# FRR Lab 2

Alberto Jovane

May 28, 2018

## 1 Reflection

In the first part we developed an environment reflection. In order to do that we used a cube texture, representing the surrounding environment.

### 1.1 Vertex shader

```
#version 330

layout (location = 0) in vec3 vert;
layout (location = 1) in vec3 normal;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
uniform mat3 normal_matrix;

smooth out vec3 world_normal;
smooth out vec3 world_vertex;
smooth out vec3 world_camera_pos;


void main(void) {
    world_camera_pos = inverse(view)[3].xyz;
    world_vertex = vec3(model*vec4(vert,1.0));
    world_normal= vec3(normal);

    gl_Position = projection * view * model * vec4(vert, 1);
}
```

In this case the vertex shader take as input the position **vert** and the normal **norm**.
As uniforms it takes the projection, view, model and normal matrices
The main purpose of the vertex shader is to generate the basic value for the fragment shader. In this case we generate the position of the object in world space, and the normals .
We also need the camera position, in order to get it, we use the last row of the inverted view matrix. Finally the position of the mesh transformed is set.

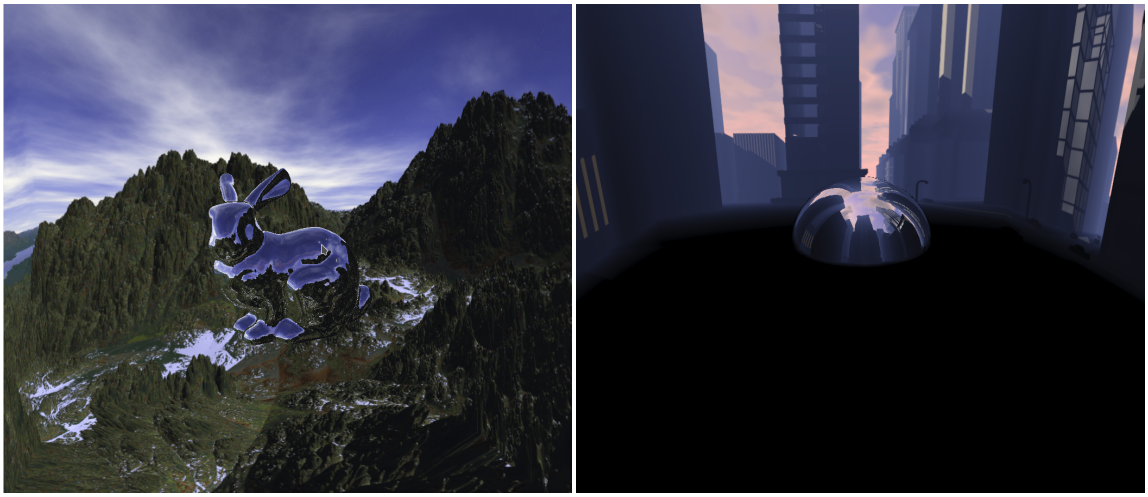

Figure 1: Two examples of reflection.

## 1.2 Fragment shader

```
#version 330

smooth in vec3 world_normal;
smooth in vec3 world_vertex;
smooth in vec3 world_camera_pos;

uniform samplerCube specular_map;
uniform samplerCube diffuse_map;
uniform vec3 fresnel;

uniform mat4 view;


out vec4 frag_color;


void main()
{
    vec3 n = normalize(world_normal);
    vec3 v = normalize(world_camera_pos - world_vertex);
    vec3 l = reflect(-v,n);

    //frag_color = texture(specular_map,l);
    //gamma correction
    vec3 final_color = texture(specular_map, l).rgb;
    frag_color = vec4(pow(final_color, vec3(1.0/2.2)),1);

}
```

The fragment shader has to generate the actual reflection, this works by sampling the correct value on the cube matrix, correspondent to the reflected vector.
The main idea is: take the position of the camera and subtracting the position of the fragment, in order to have the incident vector **v** (and then normalize it). But the vector is then going from the fragment to the camera, we nee the on in the other way, so we use $-v$.
Then from this value and the normal we generate the reflection. This reflection give us the correct position where to sample on the cube map in order to have a reflection of the environment.

# 2 BRDF

In the second part we implemented an approximated image based lighting approach. The idea is recreate a realistic light reflection, simulating different kind of materials.

## 2.1 Vertex shader

The vertex shader behave in the same way as before.

## 2.2 Fragment shader

```
#version 330

smooth in vec3 world_normal;
smooth in vec3 world_vertex;
smooth in vec3 world_camera_pos;

uniform samplerCube specular_map;
uniform samplerCube diffuse_map;
uniform vec3 fresnel;
uniform mat4 view;

out vec4 frag_color;

float G1(vec3 n, vec3 v, vec3 l, vec3 h)
{
    //geometry 2
    float Ga = (2*dot(n,h)*dot(n,v))/dot(v,h);
    float Gb = (2*dot(n,h)*dot(n,l))/dot(v,h);
    float G = min(Ga,Gb);
    if(G > 1)
        G = 1;
    return G;
}

void main (void) {

    vec3 n = normalize(world_normal);
    vec3 v = normalize(world_camera_pos - world_vertex);
    vec3 l = reflect(-v,n);

    float G2 = G1(n,v,l,n);

    float i = (fresnel.x + fresnel.y + fresnel.z)/3.0;
    vec3 F = fresnel + (1 - fresnel)*pow((1 - dot(l,n)),5.0) + (1 - i)*0.4;
  // vec3 F = fresnel + (1 - fresnel)*pow((1 - dot(l_c,h_c)),5.0);


    vec3 D = texture(diffuse_map,l).rgb;
    D = pow(D, vec3(1.0/2.2));


  //vec3 f = F*G2*D/(4*dot(n,l)*dot(n,l));
  vec3 f = (1 - i)*D + F*D/(4*dot(n,l)*dot(n,l));
    frag_color = vec4(f, 1);

}
```

