Project report for the CG 100433 course

Project Title

组队战神龙

Team member

1851964 武澳奇
1853849 范正源
1854094 汪冰海
1951247 钟伊凡
1952339 张馨月
1953597 刘云帆

Project Title

海上温馨小屋

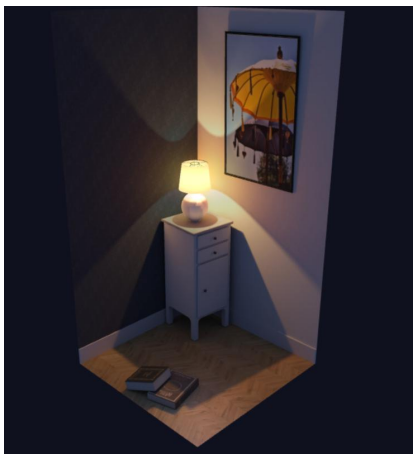
Abstract

我们的项目目标是致力于用多项技术渲染一个真实的场景
技术范围包括:

1. 镜面渲染（门户渲染、裁剪平面、模板和深度测试）
2. 阴影渲染
3. 基于物理的材质渲染
4. PBR 漫反射辐照（动态环境贴图）
5. 水体渲染
6. Gltf 格式的导入和使用

Motivation

PBR 现在得到广泛应用，小组成员通过查找图形学资料，了解了 PBR 相关内容，并在网上找到了很多 PBR 的精彩实现，想要学习 PBR 技术从而提高图形学编程水平，于是思考制作一个 PBR 的场景应用来展现材质和光照。



The Goal of the project

1、实现对室内物品的 PBR 渲染，产生较为真实的材质效果。
2、实现对室外海洋的大型水体渲染，水体为动态水体，可以产生真实的光照效果。
3、使用 IBL 提供的环境光照，将环境光源和水体进行交互产生真实的水面效果。
4、实现室内物品模型。
5、实现实时光照渲染的画面效果
6、良好的摄像机移动方式，给予观察者良好的效果展示

The Scope of the project

1、不做物理碰撞检测
2、不实现与场景物品的交互
3、不实现光线追踪
4、实现 PBR 模型渲染、PBR 水体渲染
5、追求对场景的真实再现
6、场景范围有限，只能在该空间内移动
7、天空及远景是固定贴图不可交互

Involved CG techniques

镜面渲染技术：

- 裁剪平面：使用裁剪平面剪去镜子和镜像摄像机直接不该出现的内容
- 模板测试：通过模板测试将镜像准确的只显示在镜子处
- 深度测试：通过深度测试正确设置镜像的深度
- 帧缓冲：通过使用帧缓冲，我们能够为物体的 6 个不同角度创建出场景的纹理，并在每个渲染迭代中将它们储存到一个立方体贴图中。

阴影渲染技术：

- Shadow Mapping：使用阴影映射的方法渲染了定向光源阴影
- PCF(percentage-closer filtering)：通过从深度贴图中多次采样，弱化锯齿块和硬边，产生柔和阴影

水体渲染及超大海面渲染技术：

- 海面模拟：FFT 海面，用海洋统计学公式生成 Phillips 频谱，计算高度频谱和水平偏移频谱，经快速傅里叶逆变换（IFFT）得到偏移图，计算出 Gerstner 波叠加的海面
- 实例化（Instancing）：使用实例化来加快大面积海面的渲染
- 混合（Blending）：使用混合来实现半透明的海面

PBR：

- 基于物理的材质渲染和光照：通过使用金属度贴图、粗糙度贴图、漫反射遮蔽贴图、法线贴图、基本颜色贴图，使用 Cook-Torrance 反射率方程计算光照，综合以上贴图信息得出接近真实物理世界的效果。反射率方程如下：

$$L_o(p,\omega_o) = \int_{\Omega}(k_d\frac{c}{\pi} + k_s\frac{DFG}{4(\omega_o\cdot n)(\omega_i\cdot n)})L_i(p,\omega_i)n\cdot\omega_i d\omega_i$$

