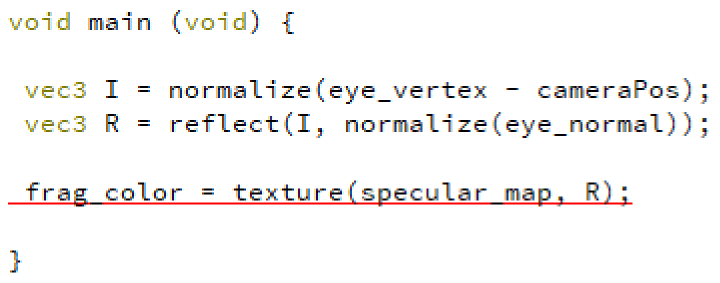
FRR - PHYSICALLY BASED RENDERING

Diego Campos Milian

March 27, 2020

1 Reflection shader

The first implemented shade for this project it's a simple reflection using a cubemap texture. It's execution it's quite simple, just obtain the incident vector I through the difference between the fragment world position and the camera position and reflect it upon the normal to obtain the reflected vector$R$ which with we can obtain the fragment color with the given cube texture.



```
void main (void) {

 vec3 I = normalize(eye_vertex - cameraPos);
 vec3 R = reflect(I, normalize(eye_normal));

 frag_color = texture(specular_map, R);

}
```

Figure 1: Reflection fragment shader

2 Image based lightning

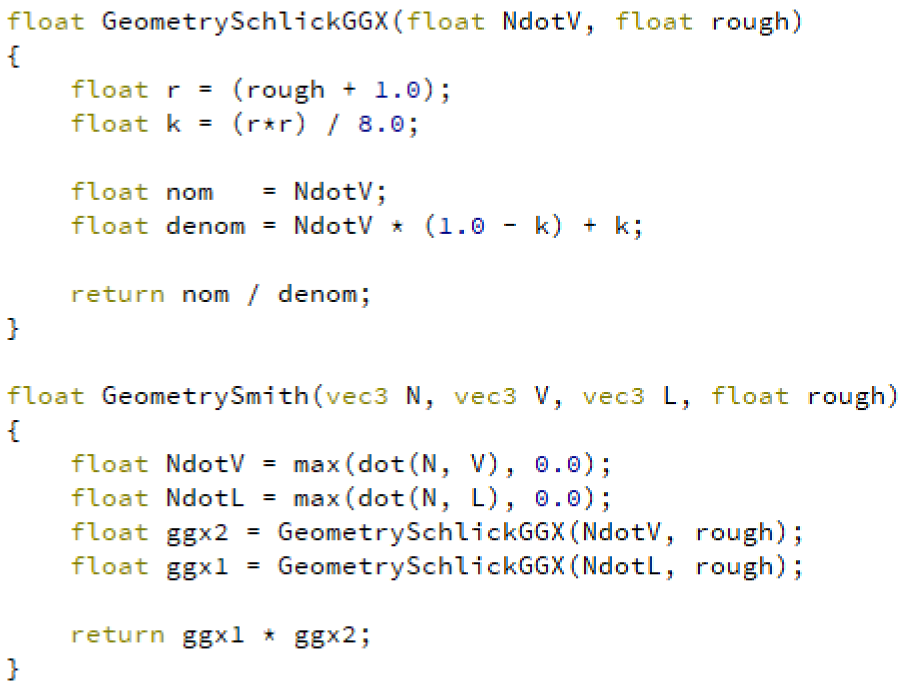
The second part consist in the implementation of an image based lightning (IGL) shader based on the BRDF microfacet theory.

2.1 BRDF equations

2.1.1 Geometry occlusion

There are implemented two geometry occlusion functions G.

