Project 3 : Cook-Torrance

ID/ Name : 20174089/ 윤석원

1. obj file load

```
Model ourModel(FileSystem::getPath("../resources/backpack/backpack.obj"));
pbrShader.setFloat("metallic", (float)2 / (float)7);
pbrShader.setFloat("roughness", glm::clamp((float)4 / (float)7, 0.05f, 1.0f));
glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3( (float)(3 - (7 / 2)) * 2.5,  (float)(3 - (7 / 2)) * 2.5, -2.0f ));
pbrShader.setMat4("model", model);
ourModel.Draw(pbrShader);
```

- texture map이 있는 backpack.obj를 assimp 라이브러를 통해 로드한다.
- metalic, roughness의 특성값을 주고 pbrshader를 통해 모델을 화면에 그린다.

2. brdf.fs

```
// efficient VanDerCorpus calculation.
float RadicalInverse_VdC(uint bits)
{
     bits = (bits << 16u) | (bits >> 16u);
     bits = ((bits & 0x55555555u) << 1u) | ((bits & 0xAAAAAAAAu) >> 1u);
     bits = ((bits & 0x33333333u) << 2u) | ((bits & 0xCCCCCCCCu) >> 2u);
     bits = ((bits & 0x0F0F0F0Fu) << 4u) | ((bits & 0xF0F0F0F0u) >> 4u);
     bits = ((bits & 0x00FF00FFu) << 8u) | ((bits & 0xFF00FF00u) >> 8u);
     return float(bits) * 2.3283064365386963e-10; // / 0x100000000
}
vec2 Hammersley(uint i, uint N)
{
          return vec2(float(i)/float(N), RadicalInverse_VdC(i));
}
```

- brdf를 구에 기반하여 정의하는 방법으로 VanDerCorpus calculation을 사용한다.
- 구에 uniform하게 point들을 샘플링 하기 위해 아래 수식에 근거하여 좌표가 정해진다. 따라서 2진 bit로 point set을 정의한다.

$$\Phi_2(i) = \frac{a_0}{2} + \frac{a_1}{2^2} + \ldots + \frac{a_r}{2^{r+1}}$$

```
vec3 ImportanceSampleGGX(vec2 Xi, vec3 N, float roughness)
{
          float a = roughness*roughness;

          float phi = 2.0 * PI * Xi.x;
          float cosTheta = sqrt((1.0 - Xi.y) / (1.0 + (a*a - 1.0) * Xi.y));
          float sinTheta = sqrt(1.0 - cosTheta*cosTheta);
```

```
        // from spherical coordinates to cartesian coordinates - halfway vector
        vec3 H;
        H.x = cos(phi) * sinTheta;
        H.y = sin(phi) * sinTheta;
        H.z = cosTheta;

        // from tangent-space H vector to world-space sample vector
        vec3 up          = abs(N.z) < 0.999 ? vec3(0.0, 0.0, 1.0) : vec3(1.0, 0.0, 0.0);
        vec3 tangent   = normalize(cross(up, N));
        vec3 bitangent = cross(N, tangent);

        vec3 sampleVec = tangent * H.x + bitangent * H.y + N * H.z;
        return normalize(sampleVec);
}
// ----------------------------------------------------------------------------
```

- sampling한 지점들에 대해서 normal vector와 거칠기 정도를 매개변수로 받아 H를 구한다.

```
float GeometrySchlickGGX(float NdotV, float roughness)
{
    // note that we use a different k for IBL
    float a = roughness;
    float k = (a * a) / 2.0;

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

    return nom / denom;
}
```

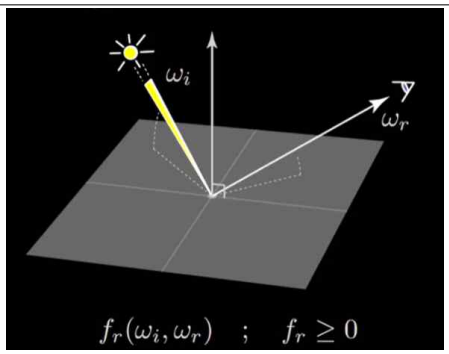
- Geometry 특성을 계산한다.