- 基于图像的光照（IBL 技术）：通过动态环境贴图的方式获取物体的环境光照，使用图像的像素点，而非单纯的颜色作为光源。对环境贴图进行预滤波获得漫反射辐照度，可以

大大减少 GPU 计算的复杂度。即将上文中反射率方程拆开为镜面部分和漫反射部分：

$$L_o(p,\omega_o) = \int_\Omega (k_d\frac{c}{\pi})L_i(p,\omega_i)n\cdot\omega_i d\omega_i + \int_\Omega (k_s\frac{DFG}{4(\omega_o\cdot n)(\omega_i\cdot n)})L_i(p,\omega_i)n\cdot\omega_i d\omega_i$$

左半部分对应于漫反射部分，预先计算L_i，即漫反射辐照度。

模型导入和修改：

- 模型导入统一使用 gltf 格式，并且具有 PBR 材质的物理模型，我们使用 blender 对模型进行修改。

Project contents

1. 实现了较为真实的镜像效果，产生了动态环境贴图。
2. 实现了定向光源的阴影，阴影的明暗度较为真实地模拟了实际场景。
3. 实现了 FFT 海面，较为真实地模拟了海面波涛起伏的效果。
4. 实现了 PBR 渲染，并结合基于图像的光照（IBL）技术，渲染出了真实的物体以及环境光照的感觉。

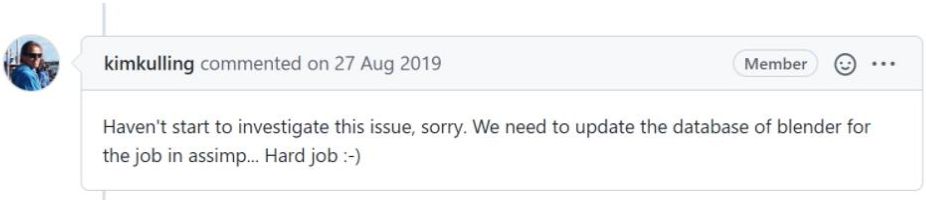
Implementation

模型导入

导入模型的时候只成功导入了 gltf 格式的模型，而另一个很流行的模型是 blend 格式的模型。但是 blend 的导入一直不成功，并会给出以下报错：

```
ERROR::ASSIMP::BLEND: Expected at least one object with no parent
```

在查找了大量的参考之后，最终在 github 上 assimp 库的 issue 里面找到了



这是一个 assimp 官方库中还没有解决的问题，所以也没有更好的办法解决，只能选择 gltf 格式的模型

在导入模型的过程中发现程序并不能在独显上运行，而是默认在 CPU 的集显上运行。后来发现在程序运行过程中添加以下代码即可解决：

```
extern "C" __declspec(dllexport) DWORD NvOptimusEnablement = 0x00000001;
```

当然这句话只对 N 卡有效，A 卡可以添加

```
extern "c"__declspec(dllexport) int AmdPowerXpressRequestHighPerformance =
0x00000001;
```

