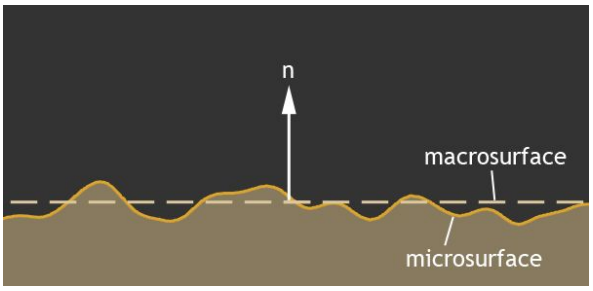
Physically Based Rendering

A4 - PBR Renderer Implementation

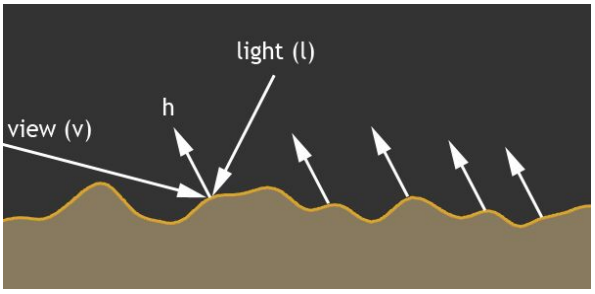
Arnau Ruiz NIA:192961
Victor Ruiz NIA:174227

I. Introduction

In this report we will explain the implementation of a complete PBR material, which is a method of shading and rendering that provides a more accurate representation of how light interacts with surfaces. We have two main points to implement and explain, the environment lighting (image based lighting) and the punctual lighting of the object. This is based on the microfacet theory, and states that, while 'n' represents a kind of average normal of the surface at a point, it is actually made up of microscopic gouges. These microscopic gouges are themselves made up of tiny microfacets that are each optically flat.



Given a view direction "v" and light direction "l", it is clear that the vector halfway between, "h", represents the normal of the surface that reflects "v" into "l" or vice versa. With "h" defined, we can describe the overall BRDF.



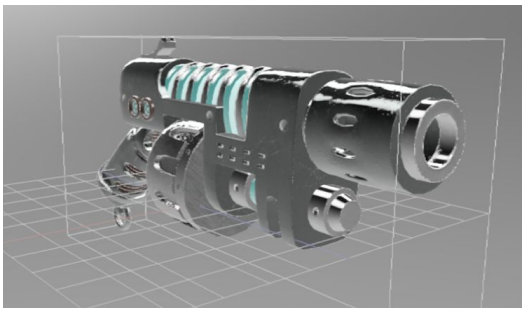
We will see with more detail why do we use these formulas and how we implemented this algorithms to our PBR renderer program.

II. Theory and implementation

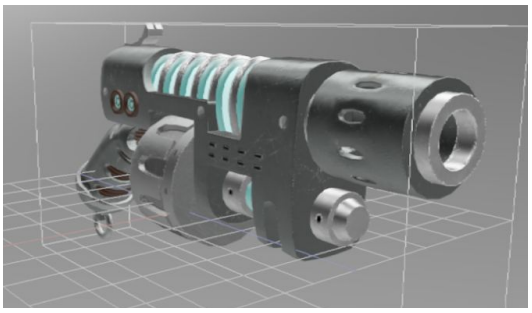
Before the implementation of the algorithms, we must know that environment textures will be tone-mapped and linearized.

Environment Lighting (IBL)

This image based lighting simulates incoming light using images. So what we are going to do is to read pixels from an environment texture and to use different HDRE levels. Each HDRE has 6 levels to simulate roughness.



