Project Report for the CG 100433 Course

Project Title

不那么自然的场景渲染

Team Members

- 2051565 龚天遥
- 2051834 陈君涛
- 2052479 孙颖斌
- 2053927 唐艺宁
- 2051506 敖佳琪
- 2050278 赵奕菲
- 2052911 顾启文

Abstract

本项目主要研究基于 OpenGL 的复杂场景渲染实现及其相关理论和技术。

我们主要完成以下几方面的工作：

- 对软件程序各方面需求及可行性的分析
- 建立 Visual Studio 2022 环境下的 OpenGL 程序框架
- OpenGL 下动态体积云、阴影、水面效果和后效的分析与具体实现

最后，基于上述理论，在 Windows 系统平台，使用 Visual Studio 2022 为构建系统，利用 OpenGL 的基本接口实现一个完整的场景渲染。

Motivation

近年来，图形图像制作技术发展迅速，尤其是计算机图形学。图形学和图像开始进入设计领域，促进设计领域的快速发展。电脑与美术的结合创造电脑美术艺术，在产品设计、动画、场景漫游等领域都有广泛的应用。

我们看到，大量游戏以及电影中的自然场景非常真实，它们实现与现实几乎别无二致的云雾、大气粒子以及水等物体或效果。更重要的是，它们在实时渲染中展示了物体与光线的复杂交互，实现逼真的阴影效果。受此震撼的同时，我们希望能够在学习计算机图形学、计算机科学、光学等多领域先进理论成果的基础上，模仿实现一个完整的场景渲染，使最终在视觉上产生动态效果和艺术效果，使观察者可以在视觉上得以享受。

The Goal of the Project

1. 实现动态体积云（或雾），其会影响地面物体的光照情况
2. 实现在自然场景中物体模型的展示，添加软阴影效果
3. 实现部分物体具有的特殊材质，如类似镜面反射的材质效果
4. 实现动态的水面并令其拥有较真实的反光效果
5. 实现后处理效果，包括颜色取反、灰度变化、周围像素混合等

The Scope of the Project

1. 实现物体的软阴影，但不考虑自身几何关系很复杂的物体以及内部的阴影
2. 暂不考虑光线追踪，不考虑云或雾内部对太阳光线的多次发射折射
3. 不考虑观察者与世界内环境与物体的实时交互，无物理系统
4. 限制观察者（相机）移动，如不能飞行进入天空得到云内部和潜入水中
5. PBR 模型按需引入

Involved CG Techniques

实时体积云

海面（水体渲染）

- Sine wave superposition 可以把一个水面当成很多个正弦波的叠加，每个正弦波的方向、移动速度和振幅都不一样，再累加到一起，就近似真实的水波了。
- gestner wave 最简单的海面模拟无非就是Sin函数或者Cos函数,但是更真实的海浪具有更加陡峭的波峰,所以使用Gerstner Wave Function。
- nomal mapping 原物体的凹凸表面的每个点上均作法线，通过RGB颜色通道来标记法线的方向

天空盒与纹理

- 纹理映射，可能需要法线或位移贴图（球面或更复杂表面的环境贴图）
- 立方体贴图

后效

- 利用帧缓存实现简单的风格化滤镜，泛光等效果（这部分可以通过各类贴图或者是类似图像处理的方法作较简单实现）

阴影（Shadow Mapping）

- shadow mapping：使用**阴影映射**实现了场景中物体在动态光源下的硬阴影
- PCF（percentage-closer filtering）:使用**百分比接近滤波技术**，从阴影深度图对应位置周围采样进行滤波来对阴影边沿进行抗锯齿，实现边沿较柔和的软阴影。
- PCSS (Percentage-Closer Soft Shadows)：**百分比接近软阴影**，在PCF基础上，根据遮挡物与光源和着色点的距离，利用相似三角形的原理，计算出PCF应该采样的范围大小，使得阴影的“软”“硬”更接近实际光照的结果。
- 泊松圆盘采样：代替传统PCF的方形滤波采样，根据一定数学规律得到中心周围一系列类似圆盘分布的采样点，使得阴影边沿结果过渡更自然。

景深

- 基于帧缓冲实现

Project Contents

- 实时动态体积云：模拟天空中的云的效果。支持更改云的形状、颜色、移动速度等。
- 软阴影：基于 Shadow Map 计算得到的软阴影，随光源和模型位置变化而实时变化。
- 景深：模拟相机对焦距离的效果。
- 场景管理：多个场景完全独立，可以互相传送。传送时上一场景的状态保存在栈内。
- 后效：基于帧缓冲，实现反色、模糊等效果。

- 动态水面：基于数学方法计算得到的动态水面。会根据相机位置反射高光。
- 天空盒：为场景添加好看的天空。

Implementation

实时体积云

基础：Ray Marching

体积云的渲染基于光线步进算法。基本思路为从相机出发发出射线，每次前进一小段距离，计算这一点的体块密度并累加。累加得到的结果即为这一方向上的云累积的量。对每个frag进行光线步进，即可得到视野中每个像素的云层密度。这个算法的消耗比较高，可以从使射线从云层底部开始，步进到云层顶部结束。增大步长可以提高效率，但渲染质量也相对降低了。

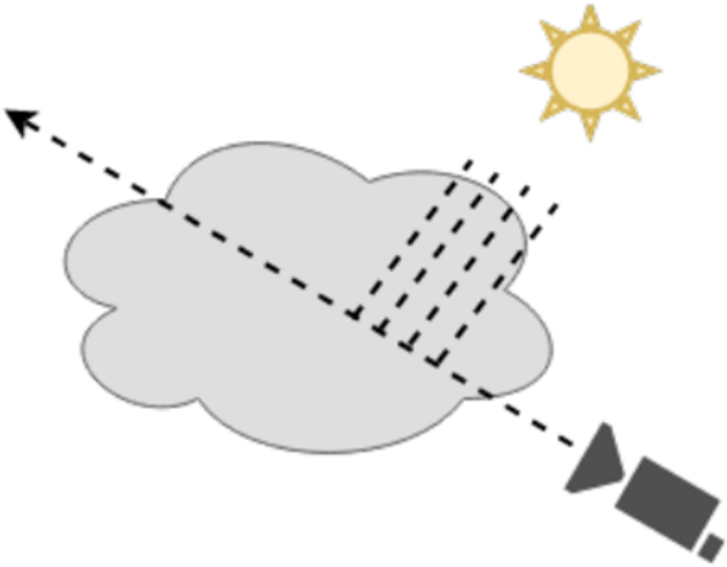
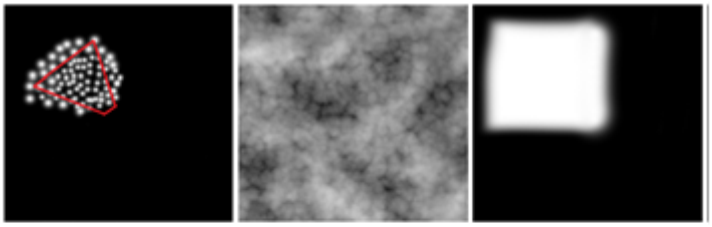


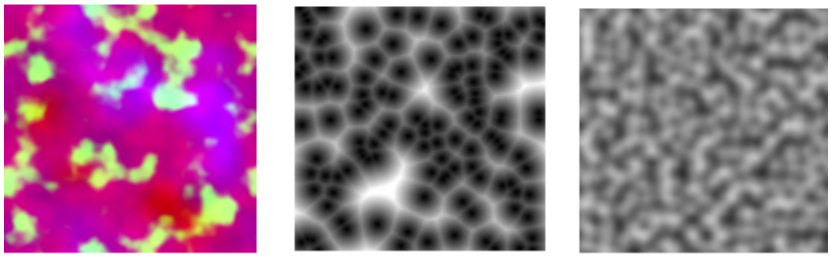
Figure 14: Ray-marching clouds. Visualizing one viewing-ray.

