CG Project-谜岛与飞机

3180106078 张文韬

3180103009 钟添芸

3180101964 胡泳欢

1 游戏简介

1.1 目录约定

报告文件：.\report.pdf

可执行文件：.\executable\CG-Project.exe

OBJ 资源及输出目录：.\resources\

纹理资源目录：.\texture\

C++代码目录：.\code\CG-Project\

GLSL-Shader 代码目录：.\shader\

VS 工程文件：.\code\props\demo.props 与.\code\CG-Project.sln

代码依赖项目录：.\code\include\与,\code\lib\

注 1：OBJ 与纹理资源来自网络免费资源（有部分修改），版权归提供方所有

注 2：如需要 Build 代码，请另外参照 2.1 节的说明

1.2 整体流程

在本游戏中，玩家将扮演一个流落荒岛的角色。主角出生在一个岛上，眼前只有一架损坏的飞机与一片草地，岛的旁边还有一个被半透明魔法罩所覆盖的深井。

主角虽然并不清楚发生了什么，但姑且先跟着提示去收集零件吧。游戏开始前这些小岛有半透明魔法墙阻隔，并不能进入，但会对谜题有所提示。主角需要按下开始键让半透明魔法墙消失并进入游戏。

这个小岛是一片沙漠。主角的步伐会在柔软的细沙上留下痕迹。沙漠中有几个奇怪的石柱，游戏开始前他们按一个古怪的顺序周期性地闪动着。主角很快反应过来——主角需要记住这些石柱的闪动顺序，然后按照这个顺序靠近石柱。

主角回到中心岛修好了飞机。天空中忽然出现了一些光圈，而主角现在也拥有了触碰这些光圈的飞行能力。跟随着光圈的指引，主角在天空中环绕了一周，最终被吸入了先前不可触碰的深井。

进入深井后，主角的飞机失去了控制，奇异的是，穿过深井之后飞机急速坠落的目标点居然是和谜岛如出一辙的岛屿。先前修好的三个零件也飞了出去，主角也逐渐闭上了眼睛。

游戏通关后，主角再次睁开眼睛，发现眼前仍然是熟悉的小岛和坏掉的飞机……

1.3 操作说明

地面行走模式操作：

- W, A, S, D 键控制前后左右移动
- 鼠标移动控制屏幕转向
- 鼠标左键持续按下可以拖动鼠标而不移动屏幕
- 玩家不可以穿越半透明墙与岸边（由空气墙保证）

小岛谜题操作：

- H 键令玩家复位并循环播放石柱闪动顺序
- R 键开始游戏，此后玩家需要操控主角按记住的顺序靠近触碰石柱。全部触碰完成之后，若顺序正确，石柱将显示为绿色，损坏的飞机将被修复且显示为红色；若顺序错误，石柱将显示为红色，玩家需要再次按 H 键记住顺序并再次按 R 键开始谜题
- Q 键重置谜题，但也会清除胜利的记录
- 在完成谜题之后，玩家需要返回主岛并靠近修复完成的飞机

飞机操作

- W, S 键控制飞机油门
- A，D 键控制飞机左右方向转向

- 鼠标移动控制飞机的正面移动方向
- 鼠标左键持续按下可以拖动鼠标检视周围画面
- 该模式下摄像机会自动逐渐追踪飞机位置
- 玩家需要操控飞机触碰天空中的所有红色光圈，触碰成功将转换为绿色

辅助操作

- F1 键截图，保存在./screenshot 中
- F2 键导出飞机的当前姿态，保存在./resource/out.obj 中

全局测试作弊操作

- Y 键强制在飞机模式与地面行走模型中进行切换
- P 键暂停飞机物理引擎，进入漫游模式。漫游模式下摄像机无碰撞体积，鼠标单击后拖动屏幕移动摄像机方向，鼠标滚轮前后滚动移近/移远摄像机位置
- F3 键开启光照变化，光照的方向与颜色将发生随机渐变
- F5 键强制播放飞机起飞 CG
- F6 键强制播放飞机坠毁（游戏结束）CG