阴影渲染

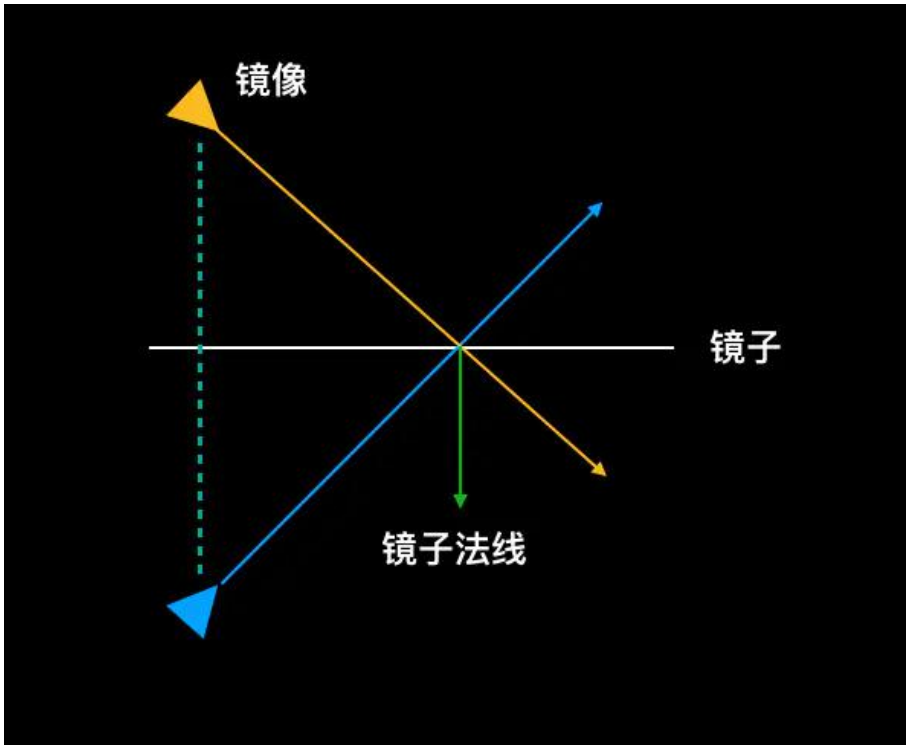
实现逻辑是先从光源的视角渲染一遍整个场景，得到深度贴图，然后像往常一样渲染场景，使用生成的深度贴图来计算片段是否在阴影之中。
由于本组项目渲染的是静态场景，因此从光源角度看到的物体深度是不变的。基于此，我们在渲染循环之前从光源角度渲染了深度贴图，在后面只需要使用即可，从而减少了接近一倍的开销，提升了效率。具体实现时，要提前定义深度贴图的帧缓冲对象和纹理对象，将纹理绑定到帧缓冲上。这样，从光源角度渲染这一遍的时候，就会把深度值更新到纹理对象 depthMap 中。随后在渲染循环中，就可以从 depthMap 纹理对象取得深度值。
渲染循环中，要向 shader 传入将物体变换到光源视角的 view 和 projection 矩阵，以及深度贴图。片段着色器中，基于深度贴图，使用 PCF 方法和我们自己设计的非线性函数，计算当前片段的阴影强度，使用这个阴影强度结合本来的颜色得到最终的颜色。

以下是片段着色器中计算阴影强度的函数：

```
float ShadowCalculation(vec4 fragPosLightSpace, vec3 normal, vec3 lightDir)
{
    // 执行透视除法
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // 变换到[0,1]的范围
    projCoords = projCoords * 0.5 + 0.5;
    // 取得最近点的深度(使用[0,1]范围下的 fragPosLight 当坐标)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // 取得当前片段在光源视角下的深度
    float currentDepth = projCoords.z;
    float bias = max(0.005 * (1.0 - dot(normal, lightDir)), 0.0005);
    // 检查当前片段是否在阴影中
    float shadow = 0.0;
    vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
    for(int x = -1; x <= 1; ++x)
    {
        for(int y = -1; y <= 1; ++y)
        {
            float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
            float curDepthDelta = currentDepth - bias - pcfDepth;
            if(curDepthDelta > 0)
            {
                float param = (1 - curDepthDelta);
```

```
                shadow += param * param;
            }
        }
    }
    shadow /= 9.0;

    if(projCoords.z > 1.0)
        shadow = 0.0;

    return shadow;
}
```

镜面渲染



镜子上显示的图像，可以看做镜像过去的另一个人所看到的的情景。

将相机镜面对称过去，相机的 front 向量沿镜子法向量对称，更新相机，此时相机看到的东西就是镜面应该显示的内容。值得注意的是，此时看到的内容是左右颠倒的，在 shader 里面需要左右置换一下。

- 算出摄像机到镜子距离，沿着镜子法向量负方向延长两倍，得到镜像摄像机位置
- 将摄像机的 front 沿着镜子发现 reflect
- 更新摄像机
- 结束镜像应该恢复原本的摄像机

镜子绘制流程：

- 计算镜像摄像机，更新摄像机
- 允许裁剪
- 正常进行渲染流程，此时是镜像摄像机，因此产生了镜像
- 深度缓冲使镜像深度正确
- 模板缓冲使镜像位置正确
- 画镜像
- 恢复摄像机
- 画正常的所有物体