云层密度采样

体积云渲染还需要知道某个点的云密度。密度采样大体可分为形状、细节两个层次，将多个采样叠加起来得到当前点的云层密度。通过天气图(weather map)确定云层的形状、总体密度和相对位置。天气图是一张具有3(或4)通道的材质，r通道表示低覆盖云层分布，g通道表示高覆盖云层分布，b通道表示云峰值高度。再混合柏林噪声、细胞噪声等丰富云层细节。



(a) Low-coverage map (red channel). (b) High-coverage map (green channel). (c) Cloud peak height (blue channel).



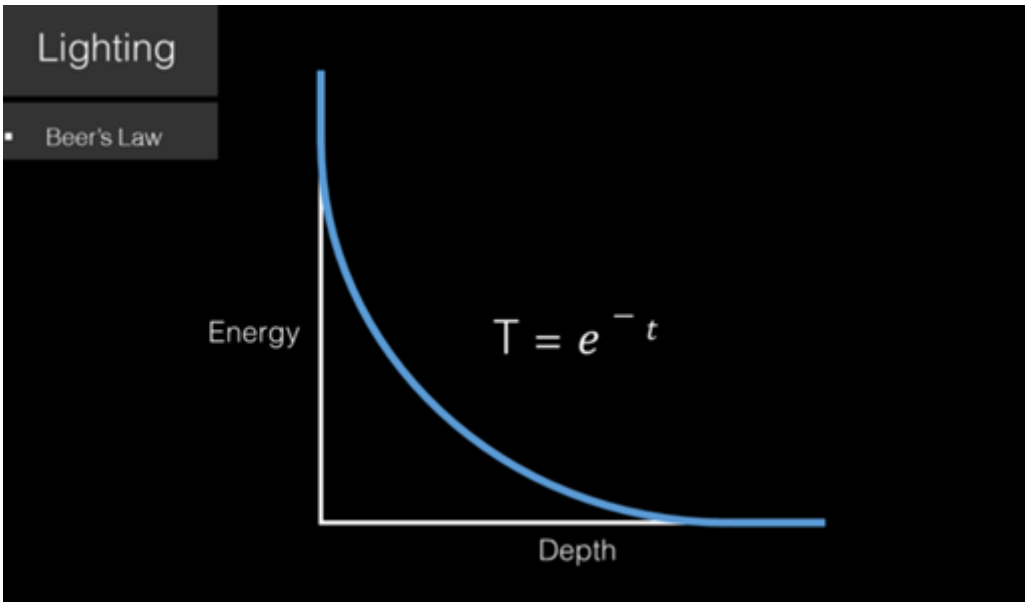
按照参考文献[1]中给出的公式进行混合，得到云层中某一点的云密度。

光照

接下来要解决的就是云层的光照。

在真实生活中，当光线进入云层中会被粒子或水滴吸收一部分，然后在被散射到其他方向，接着又被其他粒子或水滴吸收和散射。光照可以分为外散射、内散射、吸收三个部分。

首先，想要知道一点的亮度，我们就得知道这一点到光源之间隔了多厚的云层。估算计算点在光线方向上的密度，应用Beer's law通过光学密度来计算出透射率，作为吸收和外散射，就可以获得一个基本的光照。

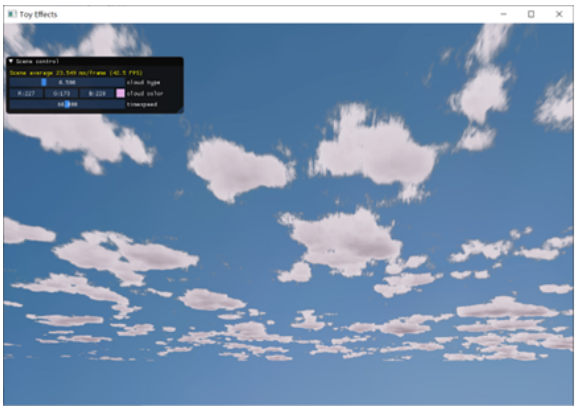


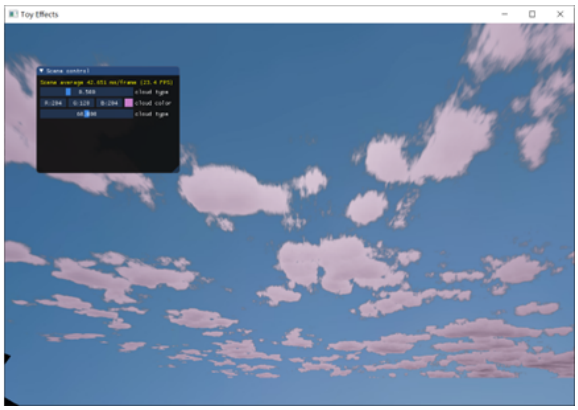
到这一步就可以获得一个看起来比较合理的光照了。不过在著名的《THE REAL-TIME VOLUMETRIC CLOUDSCAPES OF HORIZON ZERO DAWN》中提出的光照模型还考虑了更多的细节。

第一，在逆光时，光线在云中散射的方向大部分都是朝向人眼的，而且在云比较薄的地方会比较快的穿出云层，减少了被吸收、外散射的概率。HG相位函数可以量化这种现象。

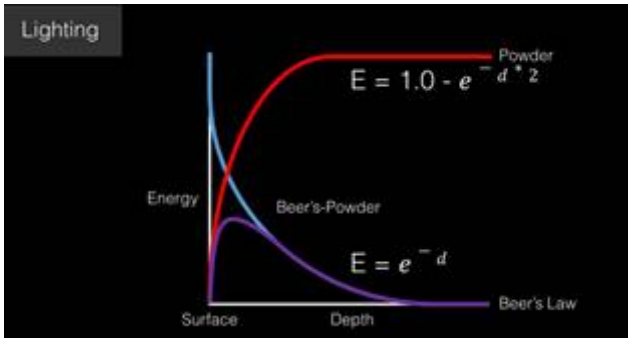
$$HG(\theta, g) = \frac{1}{4\pi} \frac{1 - g^2}{[1 + g^2 - 2g \cdot cos(\theta)]^{3/2}}$$

左右是未使用HG和使用HG的对比图。HG使得逆光方向上的光线更明显，加强了视觉表现。





第二，当我们顺着太阳光观看厚云层时，可以明显的看到云上的暗边，以及折痕地方更亮的效果。这种效果也被称为糖粉效应。Beer's Powder函数刻画了这一效果，用它来替换Beer's function。



这两步额外的光照计算在一定程度上提高了渲染的真实感，也不可避免地带来渲染成本的提升。

优化

体积云渲染的基础——光线步进本来就是一个昂贵的操作，而在实时渲染中我们不可能仅仅为云层的渲染分配过多的GPU资源。因此在对帧率要求较高的情况下，还需要加入一些优化策略。

- 降低像素。创建一个比屏幕更小的帧缓存，渲染一个更小的体积云图像，最后映射到屏幕。减少需要处理的像素能显著减少功耗。不过创建帧缓存也会带来额外的开销。
- 从云层底部开始光线步进，光线穿出云层时结束，也就是剔除了云层密度为0的部分，这种优化是不影响效果的。
- 增大光线步进的步长，减少采样次数，这对效率的提升非常明显，不过会带来渲染质量的损失。