"Blaster" with roughness value of 0



"Blaster" with roughness value of 1

In order to achieve this separated representation and crossing the object to be visualized as we want, we are going to take the different faces of our HDRE levels.
To implement this , we have created a "for" and we will save this faces into an array "cubemapTex" taking the faces of each level.

```
void HBRMaterial::loadCubemapTex(HDRE* hdre) {
    for (int i = 0; i < 6; i++) {
        sHDRELevel h = hdre->getLevel(i);
        cubemapTex[i]->createCubemap(h.width, h.height, (Uint8**)h.faces);
    }
}
```

When we calculate the IBL we take into account the diffuse (calculated before but explained with Punctual Lighting) and the specular component. This HDRE levels are used in the specular one. With an interpolation between the channels "r" and "g" of a 2D texture and the specular property of the material, and then multiplied by the function "getReflectionColor" in the shader, we get the whole specular part of the IBL environmental light.

```
vec4 text2 =  texture2D(u_brdf_text, vec2(ndotv,material.roughness)) ;
float a = text2.r;
float b = text2.g;
vec3 brdf_specular = material.specular*a + b;

vec3 IBL_diff = material.diffuse * getReflectionColor(reflect(v, normalize(n)),1.0);
vec3 IBL_spec = brdf_specular * getReflectionColor(reflect(v, normalize(n)), material.roughness);
vec3 IBL = IBL_diff + IBL_spec;
```

With "getReflectionColor" we pass a vector reflection "r" (in our case a reflection calculated with "v" view vector and "n", its normal) and the roughness coefficient (pre-defined). Then for each

case , depending of the roughness value of the material, we will print only the HDRE level that we want, one of the levels we see in the following image:

```
vec3 getReflectionColor(vec3 r, float roughness)
{
    float lod = roughness * 5.0;

    vec4 color;

    if(lod < 1.0) color = mix( textureCube(u_texture, r), textureCube(u_texture_prem_0, r), lod );
    else if(lod < 2.0) color = mix( textureCube(u_texture_prem_0, r), textureCube(u_texture_prem_1, r), lod - 1.0 );
    else if(lod < 3.0) color = mix( textureCube(u_texture_prem_1, r), textureCube(u_texture_prem_2, r), lod - 2.0 );
    else if(lod < 4.0) color = mix( textureCube(u_texture_prem_2, r), textureCube(u_texture_prem_3, r), lod - 3.0 );
    else if(lod < 5.0) color = mix( textureCube(u_texture_prem_3, r), textureCube(u_texture_prem_4, r), lod - 4.0 );
    else color = textureCube(u_texture_prem_4, r);

    color = pow(color, vec4(1.0/2.2));

    return color.rgb;
}
```

Punctual lighting

For the punctual lighting, we will use the formulas appearing in the theory. To account for the subsurface reflection, the BRDF contains a diffuse term. The simplest strategy is based on the "Lambertian model", which assumes that the diffuse reflectance is uniform in every direction,

$$f_{Lambert}(l, v) = \frac{\mathbf{c}_{diff}}{\pi}$$

where "cdiff" is an RGB to reflect wavelength dependency of the reflectance, the "diffuse colour" of the surface (or "diffuse albedo"). The model can be improved by taking into account the balance of specular and diffuse terms.
The specular term of the BRDF based on the microfacet theory, as we said in the introduction, has the following form:

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

· F(l,h) is the Fresnel function that is present in any shader with reflections and defines how reflections are low when viewed straight on and intensified at glancing angles. The difference is subtle at low roughness but very pronounced as roughness increases.
· D(h) is the Distribution function, that determines the overall 'smoothness' of the surface, and is most responsible for the size and shape of the specular highlight. This function also has some parameter representing "roughness" and basically tells us what concentration this important vector "h" has among all the facets pointed all over the place. The higher that concentration, the 'smoother' the surface.
· G(l,v,h) is the Geometry function, that accounts for occlusions in the surface. It has two parts, shadowing and masking. Shadowing means a microfacet is not visible to light direction and is not illuminated. Masking means a microfacet is not visible to the view direction. Both are not contributing to the reflection response.
· The denominator of this function is a normalization factor.

So after this theory explanation, to implement the punctual lighting first we have to calculate two interpolations to find diffuse and specular terms of the material:

```
material.roughness = u_roughness;
material.metallic = u_metallic;
material.diffuse = vec3(0.0)*material.metallic + u_color *(1.0 - material.metallic);
material.specular = u_color*material.metallic + f0_specular *(1.0 - material.metallic);
```

Following the instructions of the slides given in class, we have to multiply the property of metallic materials by 0, and the opposite point with the albedo color (non-metallic) to find diffuse material property. The inverse values to find specular material property.
Then we calculate the Geometry function “G”, the Distribution function values “D”, the Fresnel function “F” and the lambert equation, so we sum up this last one with the other micro-facet equation results as explained before.