海面

FFT 海面来自 Jerry Tessendorf 的论文 Simulating Ocean Water (2001)。
海面可以看作复杂的波形叠加，但我们难以直接求出叠加波形，因此需要先求出频谱，再用傅里叶逆变换得到波形。

FFT 海面包括以下关键点：

- Gerstner 波：由正弦波模拟的水面较为平静，而正弦波受到"挤压"形成的 Gerstner 波，波峰窄，波谷宽，有浪尖和浪槽的感觉，更适合模拟海面。
- Phillips 频谱：根据海洋统计学公式，通过风向、重力加速度等物理量计算出的频谱。
- 快速傅里叶逆变换 IFFT：由频谱变换到波形，IFFT 比 IDFT 要高效得多。

生成 FFT 海面的基本流程如下：

基于海洋统计学公式，先通过风向、重力加速度等物理参数生成 Phillips 频谱，再用它来计算高度频谱和水平方向的偏移频谱。分别对上述频谱做 IFFT，得到 x、y、z 方向上的偏移图，从而计算出 Gerstner 波叠加的海面。

海面渲染：

本项目中，海面的渲染使用 Blinn-Phong 光照模型，太阳看作点光源，手动设置其位置。
远景雾化效果通过雾化函数 color = water_color * (1-fog_factor) + fog_color * fog_factor 来实现。其中，雾化因子 fog_factor 为关于深度 z 的二次函数，距离摄像机越远的海面的雾化程度越强。雾化颜色 fog_color 为定值，预先从天空盒的适当位置手动取得，而非在着色器中朝海面边缘方向对天空盒采样得到，以免天空盒颜色影响不同方向的海面颜色。

实例化（Instancing）

在绘制一个模型的大量实例时，如果采用在循环中调用 glDrawArrays 或 glDrawElements 的方式，很快就会因为绘制调用过多而达到性能瓶颈，因为每次调用上述函数都要进行 CPU 到 GPU 的通信，每次绘制之前都需要做很多准备工作，每次都需要重新发送数据。
如果采用实例化的方法，就可以将数据一次性发送给 GPU，用一个绘制函数让 OpenGL 利用这些数据绘制多个物体，从而节省每次绘制时 CPU 到 GPU 的通信，提高了渲染效率。
实例化方法对应的渲染函数为 glDrawArraysInstanced 和 glDrawElementsInstanced。

循环绘制

先生成一块海面的顶点数据，整个海面在 x 和 z 方向上分别有 m_nx 和 m_nz 块。
每次循环中将同一块海面进行平移，再用 glDrawElements 进行绘制。关键代码如下：

```
for (int j = 0; j < m_nz; j++) {
    for (int i = 0; i < m_nx; i++) {
        glm::mat4 model = glm::translate(model0, glm::vec3(length * i + m_offset.x, m_offset.y,
length * -j + m_offset.z));
        oceanShader.setMat4("model", model);
        glDrawElements(geometry ? GL_LINES : GL_TRIANGLES, indices_count, GL_UNSIGNED_INT, 0);
    }
}
```

实例化绘制

根据每块海面的偏移量计算出每块海面的 model 矩阵，存入 mat4 数组，把这个数组也作为顶点数据传入顶点着色器，各种顶点数据只需传输一次。

调用实例化绘制函数 glDrawElementsInstanced，顶点着色器在绘制海面的每个实例时将会使用实例化数组中不同的 model 矩阵。关键代码如下:

```
//实例化数组赋值
int index = 0;
for (int j = 0; j < m_nz; j++) {
    for (int i = 0; i < m_nx; i++) {
        glm::mat4 model = glm::translate(model0, glm::vec3(length * i + m_offset.x, m_offset.y,
length * -j + m_offset.z));
        modelMats[index++] = model;
    }
}
//绑定
unsigned int instanceVBO;
glGenBuffers(1, &instanceVBO);
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::mat4) * m_nz * m_nx, &modelMats[0], GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
//属性
glEnableVertexAttribArray(3);
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)0);
glEnableVertexAttribArray(4);
glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)(sizeof(glm::vec4)));
```

```
glEnableVertexAttribArray(5);
glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)(2 *
sizeof(glm::vec4)));
glEnableVertexAttribArray(6);
glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)(3 *
sizeof(glm::vec4)));
glVertexAttribDivisor(3, 1);
glVertexAttribDivisor(4, 1);
glVertexAttribDivisor(5, 1);
glVertexAttribDivisor(6, 1);
//绘制
glBindVertexArray(VAO);
glDrawElementsInstanced(geometry ? GL_LINES : GL_TRIANGLES, indices_count, GL_UNSIGNED_INT, 0, m_nz
* m_nx);
```