经过优化，单独渲染云的场景可以跑到200帧以上，不过最终采用了质量更高的版本。

海面

我们需要运行两个表面模拟:一个是表面网格的几何波动，另一个是网格上法线图的扰动。水面高度可以由不同的方法实现，简单的有波的叠加，gestner波等等，复杂的有fft等方法。

在对比了不同制造高度场的效果后，还是采用了原理较为简单但是实现效果更加贴合场景的波的叠加来实现。

首先，正弦波的公式如下：

$$y(x,t) = A \cdot sin2\pi(\frac{t}{T} - \frac{x}{\lambda}), x > 0$$

其中，A 是振幅，T 是周期，λ 是波长

2D理论上，单层方向与水平成夹角的法线公式如下：

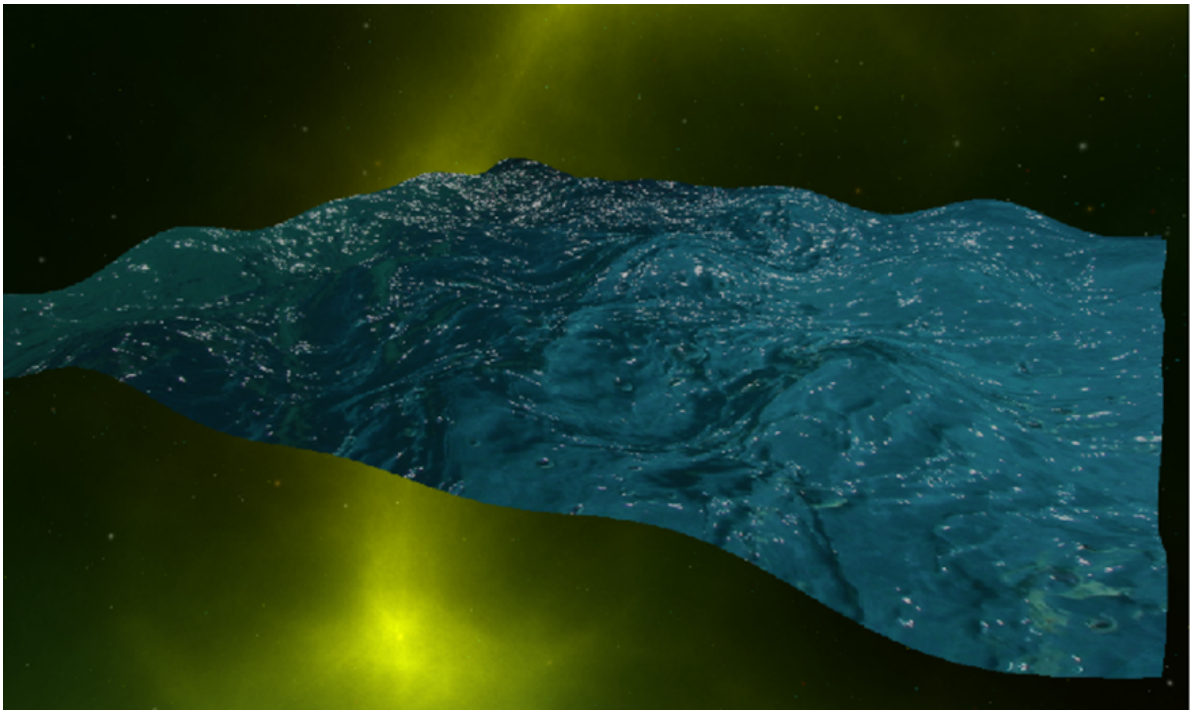
$$\mathbf{N}(x,y)=\left(-\frac{\partial}{\partial x}\left(H(x,y,t)\right),-\frac{\partial}{\partial y}\left(H(x,y,t)\right),1\right).$$

3D场景下，单层方向与水平成夹角theta的公式如下：

$$W_i(x,y,t)=A_i\,sin\left(\begin{bmatrix}cos\theta_i\\sin\theta_i\end{bmatrix}\cdot\begin{bmatrix}x\\y\end{bmatrix}\omega_i+t\varphi_i\right)$$

正弦函数叠加得到了一个函数，描述了水面上所有点的高度和方向。在处理顶点时，基于每个顶点的水平位置对函数取样，使得网格细分形成连续水面。W函数记录了单个点在某一时刻所对应的高度场，H函数则作为整个面上的高度场集合，为法线贴图的计算做准备。

单层sine波效果：



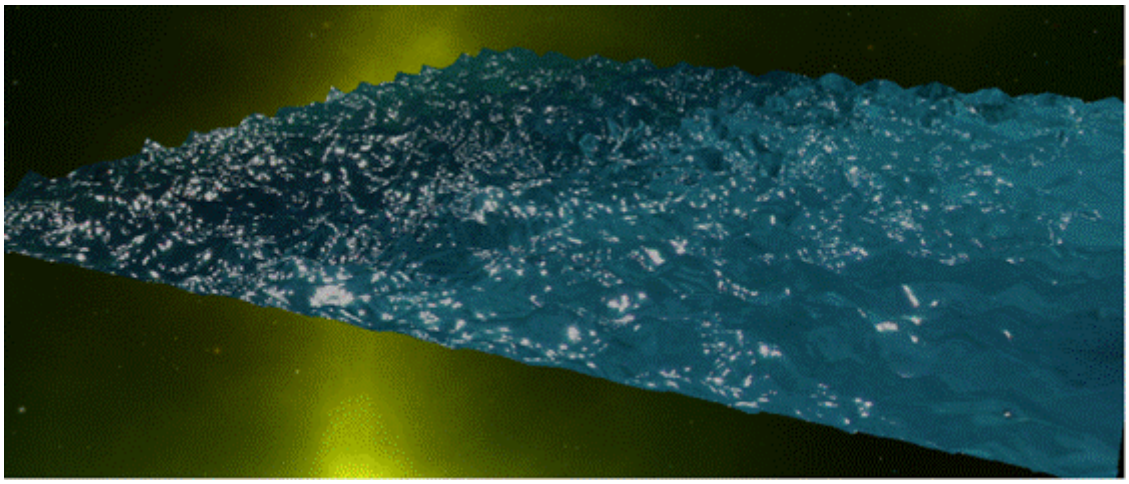
可以看到单层Sin波的效果比较圆润，适用于比较平静的湖面。gestner波则相比之下波峰更尖，由于前面得到的效果过于圆润这里我们转而尝试gestner波。

当波浪在水面上移动时，水本身不会随之移动。在正弦波的情况下，每个表面点上下移动，但不水平移动。Gestner的每个表面点都围绕一个固定锚点绕一个圆移动。当波峰接近时，该点会向它移动。波峰通过后，它向后滑动，然后下一个波峰出现，以实现水在波峰中聚集并在水槽中扩散的效果。

Gestner波shader实现：

```
...
k = (2.0 * pi) / wave_length;
float c = sqrt(9.8 / k);
f = k * (dot(d, pos.xz) - c * uvscroll);
float h1 = amplitude * sin(f);
...
return h1 + h2;
```

