Fast Realistic Rendering

LAB 2

Emeline GOT

1 Reflection

1.1 Code

The first part of the project consisted in writing an environment reflection shader using a cube texture. To do so, we had to work in world space. This provides the possibility to use a still cubemap as the surrounding world and as a source for the reflection on the models we use. By accessing to the position of the camera and of the vertices in the world space in the vertex shader, we can compute the reflection in the fragment shader.

reflection.vert

```
#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;
flat out vec3 world_camera_pos;

void main(void)  {
  //Camera position in world space
  world_camera_pos = inverse(view)[3].xyz;

  //Vertex position in world space
  world_vertex = (model * vec4(vert,1)).xyz;

  //Normal vector in world space
  world_normal= normal;

  //Vertex position in clip space
  gl_Position = projection * view * model * vec4(vert, 1);
}
```

The camera position in world space corresponds to the third column of the inverse of the view matrix. We obtain the vertex position in world space by multiplying the coordinates of the vertex in the local space by the model matrix. We send those two vectors as well as the normal vectors to the fragment shader.

As usual in vertex shaders, we have to define the vertex positions in clip space by multiplying them in model space by the model, view & projection matrices and assign it to the predefined $gl\_Position$.

reflection.frag

```
#version 330

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

uniform samplerCube specular_map;

uniform mat4 view;
out vec4 frag_color;


void main()
{
    // Normal vector
    vec3 N = normalize(world_normal);

    // - Incident vector
    vec3 I = normalize(world_camera_pos - world_vertex);

    // Reflection of - I against N
    vec3 R = reflect(-I,N);

    // Application of the gamma correction on the specular map
    vec3 gamma = pow(texture(specular_map,R).rgb,vec3(1.0/2.2));

    frag_color = vec4(gamma,1.0);

}
```

To compute the reflection, we have to compute the color obtained in the direction of the reflected direction. The reflected direction is obtained by reflecting the incidence direction against the normal. The incidence direction corresponds to the vector that goes from the camera position to the position of the vertex. We thus obtain it by doing the difference between $world\_camera\_pos$ and $world\_vertex$. The reflection is then computed with the function $reflect$ (with $-I$ to have the correct direction for the incidence vector).

Once we have the direction of the reflection, we want to get the color in that direction to give it to the considered fragment. To do so, we take the value of the specular map that is stored as a texture and used in the shaders. However, we have to take into account the gamma correction. Indeed, in the shaders, the images are treated as linear. It means that we will be able to add the contributions of different light sources or multiply light values by reflectances. But this is problematic because the images that we send in the first place are not linear but exponential. To be able to take into account this particularity, we send the textures as sRGB. This allows them to be used as linear in the shaders. We can do this by replacing GL_RGBA by GL_SRGB_ALPHA in the call of the function $glTexImage2D$.

```
glTexImage2D(cube_map_pos, 0, GL_SRGB_ALPHA, image.width(), image.height
    (), 0, GL_BGRA, GL_UNSIGNED_BYTE, image.bits());
```

When we get the values stored in the texture in the fragment shader, we have to do the inverse transformation from linear to exponential in order to take into consideration the fact that monitors are not linear either. We consider that their gamma value is 2.2 and calculate the value of the texture at the power of $\frac{1}{\gamma} = \frac{1}{2.2}$. This will allow to compensate the gamma transformation applied by the monitor — which is the values put at the power of γ — and preserve the real colors without artifacts created by the display devices.

1.2 Result

The result of this first part is completely reflective objects that show the surrounding environment (cubemap).

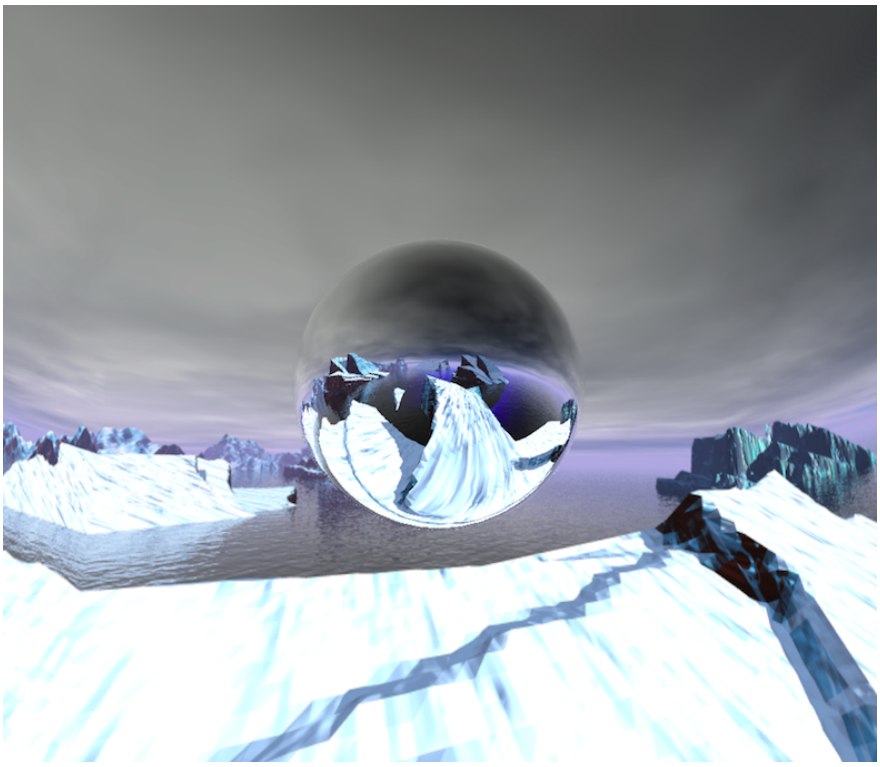

FIGURE 1 – Result of the environment reflection shader on a sphere

2 BRDF

2.1 Code

In this second part, we implement an image based lighting approach that uses microfacet approximation. We use the BRDF (Bidirectional Reflectance Distribution Function) that is defined as follow :