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

    return ggx1 * ggx2;
}
```

- 각 sampling 지점에서 ggx1, ggx2를 구해서 곱한 값을 반환한다.

```
// ----------------------------------------------------------------------------
vec2 IntegrateBRDF(float NdotV, float roughness)
{
    vec3 V;
    V.x = sqrt(1.0 - NdotV*NdotV);
    V.y = 0.0;
    V.z = NdotV;

    float A = 0.0;
    float B = 0.0;

    vec3 N = vec3(0.0, 0.0, 1.0);

    const uint SAMPLE_COUNT = 1024u;
    for(uint i = 0u; i < SAMPLE_COUNT; ++i)
    {
        // generates a sample vector that's biased towards the
        // preferred alignment direction (importance sampling).
```

```
        vec2 Xi = Hammersley(i, SAMPLE_COUNT);
        vec3 H = ImportanceSampleGGX(Xi, N, roughness);
        vec3 L = normalize(2.0 * dot(V, H) * H - V);

        float NdotL = max(L.z, 0.0);
        float NdotH = max(H.z, 0.0);
        float VdotH = max(dot(V, H), 0.0);

        if(NdotL > 0.0)
        {
            float G = GeometrySmith(N, V, L, roughness);
            float G_Vis = (G * VdotH) / (NdotH * NdotV);
            float Fc = pow(1.0 - VdotH, 5.0);

            A += (1.0 - Fc) * G_Vis;
            B += Fc * G_Vis;
        }
    }
    A /= float(SAMPLE_COUNT);
    B /= float(SAMPLE_COUNT);
    return vec2(A, B);
}
void main()
{
    vec2 integratedBRDF = IntegrateBRDF(TexCoords.x, TexCoords.y);
    FragColor = integratedBRDF;
}
```



- texture map에 정의된 정보를 바탕으로 wi로 들어온 빛에 대해 wr로 얼마만큼 반사 될것인지를 나타내는 brdf를 전부합하여 fragment의 color를 결정한다.

2. pbr.fs

```
void main()
{
    vec3 N = Normal;
    vec3 V = normalize(camPos - WorldPos);
    vec3 R = reflect(-V, N);

    // calculate reflectance at normal incidence; if dia-electric (like plastic) use F0
    // of 0.04 and if it's a metal, use the albedo color as F0 (metallic workflow)
    vec3 F0 = vec3(0.04);
    F0 = mix(F0, albedo, metallic);

    // reflectance equation
    vec3 Lo = vec3(0.0);
```

```
    for(int i = 0; i < 4; ++i)
    {
        // calculate per-light radiance
        vec3 L = normalize(lightPositions[i] - WorldPos);
        vec3 H = normalize(V + L);
        float distance = length(lightPositions[i] - WorldPos);
        float attenuation = 1.0 / (distance * distance);
        vec3 radiance = lightColors[i] * attenuation;

        // Cook-Torrance BRDF
        float NDF = DistributionGGX(N, H, roughness);
        float G   = GeometrySmith(N, V, L, roughness);
        vec3 F    = fresnelSchlick(max(dot(H, V), 0.0), F0);

        vec3 nominator    = NDF * G * F;
        float denominator = 4 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0) + 0.001; // 0.001 to prevent divide by zero.
        vec3 specular = nominator / denominator;

         // kS is equal to Fresnel
        vec3 kS = F;
        // for energy conservation, the diffuse and specular light can't
        // be above 1.0 (unless the surface emits light); to preserve this
        // relationship the diffuse component (kD) should equal 1.0 - kS.
        vec3 kD = vec3(1.0) - kS;
        // multiply kD by the inverse metalness such that only non-metals
        // have diffuse lighting, or a linear blend if partly metal (pure metals
        // have no diffuse light).
        kD *= 1.0 - metallic;