单层Gestner波效果：

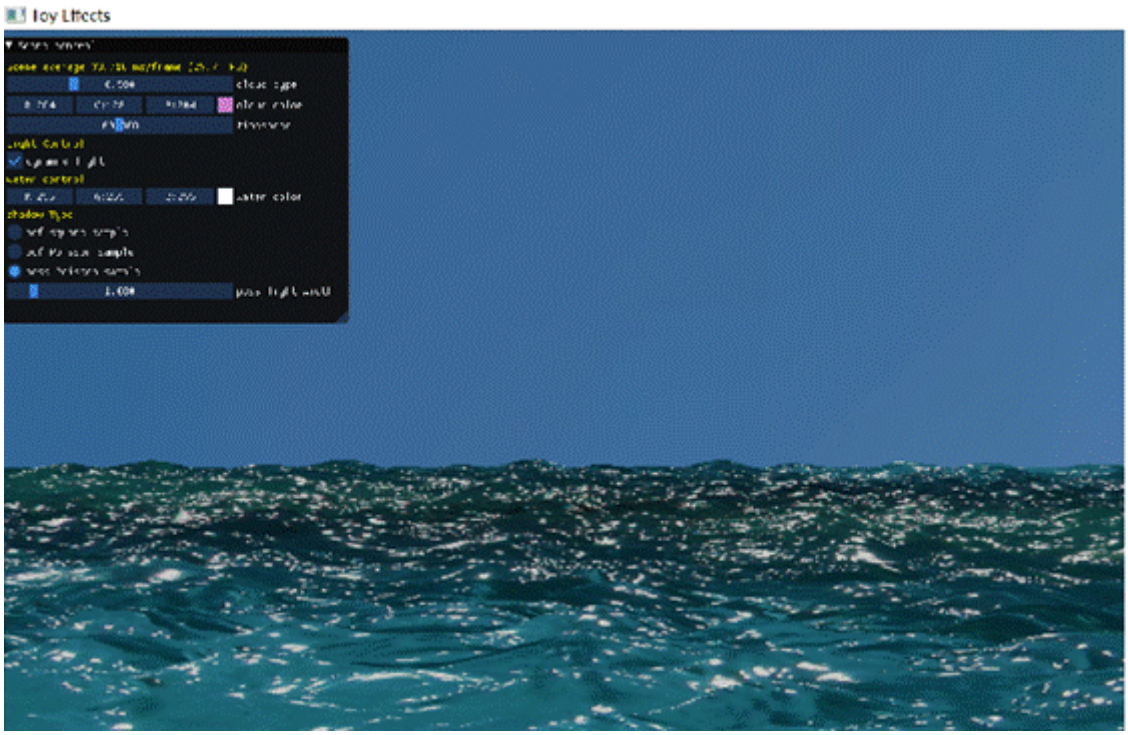


效果比较生硬，不如多层叠加的sine波，因此最终还是采用多层叠加的sine波

多层多方向sine波叠加shader如下：

```
...
float speeded = uvScroll * 1.5;
float component1 = sin(2.0 * PI * speeded - 2 * 0.707 * pos.x - 0.707 * pos.y) *
amplitude;
float component2 = sin(2.0 * PI * speeded - 2 * pos.x) * amplitude;
float component3 = sin(2.0 * PI * speeded - 2 * pos.y) * amplitude;
...
return component1 + component2+component3;
```

最终效果如下，并可以调节海面颜色：



阴影

阴影映射的基本方法

阴影映射需要对场景进行两遍渲染，可以随着光源的变化而得到动态的阴影效果。

第一次：从光源处看场景，渲染一张场景深度图

第二次：光线从相机出发投射向场景，如果投射位置和光源的距离，比在深度图中该方向的深度值更大，说明这个位置无法被光源照射到，处于阴影中，就给其按阴影着色。

由于本组渲染的场景中有噪声实现的动态体积云、各类后效以及GUI显示，不同渲染阶段需要配置不同设置的缓冲以及各类测试，因此渲染循环中的各环境模块的渲染顺序需要调整正确，否则就会导致阴影效果丢失或者影响其他效果的正常展示：

```
void WaterCloudShadowScene::render()
{

    getDirLightShadowMap();//【深度测试】第一遍得到深度图
    // 启用后效帧缓冲。
    glBindFramebuffer(GL_FRAMEBUFFER, fbo);
    glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    pSkybox->render();//【移出】这个需要深度测试，且需要在画云之前，否则会把阴影场景弄没
    renderCloud();   //【关闭深度测试、使用混合】画云【最后关闭混合，打开深度测试】
    renderSceneWithShadows();//第二遍画阴影（云是没有阴影的）
    renderWater();    //画水

    glm::vec3 p=camera->getPosition();
    //cout << "相机" << p.x << ' ' << p.y << ' ' << p.z << endl;
    // 将后效帧缓冲绘制到屏幕缓冲。
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glDisable(GL_DEPTH_TEST);
    screenShader.use();
    screenShader.setInt("effect", this->postProcessEffect);
    glBindVertexArray(scrQuadVao);
    glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
    glDrawArrays(GL_TRIANGLES, 0, 6);
    glEnable(GL_DEPTH_TEST);

    vcgui->render();//【移出】控制云的gui最后渲染，无需判断也不会被水和场景挡住

}
```

PCF方法与采样的改进

使用PCF对阴影映射得到的硬阴影边沿进行软化，在计算某个着色点顶点阴影情况时，不是简单根据其与深度图中对应位置比较从而得到“在阴影中”或“不在阴影中”这样二选一的离散结果。而采样深度图周边NxN区域内多个点的深度值，用其是否在阴影区内（0,1）的平均值作为该点阴影程度的衡量，实现能够表现不同明暗程度的、有一定过渡变化的阴影。

```
// PCF
float shadow =0.0;
vec2 texelSize=1.0/textureSize(shadowMap,0);
for(int x=-2 ;x≤2;x++)
{
    for(int y=-2 ;y≤2;y++)
    {
        float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) *
texelSize).r;
        //shadow += (currentDepth-bias > pcfDepth && projCoords.z≤1.0) ? 1.0 :
0.0;
        shadow += (currentDepth-bias > pcfDepth ) ? 1.0 : 0.0;
    }
}
shadow /= 25.0;//float(NUM_SAMPLES); 该点阴影程度是 5x5 区域的平均值。
```

泊松圆盘采样改进

采样的方法会对阴影边沿造成严重影响，使用NxN的方形滤波已经能够较大弱化阴影边沿远处可见的明显锯齿。但是稍近一些还是能看到边沿存在严重走样，就像几个正方形稍错位重叠形成的网格。我们查阅有关资料发现这是方形滤波近似的一大缺陷，一种被广泛接受的改进采样方法是使用泊松盘采样，它从圆盘中心一圈圈往外采样，每圈采样一个点，并对采样点附加一个角度的旋转，（思想类似下图，但不是所有点都使用，从中心旋转向外满足一定角度和半径就取一个点）。从几何角度可以看出，这样的采样点会在靠近中心的单位面积上分布更密集，相对于方形均匀滤波更贴切实际。

采样点的生成

```
vec2 poissonDisk[NUM_SAMPLES];

void poissonDiskSamples(const in vec2 randomSeed) {
    float ANGLE_STEP = PI2 * float(NUM_RINGS) / float(NUM_SAMPLES);
    float INV_NUM_SAMPLES = 1.0 / float(NUM_SAMPLES);

    float angle = rand_2to1(randomSeed) * PI2;
    float radius = INV_NUM_SAMPLES;
    float radiusStep = radius;

    for (int i = 0; i < NUM_SAMPLES; i++) {
        poissonDisk[i] = vec2(cos(angle), sin(angle)) * pow(radius, 0.75);
        radius += radiusStep;
        angle += ANGLE_STEP;
    }
}
```