2 代码实现

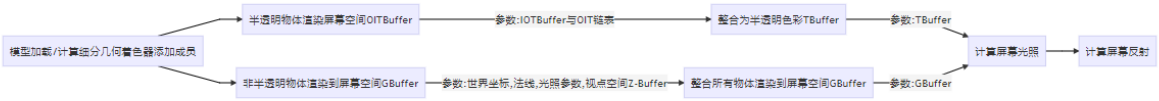
2.1 基本约定

- 系统环境：Windows 10 - 64bit
- 依赖库：GLFW、GLAD、GLM、KHR、stb_image
- 工程 IDE：Visual Studio Enterprise 2019
- Build 参数：C++17，x64，O2 优化

注 1：如果需要 Build 代码，请将根目录下所有的其他文件夹复制到./code/中，使用.\code\CG-Project.sln 进行 Build

注 2：本 Project 共 68 个 cpp、h、shader 文件，代码行数约 6800 行，因此本节仅展示关键代码，具体完整实现请参照代码文件

2.2 渲染流程架构



上图为整体的渲染流程，整体上把半透明物体与不透明物体分开渲染。透明物体使用顺序无关透明（OIT）技术，对于每个屏幕空间的像素点存储一个深度与颜色信息的列表存储到 TBuffer 中，再得到混合后的半透明颜色；对于不透明物体，使用延迟渲染技术，对于每个屏幕空间的像素点存储法线、深度、颜色等信息在 GBuffer 中，整合之后与光照进行计算。最后将 GBuffer 的颜色结果与 ZBuffer 的颜色结果进行混合，再额外计算反射。

代码实现上，为了将渲染自动化，我们做了抽象：

- 所有物体的渲染都经过全部的渲染 Pass，但部分 Pass 可以留空
- 渲染操作利用虚函数重载完成，不留参数表，参数访问使用全局信息池（并非直接的全局变量）
- 代码中可以具名地创建实体（每个实体有唯一 Handle），但必须不具名地加入渲染池（ViewGroup）

ViewObject 基类：

```cpp
class ViewObject : public ViewBase
{
public:
    virtual void RenderShadowBuffer() {}
    virtual void RenderGBuffer() {}
    virtual void RenderOITBuffer() {}
    virtual void RenderBlendOIT() {}
    virtual void RenderGBufferIlluminate() {}

public:
    // rotate this object around certain axis(x, y, z) and angle degree
    void Rotate(const GLfloat& angle, const glm::vec3& axis = glm::vec3(0.0f, 0.0f, 1.0f));
    // translate this object by displacement vector
    void Translate(const glm::vec3& displacement);
    // scale this object with (x, y, z)
    void Scale(const glm::vec3& scaler);
    // get model matrix of this object
    glm::mat4 GetM() const;
    // set model matrix of this object
    void SetM(const glm::mat4& M);
    // render this object in given pass
    virtual void Render(const ViewPassEnum& pass);
    // get object type
    ViewObjectEnum GetType() const;
    // show this object
    void Show();
    // hide this object
    void Hide();
```

ViewGroup 模板类：

```cpp
template <typename T>
class ViewGroup : public ViewBase
{
public:
    void AddObject(const std::shared_ptr<T>& object) { ... }

    std::shared_ptr<T> GetObject(const HandleT& objectID) { ... }

    void RemoveObject(const HandleT& objectID) { ... }

    std::vector<std::shared_ptr<T>> GetObjectList() { ... }

protected:
    std::vector<std::shared_ptr<T>> _objects;
};
```

Scene 中暴露的 ViewGroup：

```
namespace Scene
{
    extern ViewGroup<TexturedPlane> planeGroup;
    extern ViewGroup<TransparentPlane> transparentGroup;
    extern ViewGroup<FitTexturedPlane> fitPlaneGroup;
    extern ViewGroup<WaterPlane> waterPlaneGroup;
    extern ViewGroup<SkyBox> skyBoxGroup;
```