1. Implicit: Where the G in the equation is considered equal to the denominator part, and therefore $G_{implicit}(n, l, v) = n \cdot v * n \cdot l$
2. Shlick-GGX: Wich incorporates roughness to it's calculation and it's given by $G_{SchlickGGX}(n, v, k) = \frac{n \cdot v}{(n \cdot v)(1-k)+k}$. Which to take into account both occlusion (view direction) and shadowing (light direction) are used following the Smith's method given by $G(n, l, v, k) = G_{SchlickGGX}(n, v, k) * G_{SchlickGGX}(n, l, k)$.

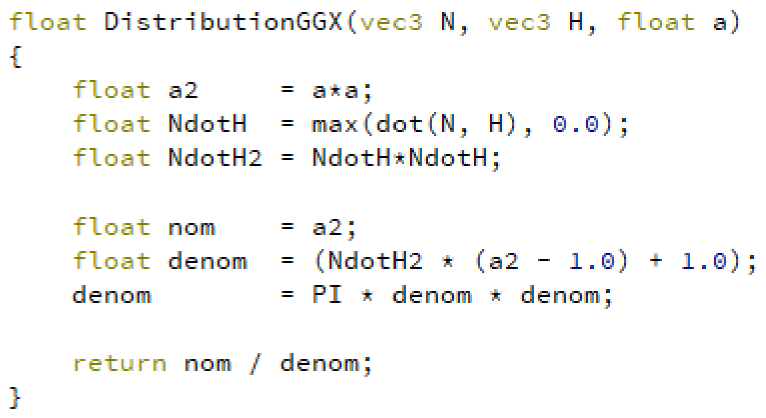


```
float GeometrySchlickGGX(float NdotV, float rough)
{
    float r = (rough + 1.0);
    float k = (r*r) / 8.0;

    float nom   = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return nom / denom;
}

float GeometrySmith(vec3 N, vec3 V, vec3 L, float rough)
{
    float NdotV = max(dot(N, V), 0.0);
    float NdotL = max(dot(N, L), 0.0);
    float ggx2 = GeometrySchlickGGX(NdotV, rough);
    float ggx1 = GeometrySchlickGGX(NdotL, rough);

    return ggx1 * ggx2;
}
```

Figure 2: Geometry Schlick-GGX function in the fragment shader

2.1.2 Normal distribution

For the normal distribution factor i use the Trowbridge-Reitz GGX that's is defined as $ND_{TRGGX}(n,h,\alpha) = \frac{\alpha^2}{\pi((n\cdot h)^2)(\alpha^2-1)+1)^2}$.

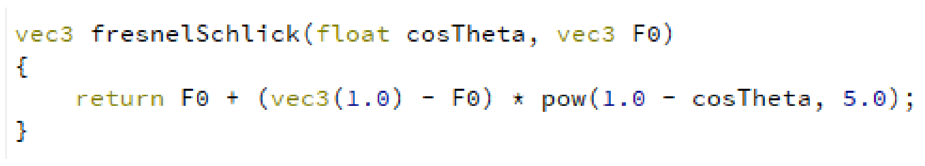


```
float DistributionGGX(vec3 N, vec3 H, float a)
{
    float a2     = a*a;
    float NdotH  = max(dot(N, H), 0.0);
    float NdotH2 = NdotH*NdotH;

    float nom    = a2;
    float denom  = (NdotH2 * (a2 - 1.0) + 1.0);
    denom        = PI * denom * denom;

    return nom / denom;
}
```

Figure 3: Trowbridge-Reitz GGX normal distribution function in the fragment shader.

2.1.3 Fresnel

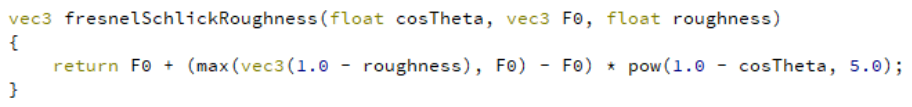
For the fresnel reflectance factor I'm using the well known Schilck approximation that is $F_{Schlick}(h, v, F_0) = F_0 + (1 - F_0)(1 - (h \cdot v))^5$.



```
vec3 fresnelSchlick(float cosTheta, vec3 F0)
{
    return F0 + (vec3(1.0) - F0) * pow(1.0 - cosTheta, 5.0);
}
```

Figure 4: Schlick fresnel factor in the fragment shader.