泊松盘上的采样点提前计算就可以得到，但将其应用到深度纹理上采样时还应注意这里圆盘上的偏移量与方形滤波不一样，要乘上一个合适的步长才能映射到shadowmap上对应的uv坐标进行计算。

```
// 泊松圆盘采样
poissonDiskSamples(projCoords.xy);
// shadow map大小
float texSize=textureSize(shadowMap,0).x;
// 滤波的步长
float filterStride = 5.0;
// 滤波窗口的范围
float filterRange = 1.0 / texSize * filterStride;
// 当前深度
float currentDepth = projCoords.z;

//add bias to solve shadow acne(与采样方法也有关)
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);

float shadow =0.0;
for( int i = 0; i < NUM_SAMPLES; i ++ ) {
    vec2 sampleCoord = poissonDisk[i] * filterRange + projCoords.xy;

    float pcfDepth = texture(shadowMap, sampleCoord).r;

    shadow += (currentDepth-bias > pcfDepth ) ? 1.0 : 0.0;
}

shadow/=float(NUM_SAMPLES);

return shadow;
```

PCSS方法

先计算有多少点在阴影里，用上一步的泊松盘采样中取的采样点计算多个点深度图深度总和与阴影中的点的总数的比值作为窗口范围内能够产生阴影的平均深度。

```
for(int i=0;i<NUM_SAMPLES;i++)
{
    vec2 sampleCoord = poissonDisk[i] * filterRange + projCoords.xy;
    float shadowDepth=texture(shadowMap,sampleCoord).r;
    if(currentDepth-bias >shadowDepth)
    {
        blockCnt++;
        totalDepth+=shadowDepth;
    }
}

if(blockCnt==0 )
    return 1.0;
else
    return totalDepth/float(blockCnt);
```

此时着色点与平均遮挡物的距离则可使用z近似计算，根据简单几何知识得到半影区penumWidth的大小。

```
float blockDepth=getBlockDepth(projCoords,normal,lightDir);
float recDepth=projCoords.z;

//半影区大小
float penumWidth=lightWidth*(recDepth-blockDepth)/blockDepth;
```

在采样点的偏移值基础上简单乘上半影长度就实现了按半影比例“拉远”了每个采样点与中心的距离，宏观上就是整个影子得到一定程度的拉长（但是filterange相对于PCF需要调小一些避免了采样率不足导致的问题）

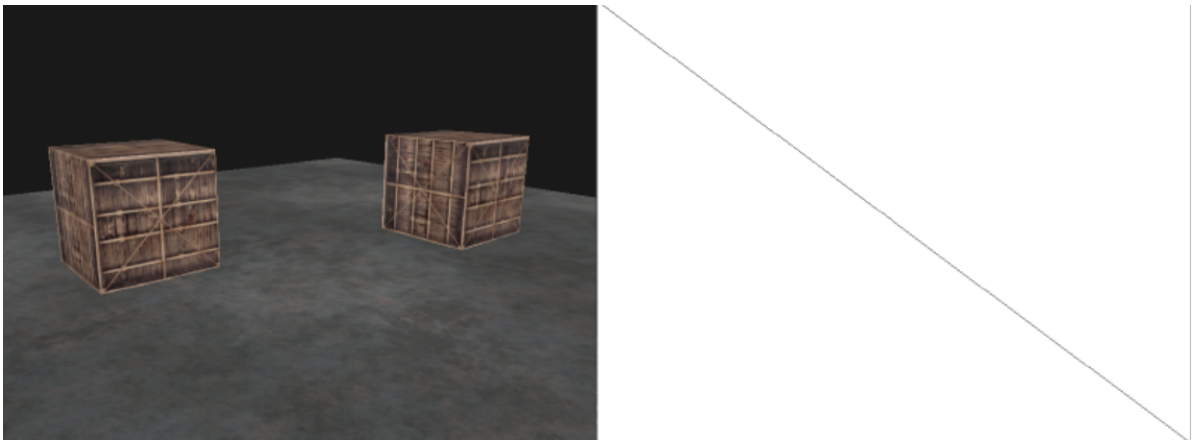
```
float shadow =0.0;
for( int i = 0; i < NUM_SAMPLES; i ++ ) {
    //注意这里多出来乘半影范围
    vec2 sampleCoord = penumWidth* poissonDisk[i] * filterRange + projCoords.xy;
    float pcfDepth = texture(shadowMap, sampleCoord).r;

    shadow += (currentDepth-bias > pcfDepth ) ? 1.0 : 0.0;
}
shadow/=float(NUM_SAMPLES);
```

后效

在初步学习图形学相关 API 时，我们经常直接将画面中的内容直接绘制到屏幕帧缓冲里。

事实上，我们可以将屏幕帧缓冲视为一张图片。也就是说，我们可以自己创建一张图片（贴图，Texture），将场景中的物体绘制到该图片上，而不是直接送入屏幕帧缓冲。之后，我们将这张图片作为整块屏幕的贴图，传递到帧缓冲。



此过程中，我们可以利用 shader 进行一些有意思的操作，如：后效。

总体流程如下：

1. 创建一个纹理对象
2. 令场景中的其他物件渲染到上一步的纹理对象上
3. 用两个三角形组成矩形，恰好覆盖整个屏幕
4. 将上述纹理对象作为屏幕的贴图，在 fragment shader 阶段添加效果。借助贴图的性质，可以在像素级别做处理，甚至可以采集周围像素的数据进行操作

反色

将每个像素做反色即可。

```
FragColor = vec4(vec3(1.0 - texture(screenTexture, TexCoords)), 1.0);
```

灰度

根据像素的三原色信息，混合得到一个亮度值。将该值设为新的三原色值，即可实现黑白效果。

```
FragColor = texture(screenTexture, TexCoords);
float average = 0.2126 * FragColor.r + 0.7152 * FragColor.g + 0.0722 *
FragColor.b;
FragColor = vec4(average, average, average, 1.0);
```

基于 kernel 的效果

部分效果的实现需要依赖周围像素。我们将周围正方形内总共9个像素取出，各自乘以一定的值，混合得到新的颜色。将每个像素乘以的值拼在一起，得到一个矩阵。该矩阵称为 kernel。

借助不同的 kernel，可以实现不同的效果。

```
float kernel[9] = ...

vec3 sampleTex[9];
for(int i = 0; i < 9; i++)
{
    sampleTex[i] = vec3(texture(screenTexture, TexCoords.st +
    offsets[i]));
}
vec3 col = vec3(0.0);

for(int i = 0; i < 9; i++)
    col += sampleTex[i] * kernel[i];

FragColor = vec4(col, 1.0);
```