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

        // add to outgoing radiance Lo
        Lo += (kD * albedo / PI + specular) * radiance * NdotL; // note that we already multiplied the BRDF by the Fresnel (kS) so we won't multiply by kS again
    }

    // ambient lighting (we now use IBL as the ambient term)
    vec3 F = fresnelSchlickRoughness(max(dot(N, V), 0.0), F0, roughness);

    vec3 kS = F;
    vec3 kD = 1.0 - kS;
    kD *= 1.0 - metallic;

    vec3 irradiance = texture(irradianceMap, N).rgb;
    vec3 diffuse      = irradiance * albedo;

    // sample both the pre-filter map and the BRDF lut and combine them together as per the Split-Sum approximation to get the IBL specular part.
    const float MAX_REFLECTION_LOD = 4.0;
    vec3 prefilteredColor = textureLod(prefilterMap, R,  roughness * MAX_REFLECTION_LOD).rgb;
    vec2 brdf  = texture(brdfLUT, vec2(max(dot(N, V), 0.0), roughness)).rg;
    vec3 specular = prefilteredColor * (F * brdf.x + brdf.y);

    vec3 ambient = (kD * diffuse + specular) * ao;
```

```
    vec3 color = ambient + Lo;

    // HDR tonemapping
    color = color / (color + vec3(1.0));

    color = pow(color, vec3(1.0/2.2));
    FragColor = texture(texture_diffuse1, TexCoords);
}
```

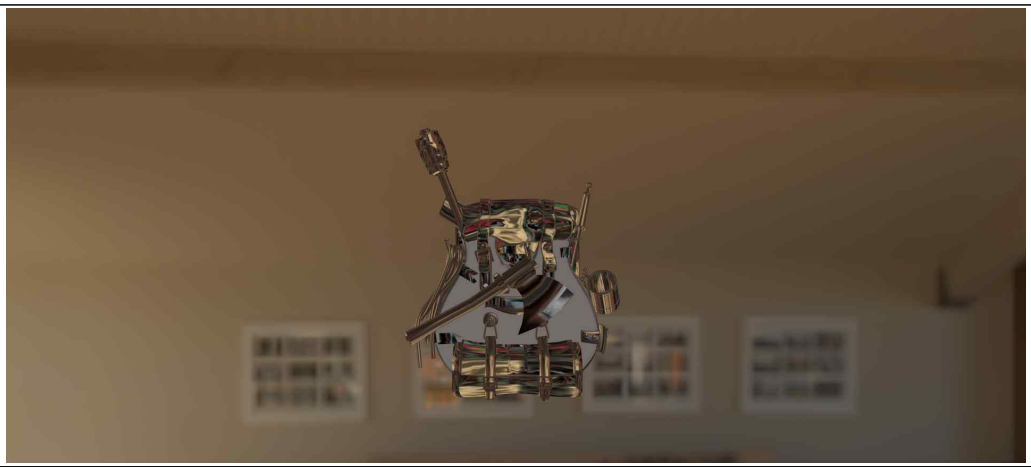
- 실제 물리법칙에 기반하여 cook-torrance brdf를 계산한다.
- 물체의 surface가 micro geometry로 구성되어 있다고 가정하여 확률에 근거해서 반사율을 구한다.

"Specular" term (really directional diffuse)

$$f_s(x, L, V) = \frac{FGD}{4(N \cdot L)(N \cdot V)}$$

- model의 각도에 따라 달라지는 에너지 분포를 함수로 나타낸다.

8. 결과 확인



- environment가 반사되는 모습을 더 보고싶어서 metallic 특성 값을 많이 주었다.
- 조명에 따라서 굉장히 사실적인 묘사가 구현되었다. 그에 따라 rendering 시간도 비교적 많이 걸렸다.