接下来以具纹理不透明多边形(TexturedPlane)为例讲解渲染流程：

```
class TexturedPlane : public ViewObject
{
public:
    virtual void RenderGrassGBuffer();
    virtual void RenderGBuffer();
    virtual void RenderShadowGrassBuffer();
    virtual void RenderShadowBuffer();

public:
    void Init(glm::vec3 position, GLfloat points[], GLint sizeofPoints, GLuint indices[], GLint sizeofIn
    TexturedPlane(glm::vec3 position, GLfloat points[], GLint sizeofPoints, GLuint indices[], GLint size
    TexturedPlane(std::vector<GLfloat> vertices) { ... }
    TexturedPlane(std::vector<GLfloat> vertices, std::vector<GLuint> indices, std::vector<TextureInfo> p
    void ChangeTexture(const std::vector<TextureInfo> &infos);
    void prepare(Light& light);

    enum { PlaneVAO, NumPlaneVAO };
    enum { PlaneArrayBuffer, PlaneElementBuffer, NumPlaneBuffer };
    enum { position = 0, texCoord = 1 };
    enum { PlaneColorTexture = AmbientTexture, PlaneNormalTexture = NormalTexture, PlaneRoughnessTexture
```

在继承的虚函数中给出具体实现，完成具体的渲染操作。

而实体创建则需要加入到渲染 Group(即 ViewGroup<TexturedPlane>)中：

```
planeGroup.AddObject(std::shared_ptr<TexturedPlane>(new TexturedPlane(glm::vec3(0, 0, 0), planePoints1, sizeof(planePoints1),
    planeIndices1, sizeof(planeIndices1), planeShaders, planeTextures)));
planeGroup.AddObject(std::shared_ptr<TexturedPlane>(new TexturedPlane(glm::vec3(0, 0, 0), planePoints2, sizeof(planePoints2),
    planeIndices2, sizeof(planeIndices2), planeShaders, redTextures)));
planeGroup.AddObject(std::shared_ptr<TexturedPlane>(new TexturedPlane(glm::vec3(0, 0, 0), planePoints3, sizeof(planePoints3),
    planeIndices3, sizeof(planeIndices3), planeShaders, yellowTextures)));
planeGroup.AddObject(std::shared_ptr<TexturedPlane>(new TexturedPlane(glm::vec3(0, 0, 0), planePoints3, sizeof(planePoints3),
    planeIndices3, sizeof(planeIndices3), planeShaders, greenTextures)));
planeGroup.AddObject(std::shared_ptr<TexturedPlane>(new TexturedPlane(glm::vec3(0, 0, 0), planePoints3, sizeof(planePoints3),
    planeIndices3, sizeof(planeIndices3), planeShaders, grayTextures)));
```

其他的抽象物体与实体也是类似的构建与操作。而在 main 的主循环中，我们进行如下渲染：

```
while (!glfwWindowShouldClose(window)) {
    UpdateData();

    RenderShadow();
    RenderGBuffer();
    if (UseOITFlag)
    {
        RenderOITBufferTM();
        RenderBlendOIT();
    }
    RenderGBufferLight();

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

而上述的 Render 函数则有类似如下实现：

```
void RenderOITBufferTM() {
    glBindFramebuffer(GL_FRAMEBUFFER, 0);

    //glClearBufferfv(GL_COLOR, 0, bgColor);
    glDisable(GL_DEPTH_TEST);
    //glDisable(GL_CULL_FACE);
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, OITBuffers[HeadPointerInit]);
    glBindTexture(GL_TEXTURE_2D, OITTextures[HeadPointerTexture]);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_R32UI, width, height, 0, GL_RED_INTEGER, GL_UNSIGNED_INT, NULL);

    // Bind head-pointer image for read-write
    glBindImageTexture(0, OITTextures[HeadPointerTexture], 0, GL_FALSE, 0, GL_READ_WRITE, GL_R32UI);
    // Bind linked-list buffer for write
    glBindImageTexture(1, OITTextures[StorageTexture], 0, GL_FALSE, 0, GL_READ_WRITE, GL_RGBA32UI);
    glBindImageTexture(2, OITTextures[StorageWorldPosTexture], 0, GL_FALSE, 0, GL_READ_WRITE, GL_RGBA32F);
    glBindImageTexture(3, OITTextures[StorageNormalTexture], 0, GL_FALSE, 0, GL_READ_WRITE, GL_RGBA32F);
    glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, OITBuffers[AtomicCounter]);
    const GLuint zero = 0;
    glBufferSubData(GL_ATOMIC_COUNTER_BUFFER, 0, sizeof(zero), &zero);

    RenderAllObject(ViewPassEnum::OITBuffer);

    glEnable(GL_DEPTH_TEST);