边缘检测

对应 kernel 如下：

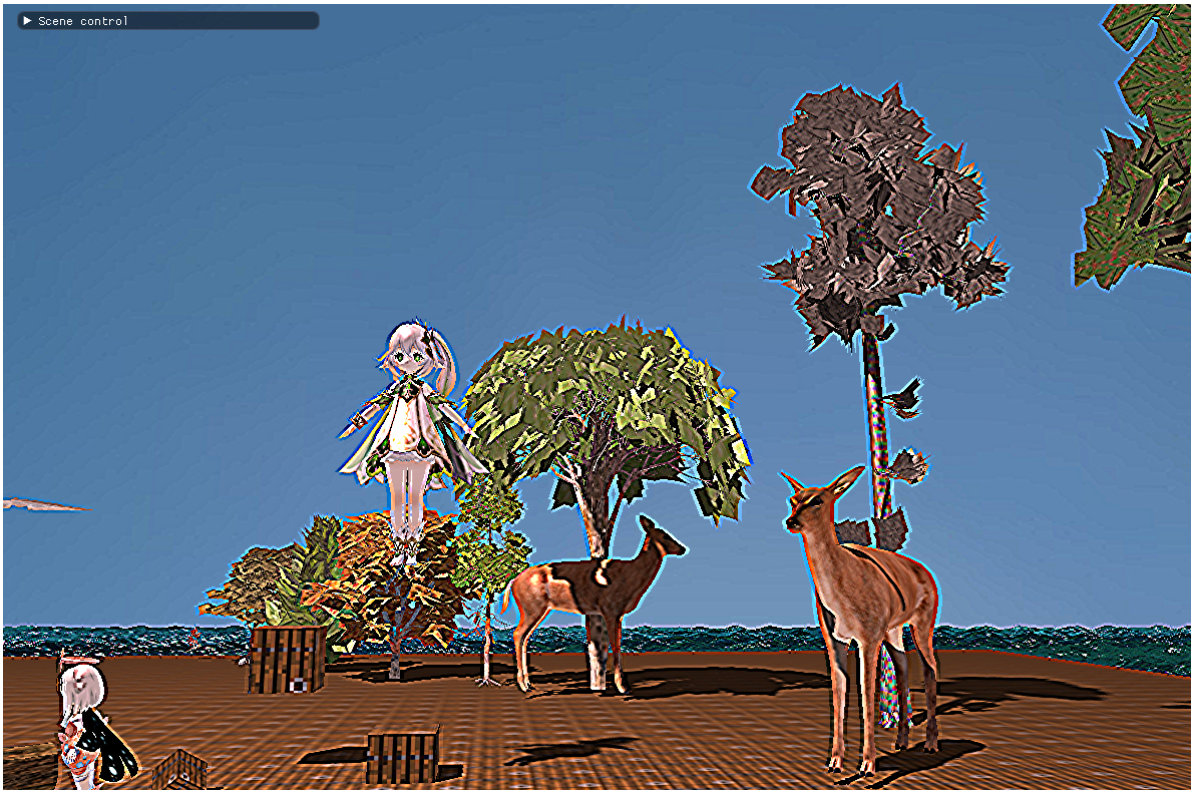
$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

盒子模糊

对应 kernel 如下：

$$\begin{pmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix}$$

锐化

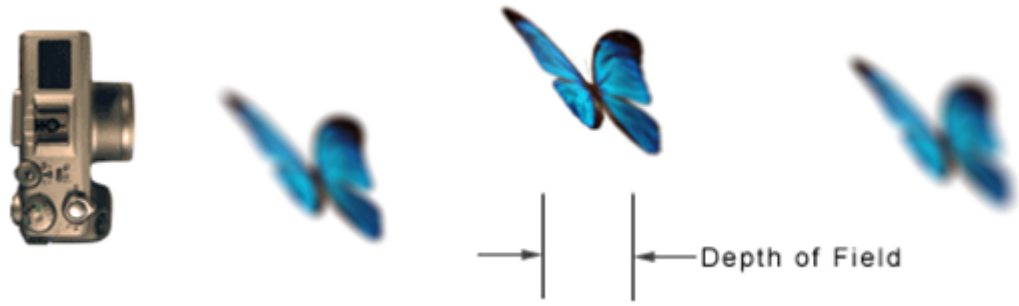


对应 kernel 如下：

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

景深

在这里我们实现的是非光线追踪的景深处理。为了在绘制中模拟景深这一效果，一个直观的想法是，获取物体和相机的距离，根据这一距离和远近平面的关系，来决定物体的清晰以及模糊状态，以及它的模糊程度。



在这里为了在绘制中模拟景深这一效果，我们需要使用帧缓冲技术。简单来说，先把场景渲染到帧缓冲中，然后再把这个帧缓冲渲染到一个和屏幕同样大小的面片上，在第二次渲染的过程中，同时进行后期处理。在模糊方面我们采用的方法是用一个滤波算子来将临近像素按照一定的比例进行混合，从而达到模糊的效果。但是简单的模糊处理没有得到相机的深度信息，也就无法根据深度来确定模糊因子。在这

里我们是采用将场景渲染到纹理的同时，获取深度，并计算对应的模糊因子，然后将这一数据写入纹理的alpha通道。在第二次渲染处理的过程中我们就可以很方便的来对其获取相关信息进行处理了。

帧缓冲技术

帧缓冲的工作流程大致如下：

1. 创建帧缓冲对象：首先需要创建一个帧缓冲对象，用于存储渲染的图像。
2. 创建缓冲区：然后需要创建一些缓冲区，用于存储颜色、深度和模板信息。
3. 绑定帧缓冲：在渲染之前，需要将帧缓冲绑定到当前的上下文中。
4. 渲染场景：接下来就可以开始渲染场景了。在渲染过程中，图形的几何数据会被转换为像素，并存储到帧缓冲的缓冲区中。
5. 多重渲染：如果需要进行多重渲染，可以在渲染完一帧图像后，再对其进行多次渲染，以添加额外的效果。
6. 显示图像：最后，将渲染完成的图像显示到屏幕上。

由此可见，我们一共需要创建两个对象，帧缓冲对象和纹理，并且需要建立两者的对应的关系。

```
glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);

glGenTextures(1, &textureColorbuffer);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, app.getWindowWidth(), app.getWindowHeight(), 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glBindTexture(GL_TEXTURE_2D, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, textureColorbuffer, 0);
```

注意在这里为了写入深度数据，纹理的通道为RGBA，其余的地方属于帧缓冲对象创建的常规方法。

```
unsigned int rbo;
glGenRenderbuffers(1, &rbo);
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, app.getWindowWidth(), app.getWindowHeight());
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << endl;
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

这一部分添加的是深度附件到帧缓冲中，为了便于OpenGL进行深度测试。由于我们只希望采样颜色缓冲，而不是其它的缓冲，我们可以为它们创建一个渲染缓冲对象。

着色器部分

在这里我们需要两次渲染，在两次渲染中分别获取不同的数据以便我们进行后续的处理。

第一次渲染

顶点着色器

```
gl_Position = view * model * vec4(aPos, 1.0);

TexCoords = aTexCoords;
v_depth = gl_Position.z;

gl_Position = projection * gl_Position;
```

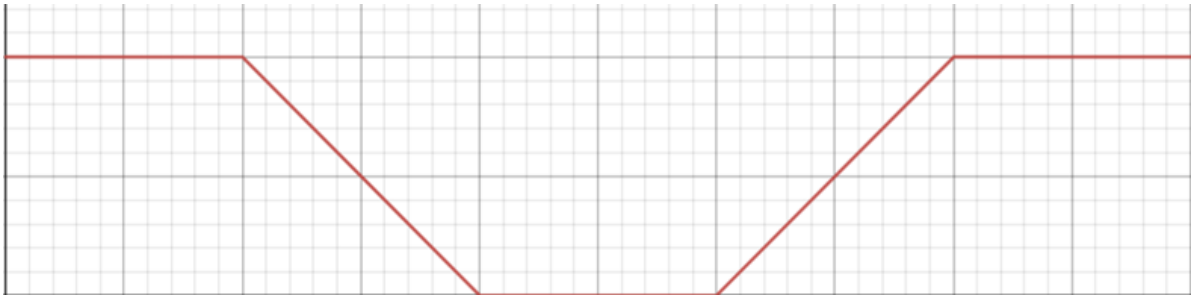
在这里我们主要是要获取相机的深度信息。注意在这里我们首先要转换到视点空间之后再记录相机深度信息，之后再转换到投影空间。

片段着色器

```
FragColor = texture2D(textureDiffuse1, TexCoords);
float blur = 0;
// 根据深度计算模糊因子
if(v_depth <= near_plane && v_depth >= far_plane)
{
    blur = 0;
}
else if(v_depth > near_plane)
{
    blur = clamp(v_depth, near_plane, near_plane + near_distance);
    blur = (blur - near_plane) / near_distance;
}
else if(v_depth < far_plane)
{
    blur = clamp(v_depth, far_plane - far_distance, far_plane);
    blur = (far_plane - blur) / far_distance;
}
// 将模糊因子写入alpha通道
FragColor.a = blur;
```

在这里我们远、近平面以及衰减范围是定义的为uniform，在后续能够通过GUI来实时调整其远近以及衰减范围大小从而改变景深距离。

这里根据深度计算模糊因子实质上是在计算一个分段函数，衰减范围即对应图中的模糊区，在这一区域中，模糊因子往靠近远/近平面的方向从1递减到0，在景深范围内为0，也就代表这一区域是完全清晰的。在这里对模糊因子的部分采用的是线性计算，结合实际情况还可以采用其他的曲线。



第二次渲染

顶点着色器