$$BRDF(l,v) = \frac{F(l,h)G(l,v,h)D(h)}{4(n\cdot l)(n\cdot l)} \tag{1}$$

Where v is the vector from the vertex to the observer in world space, n is the normal vector and l is the reflection of v against n. h is the half-vector defined as $h = \frac{v+l}{||v+l||}$.

The F function we used is given by the Schlick approximation to Fresnel reflectance :

$$F = F_0 + (1 - F_0)(1 - (n\cdot l))^5 \tag{2}$$

The Geometry function G is equal to 1. We have tried different values for the geometry function but there were no change in the results.

The Distribution function D is defined by the values stored in the diffuse maps in the direction of l.

brdf.vert

```
#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;
flat out vec3 world_camera_pos;

void main(void)  {

    //Camera position in world space
    world_camera_pos = inverse(view)[3].xyz;

    //Vertex position in world space
    world_vertex = (model * vec4(vert,1)).xyz;

    //Normal vector in world space
    world_normal= normal;

    //Vertex position in clip space
    gl_Position = projection * view * model * vec4(vert, 1);
}
```

The vertex shader we use is the same as for the reflection since we use the same vectors in world space.

brdf.frag

```
#version 330

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

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

out vec4 frag_color;

float alpha = 0.3;


//------------------------------------------------------------------
//--------------- Fresnel Schlick approximation --------------------
//------------------------------------------------------------------


vec3 FresnelSchlick(vec3 F0, vec3  l, vec3 h)
{
   return F0 + (vec3(1.0,1.0,1.0)-F0) * pow((1-dot(l,h)),5.0);
}

//------------------------------------------------------------------
//--------------------- Geometry functions -------------------------
//------------------------------------------------------------------

float Geometry(vec3 n, vec3 v)
{
    float k = alpha * alpha / 2;
    return (dot(n,v) / (dot(n,v)*(1-k)+k));
}

float G2(vec3 n, vec3 v, vec3 l){
    float NdotV = max(dot(n,v),0.0);
    float NdotL = max(dot(n,l),0.0);
    float gg1 = Geometry(n,v);
    float gg2 = Geometry(n,l);
    return gg1*gg2;
}

float G3(vec3 n, vec3 l, vec3 v, vec3 h){
    return min(1,min((2*dot(n,h)*dot(n,v)/dot(v,h)),(2*dot(n,h)*dot(n,l)/
        dot(v,h))));
}

//------------------------------------------------------------------

```

```
51 void main()
52 {
53     //Vector definitions in world space
54     vec3 n = normalize(world_normal);
55     vec3 v = normalize(world_camera_pos - world_vertex);
56     vec3 l = reflect(-v,n);
57     vec3 h = normalize(l+v);
58
59     //Intensity of F0
60     float I = (fresnel.r + fresnel.g + fresnel.b)/3.0;
61
62     //Fresnel reflectance
63     vec3 F = FresnelSchlick(fresnel,n,l) + 0.4*(1-I);
64
65     //Geometry functions
66     float G = 1;
67     float G_2 = G2(n,v,l);
68     float G_3 = G3(n,l,v,h);
69
70     //Normal distribution with gamma correction
71     vec3 D = pow(texture(diffuse_map,l).rgb,vec3(1.0/2.2));
72
73     //BRDF Formula
74     vec3 color = (1-I)*D + F*G*D / (4*dot(n,l)*dot(n,l));
75
76     frag_color = vec4(color,1.0);
77
78 }
```

In the fragment shaders, we compute the Fresnel reflectance using the formula (2). For the geometry functions, there is several possibilities but since there is no noticeable change between the different formulas, we use $G = 1$.

For the distribution function, we get the values stored in the diffuse maps by taking into account the gamma correction as we did for the reflection part.

Since the results obtained were still a little bit dark for some values of the Fresnel term (F_0), we added some improvements by computing I the intensity of F_0. Then we added a fraction of $(1 - I)$ to the Fresnel reflectance and $(1 - I)D$ to the total color.

2.2 Results

After computing cubemaps with different material roughness, we can obtain objects that are very reflective or rather a little bit reflective, almost opaque.

FIGURE 2 – Example of BRDF with a very specular map

FIGURE 3 – Example of BRDF with a very specular map and a gold color

FIGURE 4 – Example of BRDF with a rougher map

FIGURE 5 – Example of BRDF with a very rough map

FIGURE 6 – Example of BRDF with a very rough map and a gold color

FIGURE 7 – Example of BRDF with a very rough map and a silver color