混合（Blending）

混合是 OpenGL 中用来实现物体透明度的一种技术。透明是指一个物体的颜色由物体本身的颜色和它背后其他物体的颜色以某种比例混合而成。
本项目中，海面的通透感并不是通过计算折射实现的，而是通过设置海面的透明度实现的。
混合函数的设置为 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)。
先渲染天空盒和其他物体，再渲染海面（alpha < 1.0），即可表现出半透明的海面效果。

PBR

PBR 主要使用的贴图意义解释如下：

反照率：反照率(Albedo)纹理为每一个金属的纹素(Texel)（纹理像素）指定表面颜色或者基础反射率。这和漫反射纹理相当类似。

法线：法线贴图使我们可以逐片段的指定独特的法线，来为表面制造出起伏不平的假象。

金属度：金属(Metallic)贴图逐个纹素的指定该纹素是不是金属质地的。

粗糙度：粗糙度(Roughness)贴图可以以纹素为单位指定某个表面有多粗糙。采样得来的粗糙度数值会影响一个表面的微平面统计学上的取向度。

AO：环境光遮蔽(Ambient Occlusion)贴图为表面和周围潜在的几何图形指定了一个额外的阴影因子。

PBR 实现集中在 shader 的编写中，PBR 的 shader 如下：

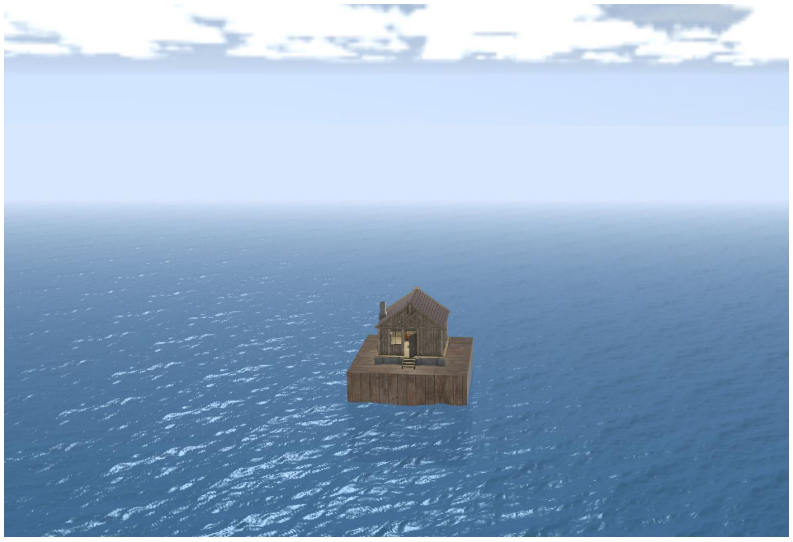
```
const float PI = 3.14159265359;
vec3 getNormalFromMap()
{
    vec3 tangentNormal = (texture(normalMap, TexCoords).xyz) * 2.0 - 1.0;

    vec3 Q1  = dFdx(WorldPos);
    vec3 Q2  = dFdy(WorldPos);
    vec2 st1 = dFdx(TexCoords);
    vec2 st2 = dFdy(TexCoords);

    vec3 N   = normalize(Normal);
    vec3 T  = normalize(Q1*st2.t - Q2*st1.t);
    vec3 B  = -normalize(cross(N, T));
    mat3 TBN = mat3(T, B, N);

    return normalize(TBN * tangentNormal);
}
// ----------------------------------------------------------------------------
float DistributionGGX(vec3 N, vec3 H, float roughness)
{
    float a = roughness*roughness;
    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;
}
// ----------------------------------------------------------------------------
float GeometrySchlickGGX(float NdotV, float roughness)
{
    float r = (roughness + 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 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;
}
// ----------------------------------------------------------------------------
vec3 fresnelSchlick(float cosTheta, vec3 F0)
{
    return F0 + (1.0 - F0) * pow(clamp(1.0 - cosTheta, 0.0, 1.0), 5.0);
}
vec3 fresnelSchlickRoughness(float cosTheta, vec3 F0, float roughness)
{
    return F0 + (max(vec3(1.0 - roughness), F0) - F0) * pow(clamp(1.0 - cosTheta, 0.0, 1.0), 5.0);
}
// ----------------------------------------------------------------------------
void main()
{
    vec3 albedo     = pow(texture(albedoMap, TexCoords).rgb, vec3(2.2));
    float metallic  = texture(armMap, TexCoords).b;
    float roughness = texture(armMap, TexCoords).g;
    float ao        = texture(armMap, TexCoords).r;


    vec3 N = getNormalFromMap();
    vec3 V = normalize(camPos - WorldPos);
    vec3 R = reflect(-V, N);

    vec3 F0 = vec3(0.04);
    F0 = mix(F0, albedo, metallic);
    vec3 Lo = vec3(0.0);
    for(int i = 0; i < 8; ++i)
    {
        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 numerator      = NDF * G * F;
        float denominator = 4 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0) + 0.0001;
        vec3 specular = numerator / denominator;

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

        float NdotL = max(dot(N, L), 0.0);

        Lo += (kD * albedo / PI + specular) * radiance * NdotL;
    }

    vec3 ambient = vec3(0.03) * albedo * ao;
    if(isIBL==1)
    {
        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;

        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);

        ambient = (kD * diffuse + specular) * ao;
    }
    vec3 color = ambient + Lo;

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