However, as explain in [1] as we currently aren't taking any roughness into account, the surface's reflective ratio will tend to end up relatively high. Indirect light follows the same properties of direct light so we expect rougher surfaces to reflect less strongly on the surface edges. To alleviate this issue we can add roughness into the equation as shown in Figure 5.

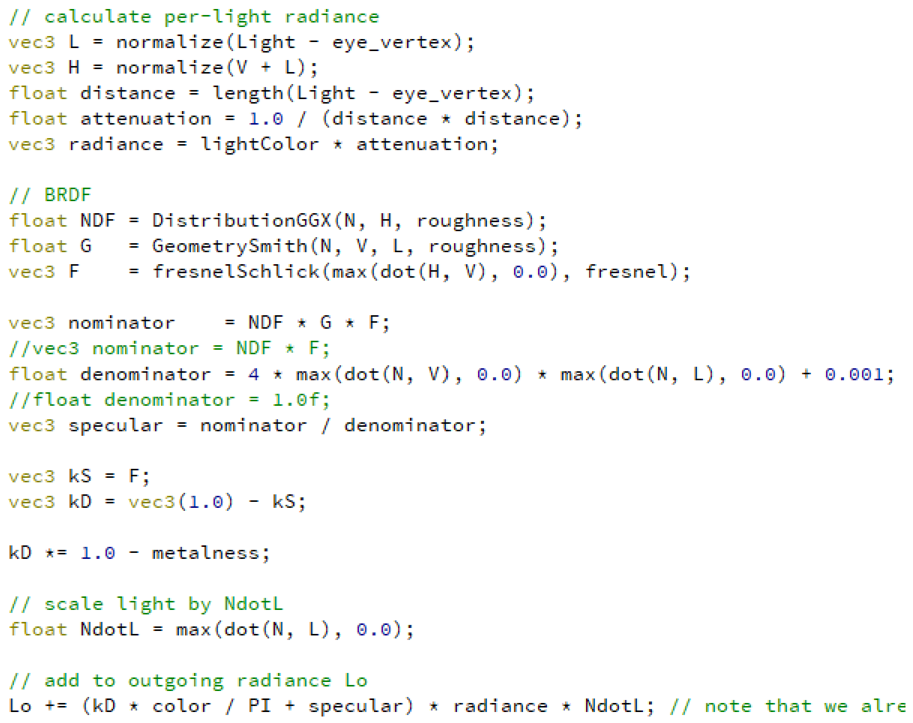


```
vec3 fresnelSchlickRoughness(float cosTheta, vec3 F0, float roughness)
{
    return F0 + (max(vec3(1.0 - roughness), F0) - F0) * pow(1.0 - cosTheta, 5.0);
}
```

Figure 5: Schlick fresnel with roughness factor in the fragment shader.

2.2 Reflectance

The IBL implementation is basically form of 3 parts, the first one being the reflectance of the materials, that is how direct lights will affect it's appearance. This part follows the Cook-Torrance BRDF approach similar to the specular term, therefor for each light in the scene will calculate it's contribution as $\frac{DFG}{4(n \cdot v)(n \cdot l)}$.