Adding textures map

The next step is to add textures for the roughness, metalness, color and normal vectors. The first three mentioned textures are applied directly. That means, we upload a texture previously generated by an 3D artist to the shader, then using the texture coordinates “UV” we get the RGB value from the texture.

This RGB value is used to pass 3D vectors, but on some cases as the roughness and metalness we only need one channel, for example, the “R” channel. This is the implementation for the color, roughness and metalness.

```
vec4 color = texture2D(u_albedo, v_uv);
```

```
aux = texture2D(u_rough_map, v_uv);
material.roughness = aux.r;

aux = texture2D(u_metal_map, v_uv);
material.metallic = aux.r;
```

What about the normal map? First of all we have to take into account that textures return values between [0,1], but the normal vector should be between [-1, 1]. So, after we get the RGB value we extend the range of the value.

```
vec4 norm = texture2D(u_normal_map, v_uv);
vec4 normal4 = normalize(norm*2.0-1.0);
```

But these values only tell us how the normal vector is deviated on each vertex, but is not oriented on the mesh. For example, if the normal map value equals (0,0.707,0.707) means we have the normal vector deviated 45º on the x axis, so what we will do is to rotate the normal vector of the mesh on that vertex 45º.

To do that we need to compute the angle of the normal vector on each axis:

```
vec3 getAngle( vec3 v ){

    vec3 angle;
    angle.x = atan(v.y/v.z);
    angle.y = atan(v.x/v.z);
    angle.z = atan(v.y/v.z);

    return angle;
}
```

After computing the rotation on each axis, we create the 3 rotation matrix and then we apply it on the normalized Normal vector.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

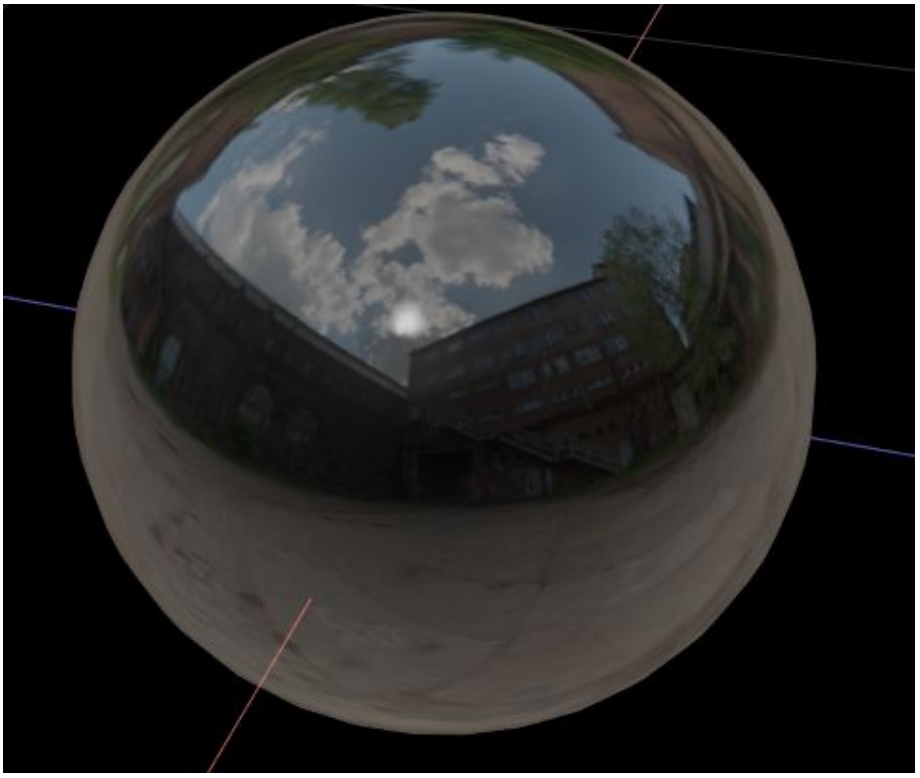
$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