```
void main() {
    TexCoords = aTexCoords;
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
}
```

在这里只是简单进行数据的传输，没有其他的操作

片段着色器

```
vec4 v4Original = texture2D(textureDiffuse1, TexCoords);

vec2 v4ScreenSize = screenSize / 5;
vec3 v3Blurred = vec3(0, 0, 0);
for(int i = 0; i < kernelNum; i++)
{
    vec2 v2Offset = vec2(g_v2TwelveKernelBase[i].x / v4ScreenSize.x,
    g_v2TwelveKernelBase[i].y / v4ScreenSize.y);
    vec4 v4Current = texture2D(textureDiffuse1, TexCoords + v2Offset);
    v3Blurred += mix(v4Original.rgb, v4Current.rgb, v4Original.a);
}
FragColor = vec4 (v3Blurred / kernelNum , 1.0f);
```

在这里我们采用的算子记录的是纹理的偏移值，利用这一偏移数据，获取当前像素的周围12个像素，并计算这12个像素的平均值，最终得到一张模糊的图像。

之后，根据模糊因子(取值范围在0~1) 之间，在原图像和模糊图像之间进行插值，得到最终的数据。

Results

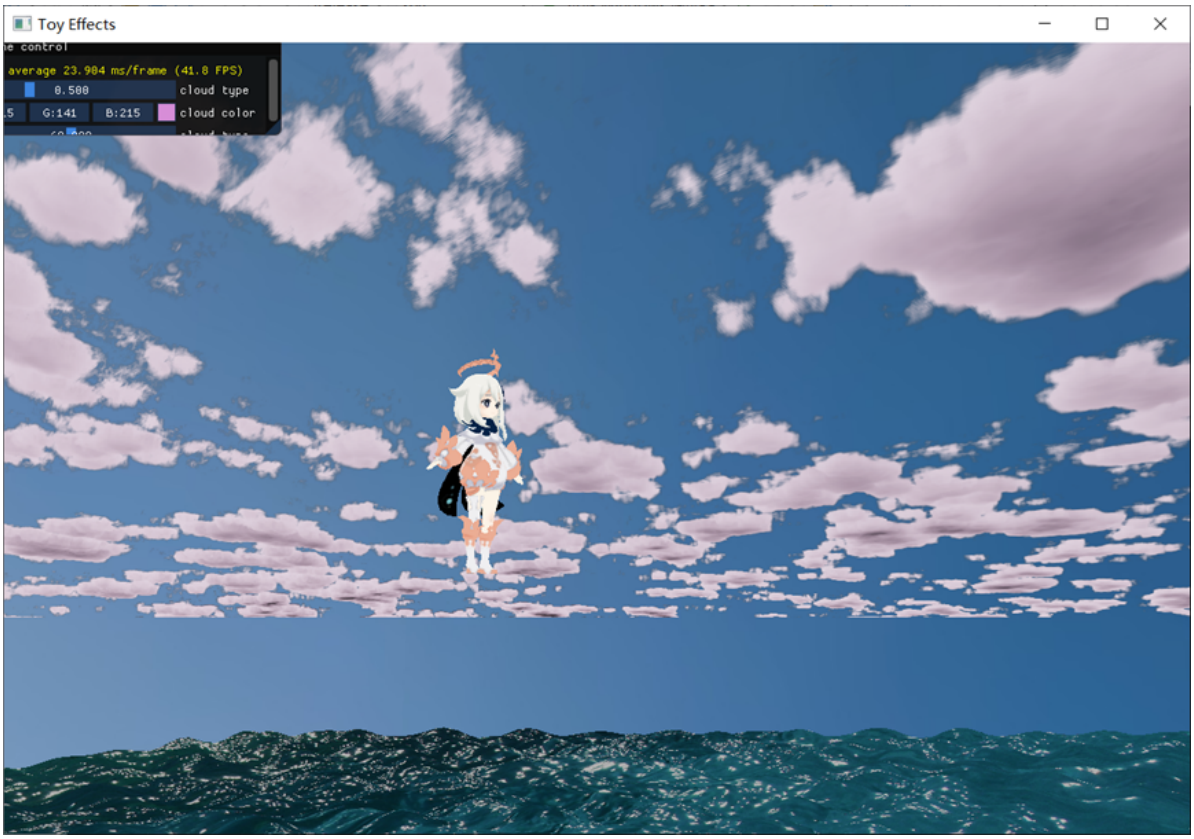
演示视频见提交的附件。

也可以在线观看：https://www.bilibili.com/video/BV1r84y1W7fs

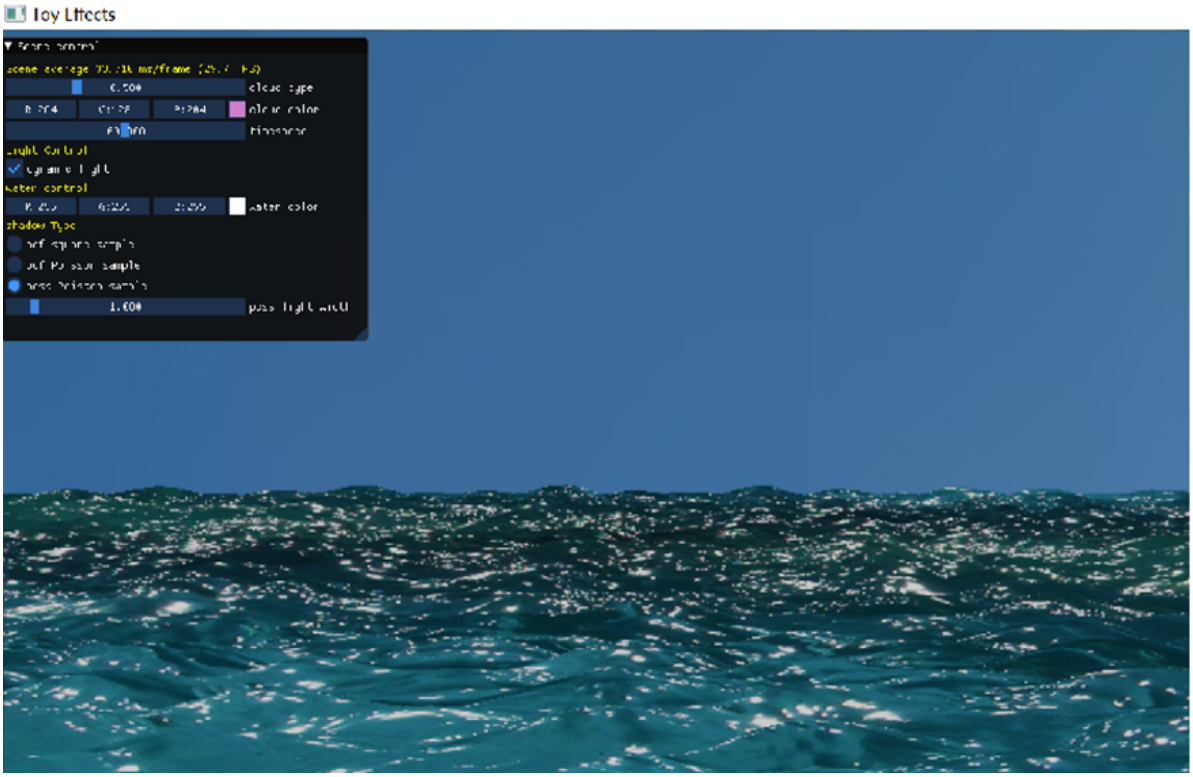
项目仓库：https://github.com/FlowerBlackG/ToyEffects

总体场景

实时体积云



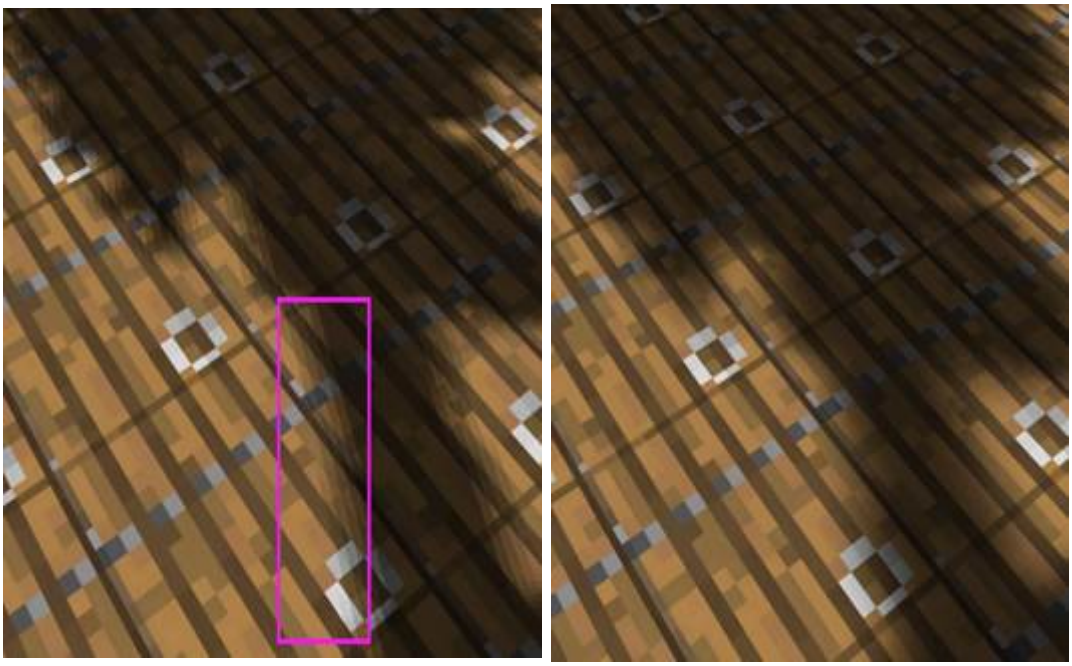
海面



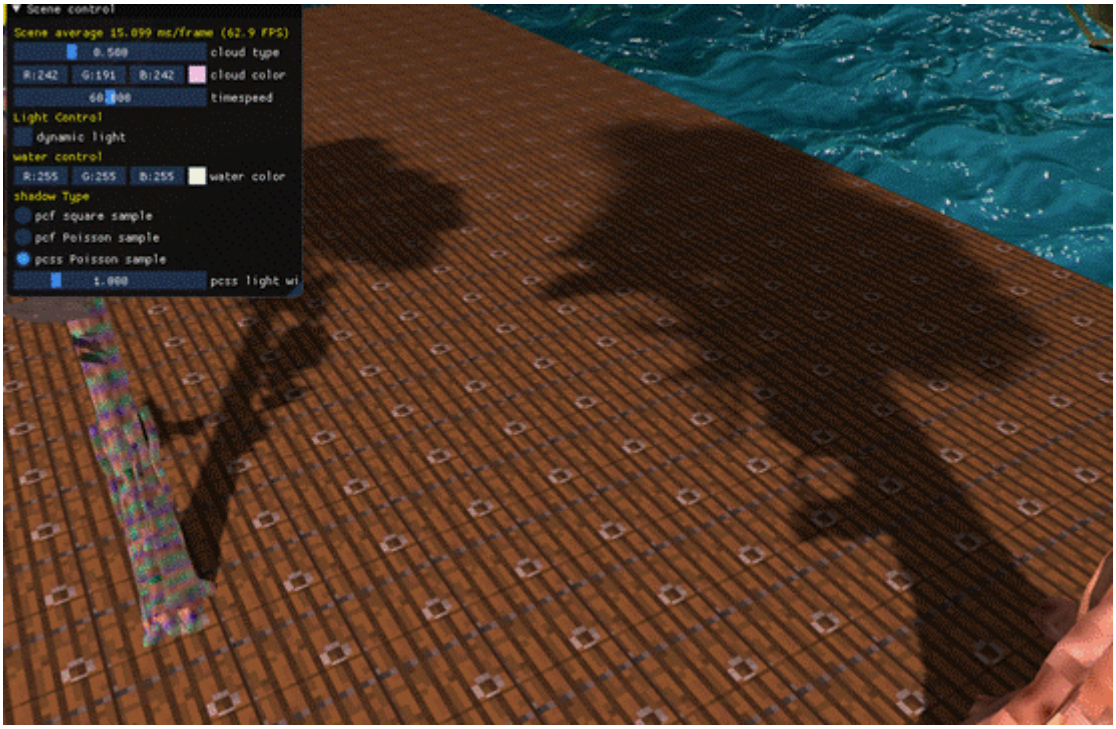
阴影

可以明显看到采样数分别为49、50、50的情况下，PCSS的效果在离光源较远的树顶处影子明显模糊，而根部的阴影较硬些，而两种采样方法的PCF阴影边沿无论远近都进行了统一的模糊与软化。