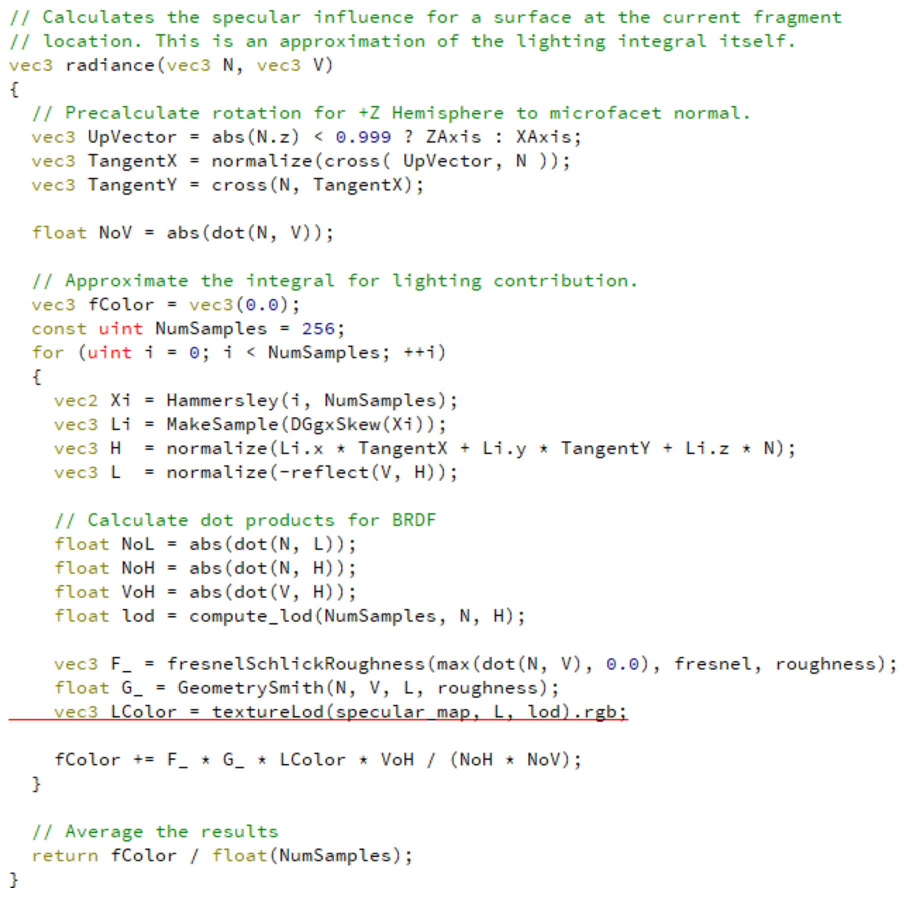


```
// calculate per-light radiance
vec3 L = normalize(Light - eye_vertex);
vec3 H = normalize(V + L);
float distance = length(Light - eye_vertex);
float attenuation = 1.0 / (distance * distance);
vec3 radiance = lightColor * attenuation;

// BRDF
float NDF = DistributionGGX(N, H, roughness);
float G   = GeometrySmith(N, V, L, roughness);
vec3 F    = fresnelSchlick(max(dot(H, V), 0.0), fresnel);

vec3 nominator    = NDF * G * F;
//vec3 nominator = NDF * F;
float denominator = 4 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0) + 0.001;
//float denominator = 1.0f;
vec3 specular = nominator / denominator;

vec3 kS = F;
vec3 kD = vec3(1.0) - kS;

kD *= 1.0 - metalness;

// scale light by NdotL
float NdotL = max(dot(N, L), 0.0);

// add to outgoing radiance Lo
Lo += (kD * color / PI + specular) * radiance * NdotL; // note that we alre
```

Figure 6: Reflectance calculus per light in the fragment shader.

2.3 Specular contribution

For calculating the material specularity we follow a similar approach as the reflectance, however instead of using the well known Epic technique of prefiltering the radiance cubemap we do the calculus directly at execution as explain in [2]. Note that that means that we have to compute the equivalent part of the LUT texture every frame for each fragment which reduces the framerate considerably for models with a high vertex count, however we can reduce this by reducing the number of samples we make at the cost of some image fidelity.