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

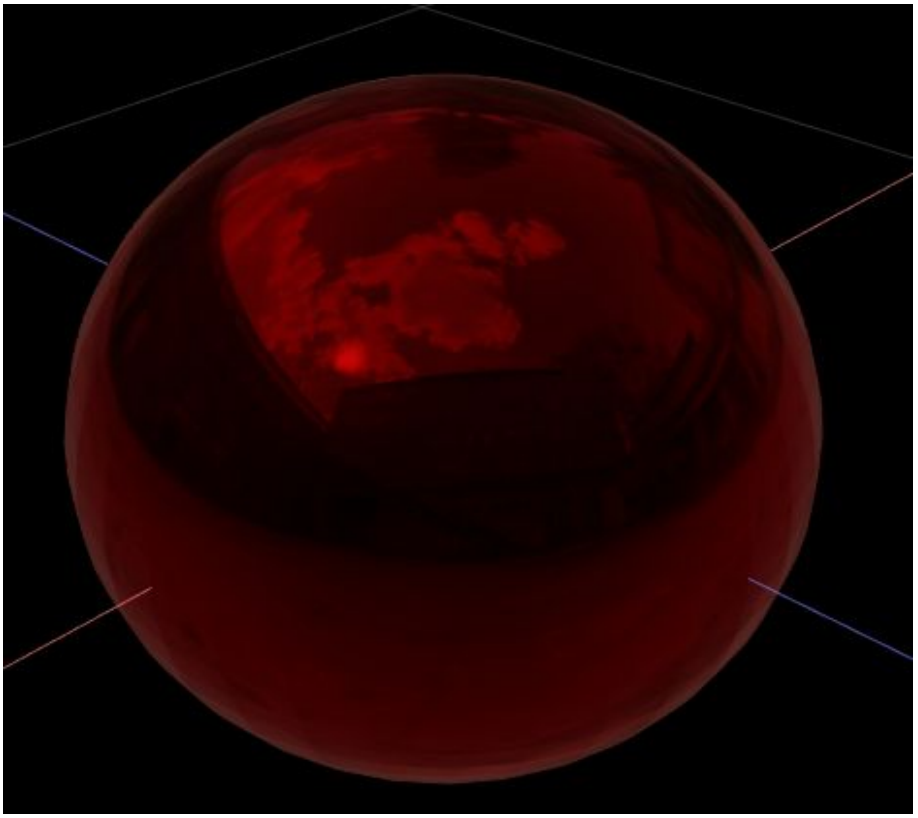
```
vec3 angle_map = getAngle(normal4.xyz);
mat3 rotx;
rotx[0] = vec3(1,0,0);
rotx[1] = vec3(0, cos(angle_map.x), sin(angle_map.x));
rotx[2] = vec3(0, -sin(angle_map.x), cos(angle_map.x));

mat3 roty;
roty[0] = vec3(cos(angle_map.y),0,-sin(angle_map.y));
roty[1] = vec3(0, 1, 0);
roty[2] = vec3(sin(angle_map.y), 0, cos(angle_map.y));

mat3 rotz;
rotz[0] = vec3(cos(angle_map.z),sin(angle_map.z),0);
rotz[1] = vec3(-sin(angle_map.z), cos(angle_map.z), 0);
rotz[2] = vec3(0, 0, 1);

vec3 n2 = rotz*roty*rotx*normalize(v_normal);
```

Now we have all the normal map added to the “natural” normal vectors of the mesh.