}
```

其中的 RenderAllObject 完成了对所有 ViewGroup 的渲染：

```cpp
void RenderAllObject(const ViewPassEnum& pass)
{
    for (const auto& object : skyBoxGroup.GetObjectList())
    {
        object->Render(pass);
    }
    for (const auto& object : planeGroup.GetObjectList())
    {
        object->Render(pass);
    }
    for (const auto& object : transparentGroup.GetObjectList())
    {
        object->Render(pass);
    }
    for (const auto& object : fitPlaneGroup.GetObjectList())
    {
        object->Render(pass);
    }
    for (const auto& object : waterPlaneGroup.GetObjectList())
    {
        object->Render(pass);
    }

}
```

2.3 阴影投射

首先初始化一个 Shadow FrameBuffer：

```cpp
void InitShadow()
{
    glGenFramebuffers(NumShadowFrameBuffer, shadowFrameBuffers);

    glGenBuffers(NumShadowTextureBuffer, shadowBuffers);
    glBindTexture(GL_TEXTURE_2D, shadowBuffers[ShadowBuffer]);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

    glBindFramebuffer(GL_FRAMEBUFFER, shadowFrameBuffers[ShadowFrameBuffer]);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, shadowBuffers[ShadowBuffer], 0);
    glDrawBuffer(GL_NONE);
    glReadBuffer(GL_NONE);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

之后的阴影信息都写在这个 framebuffer 中。以 TexturedPlane 为例，渲染阴影时首先绑定：

```
void RenderShadow() {
    using namespace Shadow;
    glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
    glBindFramebuffer(GL_FRAMEBUFFER, shadowFrameBuffers[ShadowFrameBuffer]);
    glClear(GL_DEPTH_BUFFER_BIT);

    RenderAllObject(ViewPassEnum::ShadowBuffer);

    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glViewport(0, 0, width, height);
}
```

然后传入需要的光空间 MVP 矩阵：

```
void TexturedPlane::RenderShadowBuffer()
{
    glm::mat4 lightProjection = GlobalDataPool::GetData<glm::mat4>("lightProjection");
    glm::mat4 lightView = GlobalDataPool::GetData<glm::mat4>("lightView");
    if (renderGrassFlag)
    {
        this->RenderShadowGrassBuffer();
    }
    GLint location;
    GLuint program = Shadow::shadowPrograms[Shadow::PlaneProgram];
    glUseProgram(program);

    location = glGetUniformLocation(program, "uniM");
    glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(this->modelMat));
    location = glGetUniformLocation(program, "uniV");
    glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(lightView));
    location = glGetUniformLocation(program, "uniP");
    glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(lightProjection));

    GLfloat uniNear = GlobalDataPool::GetData<GLfloat>("uniNear");
    location = glGetUniformLocation(program, "uniNear");
    glUniform1f(location, uniNear);
    GLfloat uniFar = GlobalDataPool::GetData<GLfloat>("uniFar");
    location = glGetUniformLocation(program, "uniFar");
    glUniform1f(location, uniFar);

    glBindVertexArray(this->VAOs[PlaneVAO]);
    glDrawElements(GL_TRIANGLES, this->VertexNum, GL_UNSIGNED_INT, 0);
}
```

2.4 延迟渲染

创建一个 FrameBuffer 以及相应的 Color Attachment：

```cpp
void Scene::InitGBuffer() {
    glGenFramebuffers(NumBuffer, GBuffers);
    glBindFramebuffer(GL_FRAMEBUFFER, GBuffers[GBuffer]);

    glGenTextures(NumGTexture, GTextures);
    // - 位置颜色缓冲
    glBindTexture(GL_TEXTURE_2D, GTextures[GPositionTexture]);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB32F, width, height, 0, GL_RGB, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, GTextures[GPositionTexture], 0);

    // - 法线颜色缓冲
    glBindTexture(GL_TEXTURE_2D, GTextures[GNormalTexture]);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, width, height, 0, GL_RGB, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, GTextures[GNormalTexture], 0);

    // - 颜色 + 镜面颜色缓冲
    glBindTexture(GL_TEXTURE_2D, GTextures[GAlbedoSpecTexture]);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

每次渲染之前绑定 G FrameBuffer：

```cpp
void RenderGBuffer() {
    glBindFramebuffer(GL_FRAMEBUFFER, GBuffers[GBuffer]);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    RenderAllObject(ViewPassEnum::GBuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