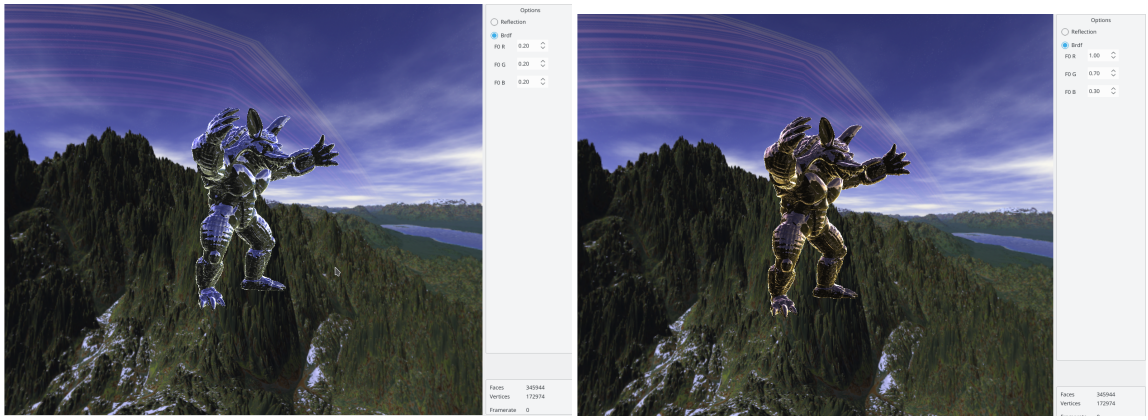

Figure 2: Brdf with high reflectance diffuse map.

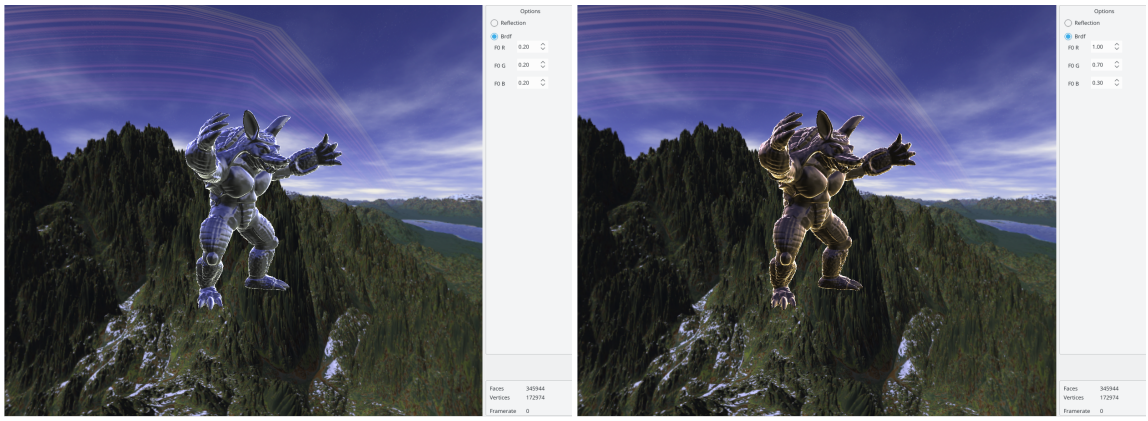

Figure 3: Brdf with rough diffuse map.

The vertex shader is the part where all the computation is done.
The first thing we do is normalize the coordinates, and, as for the reflection, compute the reflected vertex **l**. The approximation of the brdf we developed is the Microfacet Specular Brdf, this is based on three factors:

- Fresnel Reflectance **F0**: is a fraction of incoming light. The physical theory said that in most of the material a part of light is absorbed by the object, and the one that we see is the reflected one. Our approximation works with three parameters F0 as input: one for the color Red, one for Green, and one for Blue.
- Normal Distribution Function **D**: we are going to approximate it by a value taken from a pre filtered diffuse map. This diffuse map simulates the roughness of the object, that usually are not perfectly flat. We generated 5 different map, with different parameters, in order to simulate objects that are more specular or more rough.
- Geometry Function **G**: the last item is the geometry function. It has a value between 0 and 1, that compute if a microfacet is lit and visible. For the final computation we approximated it to 1, because there are not visible different with the computed G.

In the fragment shader we can see all the steps of the approximation. First we compute the Geometry function (we are not going to use it in the final approximation). Then the Fresnel reflectance and, in order to get a better result, we introduced the intensity i, as a value to weight the Fresnel: $i = \frac{F0_R + F0_G + F0_B}{3}$
Finally we extract from the selected pre-filtered diffuse map the correspondent reflectance.
The final computation of the BRDF is then: $brdf = \frac{(1-i)*D + F*D}{4(n \cdot l)^2}$

# 3 Other parts

## 3.1 Generation of diffuse maps

The diffuse maps were generated with an external software, that allow the user to regulate the parameters, in order to obtain different level of roughness. (On the delivered folder, there are more diffuse map then the one showed in this paper).

## 3.2 Gamma correction

In order to improve the quality of the image we deal directly we the problem of gamma correction. We used a gamma of 2.2, that generally works for all kind of monitors. We manage linear inputs, so we set the **glTexImage2D** with the parameter **GL_SRGB_EXT** to work in sRGB space. Then after extracting the pixel-value we apply the gamma correction, by elevating it to $\frac{1.0}{2.2}$, and then we send it to the be displayed.