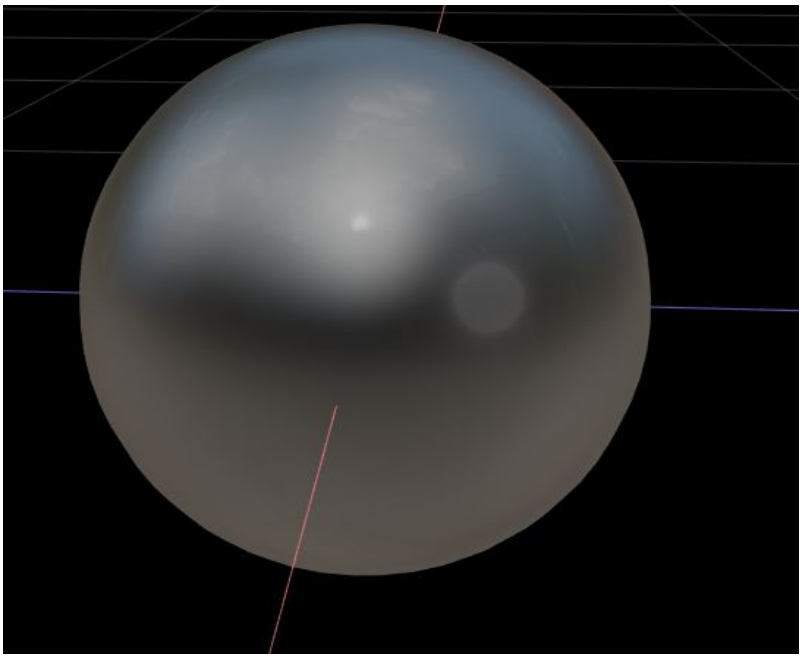
III. Results



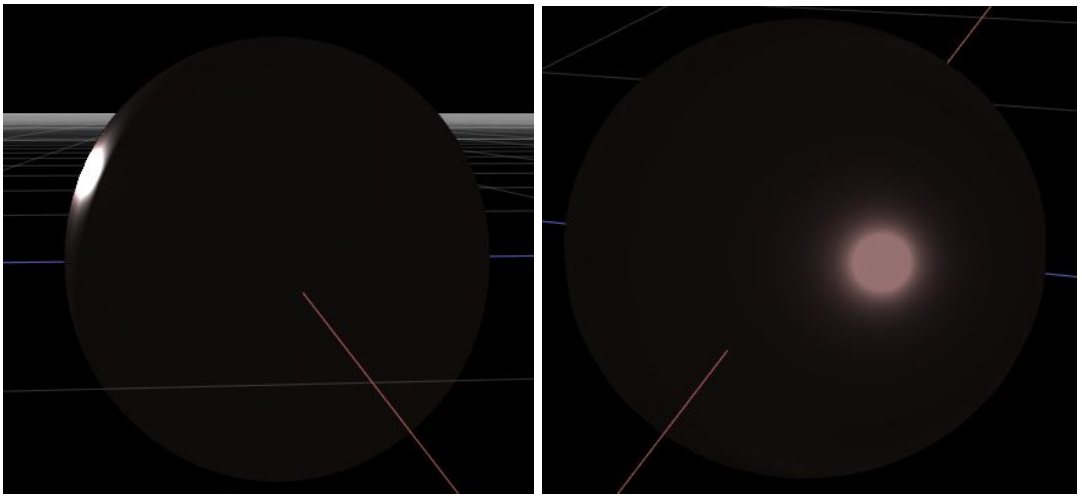
Metallic=1, roughness = 0, u_color (255,255,255)



Metallic=1, roughness = 0, u_color (176,0,0)