传入 uniform，Bind Texture，调用 Program：

```
    glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(this->modelMat));
    location = glGetUniformLocation(GBufferProgram, "uniV");
    glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(uniV));
    location = glGetUniformLocation(GBufferProgram, "uniP");
    glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(uniP));
    location = glGetUniformLocation(GBufferProgram, "TBN");
    glUniformMatrix3fv(location, 1, GL_FALSE, glm::value_ptr(this->TBN));
    location = glGetUniformLocation(GBufferProgram, "colorTexture");
    glUniform1i(location, PlaneColorTexture);
    location = glGetUniformLocation(GBufferProgram, "normalTexture");
    glUniform1i(location, PlaneNormalTexture);
    location = glGetUniformLocation(GBufferProgram, "roughnessTexture");
    glUniform1i(location, PlaneRoughnessTexture);
    GLfloat uniNear = GlobalDataPool::GetData<GLfloat>("uniNear");
    location = glGetUniformLocation(GBufferProgram, "uniNear");
    glUniform1f(location, uniNear);
    GLfloat uniFar = GlobalDataPool::GetData<GLfloat>("uniFar");
    location = glGetUniformLocation(GBufferProgram, "uniFar");
    glUniform1f(location, uniFar);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, this->Textures[PlaneColorTexture]);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, this->Textures[PlaneNormalTexture]);
    glActiveTexture(GL_TEXTURE2);
    glBindTexture(GL_TEXTURE_2D, this->Textures[PlaneRoughnessTexture]);
```

```
glBindVertexArray(this->VAOs[PlaneVAO]);
glDrawElements(GL_TRIANGLES, this->VertexNum, GL_UNSIGNED_INT, 0);
glUseProgram(0);
```

最终渲染光照的 Shader：

```
void main() {
    // 从G缓冲中获取数据
    float specularStrength = 0.5;
    float roughness = texture(gAlbedoSpec, textureCoord).a;
    vec3 fragPos = texture(gPosition, textureCoord).rgb;
    vec3 originalColor = texture(gAlbedoSpec, textureCoord).rgb;
    vec3 normal = texture(gNormal, textureCoord).rgb;

    //part1
    vec3 ambient = light.ambient * lightColor;
    //part2
    vec3 lightDir = normalize(light.position - fragPos);
    float diff = max(dot(lightDir, normal), 0.0f);
    vec3 diffuse = max(dot(lightDir, normal), 0.0f) * lightColor;
    //part3
    vec3 viewDir = normalize(viewPos - fragPos);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 64);
    //spec = 0;
    vec3 specular = light.specular * spec * lightColor * vec3(1 - roughness);
    //part4
    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance * distance));
    //float isLight = dot(lightDir, normal) * dot(viewDir, normal) < 0 ? 0 : 1;
    //part5 shadow
    //float bias = max(0.005 * (1.0 - dot(normal, lightDir)), 0.001);
    float bias = 0.0001;
    vec4 FragPosLightSpace = lightSpaceMatrix * vec4(fragPos, 1.0);
    float shadow = ShadowCalculation(bias, FragPosLightSpace);
```

```
    //shadow = 0;
    //part6 reflect
    vec4 color = vec4(attenuation * originalColor * (ambient + (diffuse + specular) * (1.0 - shadow)), 1.0f);
    //vec3 ssr = reflectEffect(color.rgb, normal, textureCoord, fragPos, 1.0f);
    //ssr = 0.01 / (abs(ssr - texture(gDepthID, textureCoord).r) + 0.01);

    //part7 transparent mix
    vec4 colorTrans = texture(gTransColor, textureCoord);
    vec4 colorTransMixed;
    if (colorTrans != vec4(0.0f, 0.0f, 0.0f, 0.0f))
        colorTransMixed = mix(color, vec4(colorTrans.rgb, 1.0f), colorTrans.a);
    else
        colorTransMixed = color;
    //final

//  fColor = vec4(reflectEffect(color.rgb, normal, textureCoord, fragPos, texture(gDepthID, textureCoord).a), 1.0f);
    fColor = vec4(reflectEffect(colorTransMixed.rgb, normal, textureCoord, fragPos, texture(gDepthID, textureCoord).a), 1.0f);
    fColor.rgb = vec3(1.0) - exp(-fColor.rgb * 2.5);

}
```