采用NxN方格区域采样滤波的PCF将锯齿转化为平行或垂直的采样纹，减轻了走样，但是由于采样方式固有的不足，始终无法解决这种近距离下的“网格线”状锯齿，引入泊松盘采样后，边沿的模糊变得非常自然，只有在运动以及复杂模型的情况会看到散立的采样噪声。



完整场景中的阴影效果如下，可以开启光源的动态，阴影也会动态变化。


Scene control
Scene average 15.899 ms/frame (62.9 FPS)
0.500
cloud type
R:242
G:191
B:242
cloud color
timespeed
Light Control
dynamic light
water control
R:255
G:255
B:255
water color
shadow Type
pcf square sample
pcf Poisson sample
pcss Poisson sample
1.000
pcss light wi

后效

景深


Scene control
Scene average 15.724 ms/frame (63.6 FPS)
10.000
near_distance
10.000
far_distance
-12.000
near_plane
-20.000
far_plane

Roles in Group

2051565 龚天遥：场景管理器，天空盒，后效

2051834 陈君涛：软阴影，分支整合，模型调整

2052479 孙颖斌：景深，多光源（尝试）

2053927 唐艺宁：实时动态体积云，光照

2051506 敖佳琪：动态水面，分支整合

2050278 赵奕菲：镜面（尝试），模型调整

2052911 顾启文：动态水面，分支整合

我们认为，小组内每位同学享有相同的贡献度。

References

- [1] Fredrik Haggstrom. Real-time rendering of volumetric clouds. 2018
- [2] GPU Gems Chapter 1. Effective Water Simulation from Physical Models
- [3] https://catlikecoding.com/unity/tutorials/flow/waves Catlike Coding
- [4] 实时渲染 | Shadow Map：PCF、PCSS、VSM、MSM-知乎 (zhihu.com)
- [5] https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping
- [6] GAMES202-高质量实时渲染. https://www.bilibili.com/video/BV1YK4y1T7yY