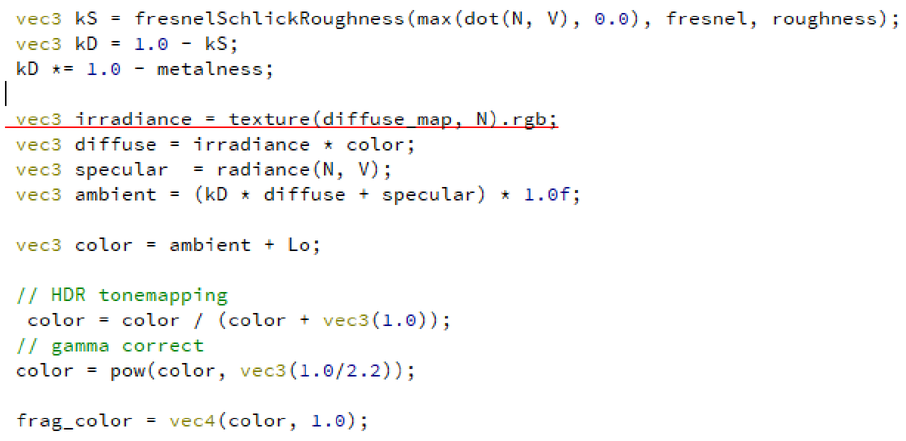


```glsl
// Calculates the specular influence for a surface at the current fragment
// location. This is an approximation of the lighting integral itself.
vec3 radiance(vec3 N, vec3 V)
{
  // Precalculate rotation for +Z Hemisphere to microfacet normal.
  vec3 UpVector = abs(N.z) < 0.999 ? ZAxis : XAxis;
  vec3 TangentX = normalize(cross( UpVector, N ));
  vec3 TangentY = cross(N, TangentX);

  float NoV = abs(dot(N, V));

  // Approximate the integral for lighting contribution.
  vec3 fColor = vec3(0.0);
  const uint NumSamples = 256;
  for (uint i = 0; i < NumSamples; ++i)
  {
    vec2 Xi = Hammersley(i, NumSamples);
    vec3 Li = MakeSample(DGgxSkew(Xi));
    vec3 H  = normalize(Li.x * TangentX + Li.y * TangentY + Li.z * N);
    vec3 L  = normalize(-reflect(V, H));

    // Calculate dot products for BRDF
    float NoL = abs(dot(N, L));
    float NoH = abs(dot(N, H));
    float VoH = abs(dot(V, H));
    float lod = compute_lod(NumSamples, N, H);

    vec3 F_ = fresnelSchlickRoughness(max(dot(N, V), 0.0), fresnel, roughness);
    float G_ = GeometrySmith(N, V, L, roughness);
    vec3 LColor = textureLod(specular_map, L, lod).rgb;

    fColor += F_ * G_ * LColor * VoH / (NoH * NoV);
  }

  // Average the results
  return fColor / float(NumSamples);
}
```

Figure 7: Specular term calculus in the fragment shader.

2.4 Diffuse contribution

Finally, the diffuse contribution is given by the precomputed cubemap as shown. And then added to the specular and reflectance lighting to obtain the final color.



```
vec3 kS = fresnelSchlickRoughness(max(dot(N, V), 0.0), fresnel, roughness);
vec3 kD = 1.0 - kS;
kD *= 1.0 - metalness;

vec3 irradiance = texture(diffuse_map, N).rgb;
vec3 diffuse = irradiance * color;
vec3 specular  = radiance(N, V);
vec3 ambient = (kD * diffuse + specular) * 1.0f;

vec3 color = ambient + Lo;

// HDR tonemapping
 color = color / (color + vec3(1.0));
// gamma correct
color = pow(color, vec3(1.0/2.2));

frag_color = vec4(color, 1.0);
```

Figure 8: Diffuse term and final result in the fragment shader.

References

[1] Joey de Vries. IBL Physically Based Rendering. https://learnopengl.com/PBR/Theory. Last access: March 27, 2020.

[2] Trent Reed. Physically Based Shading and Image Based Lighting. https://www.trentreed.net/blog/physically-based-shading-and-image-based-lighting/. Last access: March 27, 2020.