```
    FragColor = vec4(color, 1.0);
}
```

项目结果展示：



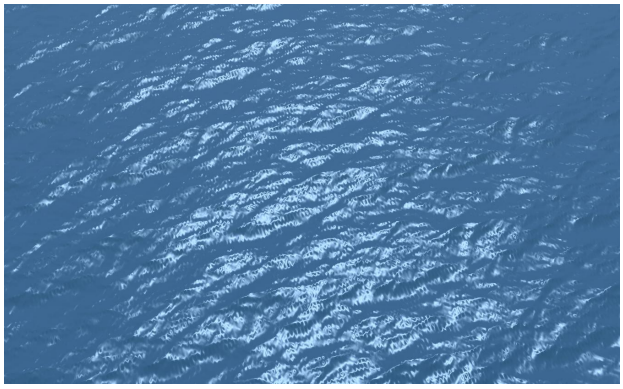
有无阴影的对比：

PBR 材质：

IBL 有无的对比：

镜面展示：

水体展示：



Roles in group

各成员主要工作如下：
海洋渲染：范正源
阴影渲染：钟伊凡
镜面渲染：武澳奇
PBR 和 IBL 技术：张馨月、汪冰海
模型寻找和导入：刘云帆
项目统筹和进度安排：汪冰海

References

[1] https://learnopengl-cn.github.io/
[2] https://github.com/QianMo/PBR-White-Paper
[3] https://www.jianshu.com/p/723a8560b1d6
[4] https://people.engr.tamu.edu/sueda/courses/CSC471/2016S/demos/khongton
[5] Tessendorf, Jerry. Simulating ocean water. Simulating nature: realistic and interactive techniques. SIGGRAPH 1.2 (2001): 5.
[6] https://www.keithlantz.net/2011/11/ocean-simulation-part-two-using-the-fast-fourier-transform/
[7] https://zhuanlan.zhihu.com/p/95482541
[8] https://zhuanlan.zhihu.com/p/31670275
[9] https://zhuanlan.zhihu.com/p/95917609
[10] https://polyhaven.com/models
[11] https://github.com/KhronosGroup/glTF