
shaderSun.setInteger("textureSampler", 2);
```

Figure3. Texture sampler passed to shader.

To have access to the texture, we pass as a uniform a 2d sampler that correspond to the texture. Here we passed 2 since when loading the texture (see Annex), we saved it in location 2. Then the sun was ready to be drawn.

Earth

The earth was a bit more complex to make as I added multiple elements to it. The texture was done the same way as for the sun but using a picture of the earth instead. What is interesting about the earth is I added bump mapping to make it seem like there is elevation on the surface.

Figure 4. Bump mapping

This was achieving using a normal map.

Figure 5. Normal map of the earth

The normal map seems all purple but if you look closely, you can see some differences. To correctly apply the normal map to the texture, the tangents and bitangents of the surface had to be calculated. Then those calculations were passed to the vertex shader using the buffer where the TBN (tangent, bitangent, normal) matrix was computed and passed to the fragment shader.

```
void main() {
    vec4 frag_coord = M * vec4(position, 1.0);
    gl_Position = P * V * frag_coord;
    vec3 T = normalize(vec3(M * vec4(tangent, 0.0)));
    vec3 B = normalize(vec3(M * vec4(bitangent, 0.0)));
    vec3 N = normalize(vec3(M * vec4(normal, 0.0)));
    TBN = mat3(T, B, N);

    f_tex_coord = tex_coord;

    v_frag_coord = frag_coord.xyz;
}
```

Figure 6. Vertex shader of the Earth

```
float specularCalculation(vec3 N, vec3 L, vec3 V, float shininess) {
    vec3 R = reflect(-L, N);
    float cosTheta = dot(R, V);
    float spec = pow(max(cosTheta, 0.0), shininess);
    return 0.8 * spec;
}

void main() {

    vec3 L = normalize(u_light_pos - v_frag_coord);
    vec3 V = normalize(u_view_pos - v_frag_coord);
    // normals are [-1,1] and color is [0,1]
    vec3 normalMap = normalize(texture(normalMap, f_tex_coord).rgb * 2.0 - 1.0);
    vec3 N = normalize(TBN * normalMap);
    float diff = max(dot(N, L), 0.0);

    float shininess = 32.0;
    float specular = specularCalculation(N, L, V, shininess);


    float distance = length(u_light_pos - v_frag_coord);
    float attenuation = 1.0 / (0.03 * distance + 0.008 * distance * distance);

    float light = 0.06 + attenuation * (diff + specular);

    vec4 textureColor = texture(textureSampler, f_tex_coord);
    vec3 finalColor = vec3(textureColor.rgb * vec3(light) * u_light_color);

    FragColor = vec4(finalColor, 1.0);
}
```

Figure 7. Fragment shader of the Earth

In the fragment shader presented above, we create and normalize a texture (normalMap) which is the picture presented in figure 5. This is then multiplied with the TBN matrix and normalized to have unit vectors. We then have the new normals which are used to calculate the lighting equations.

The earth also has different model transformations for the animation which you can see in the source code and the short video.

Reflective and refractive sphere

The implementation of the reflective sphere was straightforward. The work must be done in the fragment shader in order to have a cubemap reflection on the sphere.

```
void main() {
    vec3 N = normalize(v_normal);
    vec3 V = normalize(u_view_pos - v_frag_coord);
    vec3 R = reflect(-V, N);

    // Sample from the cube map
    vec3 cubeMapColor = texture(cubemapSampler, R).rgb;


    vec3 finalColor = 0.9 * cubeMapColor;

    FragColor = vec4(finalColor, 1.0);
}
```

Figure 8. Fragment shader for reflection

From the normalized normals and the view vector, we can calculate the reflection vector used to sample the cube. The nifty reflect function from glsl does the math for us. Otherwise, equation 1 could be used to calculate it.

$$\mathrm{R} = 2\,(\mathrm{L} \cdot \mathrm{N})\,\mathrm{N} - \mathrm{L}$$

Eq 1. Reflection calculations from vector

The refractive sphere was quite similar but the fragment shader differs a bit.