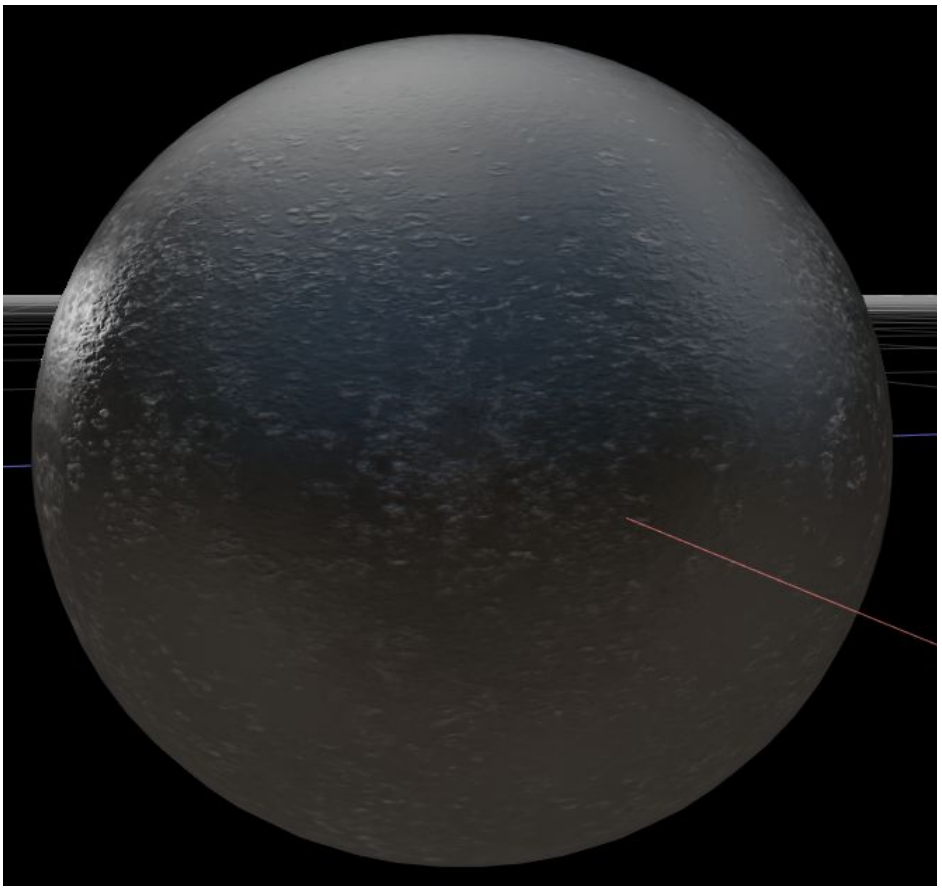


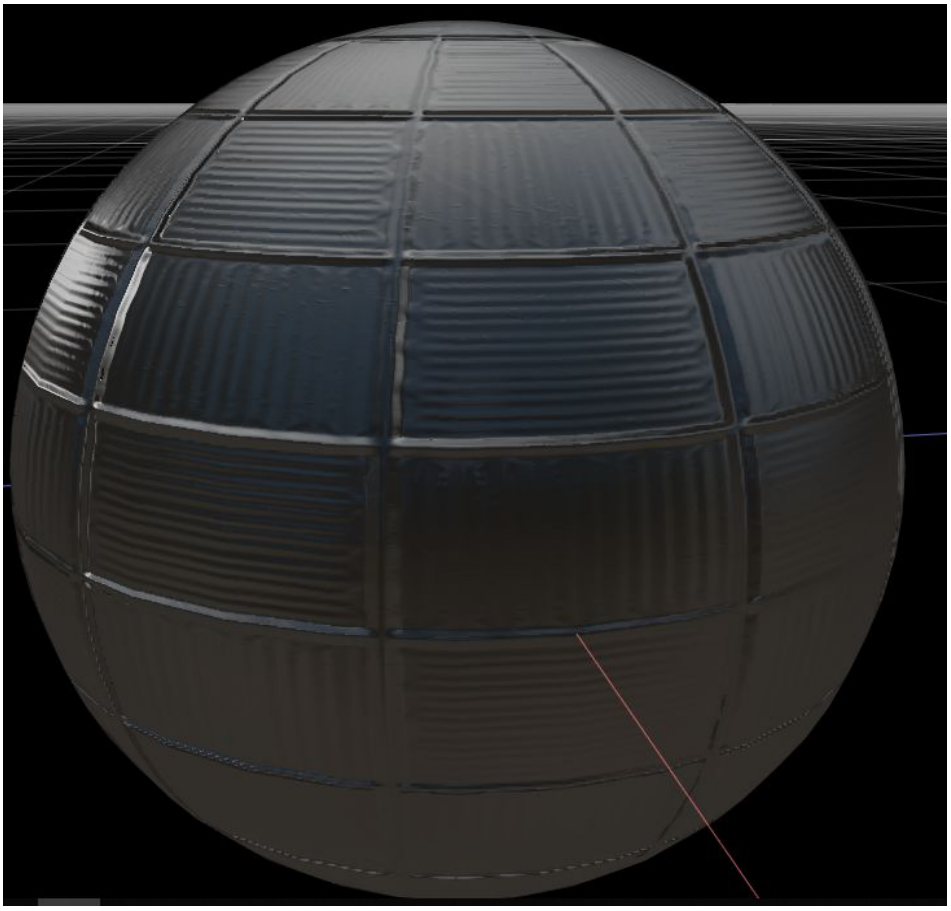
Rougness = 0.18, metalness 0.865



Only punctual light, Metalness=0.715, Roughness = 0.41, Color(169,126,126)