```
#version 400 core
out vec4 FragColor;
precision mediump float;
in vec3 v_frag_coord;
in vec3 v_normal;
uniform vec3 u_view_pos;

uniform samplerCube cubemapSampler;
uniform float refractionIndice;

void main() {
    float ratio = 1.00 / refractionIndice;
    vec3 N = normalize(v_normal);
    vec3 V = normalize(u_view_pos - v_frag_coord);
    vec3 R = refract(-V,N,ratio);
    FragColor = texture(cubemapSampler,R);
}
```

Figure 9. Fragment shader for refraction

Here, the refract function from glsl is used with a ratio which is denoted as eta in the documentation. It is the ratio from material A to B in other words going from air to water let's say. We used 1.00 since we are in space so we should not have any refraction from the environment. Then we call the refract function and generate the texture with the refraction vector. In the simulation the two balls were put side to side to see the difference between reflection and refraction.

Particle system

One aspect of the animation that was complex is the particle system. Around the sun, we can see some orange particles flying of in every direction. This is representing the sparkling of the sun. To achieve this particle system, hundred of particles were generated following this struct.

```
struct Particle {
    float x, y, z;
    float vx, vy, vz;
    float lifetime;
    float r, g, b;
};
```

Figure 10. Particles struct.

Each particle has a position, velocity, lifetime and a color. A vector of Particle could then be generated with random starting positions inside the sphere and move out with a certain velocity. At the end of their lifetime, the particles would be deleted, and new ones would be generated.

```
void updateParticles() {
    for (auto& particle : particles) {
        particle.x += particle.vx;
        particle.y += particle.vy;
        particle.z += particle.vz;
        particle.r -= 0.01f;
        particle.g -= 0.005f;
        particle.lifetime -= 0.005f;
    }
    // Remove particle if they died
    particles.erase(std::remove_if(particles.begin(), particles.end(),
        [](const Particle& p) { return p.lifetime <= 0.0f; }),
        particles.end());
}
```

Figure 11. Particle update function

Basically, at each iteration, particles would move at a constant speed, diminish in color and reduce their lifetime. The last few lines of code shows a concise way of removing dead particles. It uses two iterators to rearrange the vector by putting the particles meeting the condition at the end. After that, it erases the

particles that are at the end of the vector. Then, we render the vector as points, and we have a particle system.

```
glDrawArrays(GL_POINTS, 0, particles.size());
```

Figure 12. Particle generation

```
void main()
{
    vec3 FragPos = vec3(M * vec4(position, 1.0));
    gl_Position = P * V * vec4(FragPos, 1.0);
    ParticleColor = vec3(color.r,color.g, color.b);
}
```

Figure 13. Particle vertex shader

```
void main()
{
    FragColor = vec4(ParticleColor, 1.0);
}
```

Figure 14. Particle fragment shader

The fragment and vertex shader are very simple as only colored points were required.

Cube Map

The cube map consists of loading a picture on each face of a cube and then making each vertex the deepest possible.

```
void main(){
        texCoord_v = position;
        //remove translation info from view matrix to only keep rotation
        mat4 V_no_rot = mat4(mat3(V)) ;
        vec4 pos = P * V_no_rot * vec4(position, 1.0);
        // the positions xyz are divided by w after the vertex shader
        // the z component is equal to the depth value
        // we want a z always equal to 1.0 here, so we set z = w!
        // Remember: z=1.0 is the MAXIMUM depth value ;)
        gl_Position = pos.xyww;
}
```

Figure 15. Vertex shader of cubemap

It is explained in figure 15 that we remove the translation because we want to make it infinitely far so we should not move the cube. Afterwards, we calculate the position of each vertex as their position with z = w since w will always be 1.0 because we set it so in vec4(position, 1,0). Hence, we can pass z as w and

we have the cube at maximum depth. Then it's just a matter of loading the right images on the right faces. We can do so by creating a GL_TEXTURE_CUBE_MAP which is similar to a 2d texture but each mipmap level has 6 faces. So, using a samplerCube in glsl let's us create a cubemap with ease.

Annex A:

```
void loadTexture(const char* imagePath, GLenum textureUnit) {
    int width, height, channels;
    stbi_set_flip_vertically_on_load(true);
    unsigned char* image = stbi_load(imagePath, &width, &height, &channels, STBI_rgb);

    if (image) {
        glActiveTexture(textureUnit); // Activate the specified texture unit
        glGenTextures(1, &textureID);
        glBindTexture(GL_TEXTURE_2D, textureID);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
        glGenerateMipmap(GL_TEXTURE_2D);

        stbi_image_free(image);
    }
    else {
        std::cerr << "Failed to load texture: " << stbi_failure_reason() << std::endl;
    }
}
```

Figure 16. texture loading

Loads texture and assigns it to a textureUnit number to have multiple textures.