ADDING TEXTURES

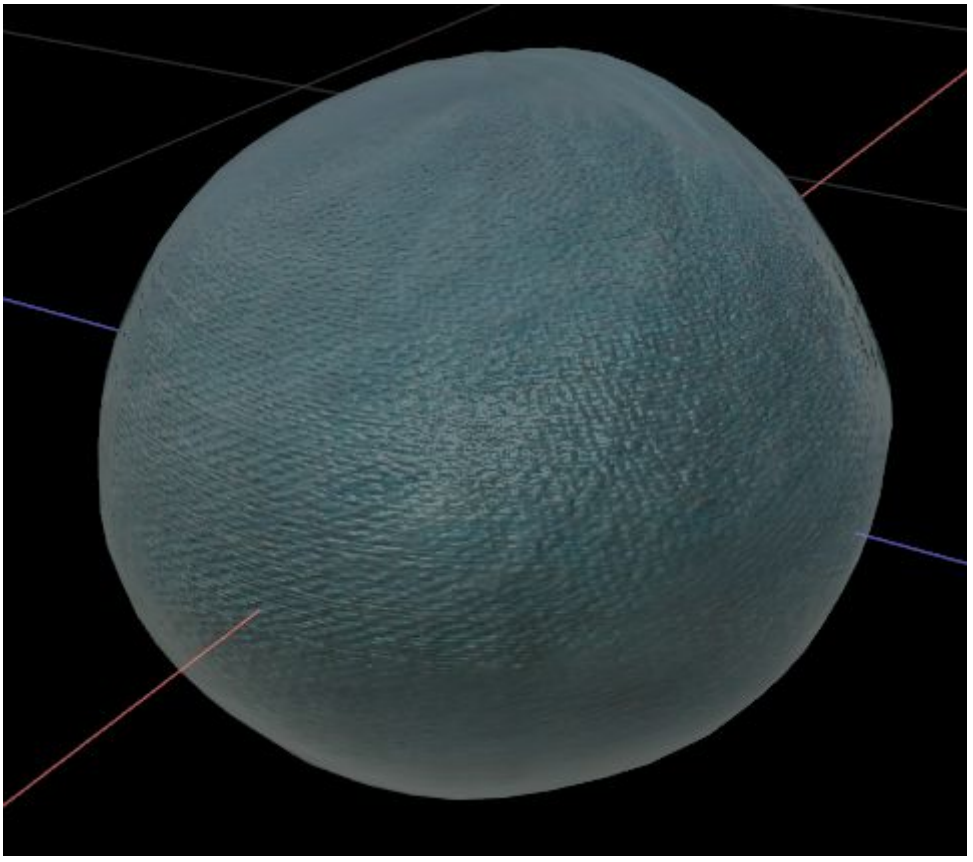




EXTRA

We added also a height map, which is implemented by summing to each vertex (on the vertex shader) an extra height in the direction of the normal. This height can be scaled by an extra factor.

```
//calcule the vertex in object space
vec4 aux = texture2D(u_height_map, a_uv);
v_position = a_vertex+normalize(v_normal)*aux.x*0.2;
```



IV. References

[1] A Reflectance Model for Computer Graphics
[2] Microfacet Models for Refraction through Rough Surfaces
[3] Microfacet Models for reflection and refraction - slides
[4] Advanced-CG-Theory notes of Advanced Visualization 2019 course at UPF
[5] PBR I and II slides of Advanced Visualization 2019 course at UPF
[6] PBR A4 slides of Advanced Visualization 2019 course at UPF
[7]https://webglstudio.org/users/arodriguez/projects/HDR4EU/