2.5 顺序无关透明

初始化 T-FrameBuffer，在每个颜色分量设置为一个列表：

```
void Scene::InitOIT() {
    GLuint* data;

    glGenBuffers(NumOITBuffer, OITBuffers);
    glGenTextures(NumOITTexture, OITTextures);

    glBindTexture(GL_TEXTURE_2D, OITTextures[HeadPointerTexture]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_R32UI, width, height, 0, GL_RED_INTEGER, GL_UNSIGNED_INT, NULL);
    glBindTexture(GL_TEXTURE_2D, 0);

    glBindImageTexture(0, OITTextures[HeadPointerTexture], 0, GL_TRUE, 0, GL_READ_WRITE, GL_R32UI);

    constexpr int MemorySize = 18;
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, OITBuffers[HeadPointerInit]);
    glBufferData(GL_PIXEL_UNPACK_BUFFER, width * height * sizeof(GLuint), NULL, GL_STATIC_DRAW);
    data = (GLuint*)glMapBuffer(GL_PIXEL_UNPACK_BUFFER, GL_WRITE_ONLY);
    memset(data, 0x00, width * height * sizeof(GLuint));
    glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER);
```

```
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, OITBuffers[AtomicCounter]);
glBufferData(GL_ATOMIC_COUNTER_BUFFER, sizeof(GLuint), NULL, GL_DYNAMIC_COPY);

glBindBuffer(GL_TEXTURE_BUFFER, OITBuffers[Storage]);
glBufferData(GL_TEXTURE_BUFFER, MemorySize * width * height * sizeof(glm::vec4), NULL, GL_DYNAMIC_COPY);

glBindTexture(GL_TEXTURE_BUFFER, OITTextures[StorageTexture]);
glTexBuffer(GL_TEXTURE_BUFFER, GL_RGBA32UI, OITTextures[StorageTexture]);
glBindTexture(GL_TEXTURE_BUFFER, 0);

glBindBuffer(GL_TEXTURE_BUFFER, OITBuffers[StorageWorldPos]);
glBufferData(GL_TEXTURE_BUFFER, MemorySize * width * height * sizeof(glm::vec4), NULL, GL_DYNAMIC_COPY);

glBindTexture(GL_TEXTURE_BUFFER, OITTextures[StorageWorldPosTexture]);
glTexBuffer(GL_TEXTURE_BUFFER, GL_RGBA32F, OITTextures[StorageWorldPosTexture]);
glBindTexture(GL_TEXTURE_BUFFER, 0);

glBindBuffer(GL_TEXTURE_BUFFER, OITBuffers[StorageNormal]);
glBufferData(GL_TEXTURE_BUFFER, MemorySize * width * height * sizeof(glm::vec4), NULL, GL_DYNAMIC_COPY);

glBindTexture(GL_TEXTURE_BUFFER, OITTextures[StorageNormalTexture]);
glTexBuffer(GL_TEXTURE_BUFFER, GL_RGBA32F, OITTextures[StorageNormalTexture]);
glBindTexture(GL_TEXTURE_BUFFER, 0);
```

最后的半透明渲染流程：

```
int build_local_fragment_list(float limit) {
    uint current;
    int frag_count = 0;

    //current = texelFetch(head_pointer_image, ivec2(gl_FragCoord.xy), 0);
    current = imageLoad(head_pointer_image, ivec2(gl_FragCoord.xy)).x;

    while (current != 0x0 && current != 0xFFFFFFFF && frag_count < MAX_FRAGMENTS) {
        uvec4 item = imageLoad(list_buffer, int(current));
        vec4 worldPos_temp = imageLoad(list_buffer_worldPos, int(current));
        vec4 normal_temp = imageLoad(list_buffer_normal, int(current));
        if (uintBitsToFloat(item.w) <= limit || limit == 0) {
            fragments[frag_count] = item;
            worldPos[frag_count] = worldPos_temp;
            normal[frag_count] = normal_temp;
            frag_count++;
        }
        current = item.x;
    }
    return frag_count;
}
```

```
void sort_fragment_list(int frag_count) {

    int i;
    int j;
    for (i = 0; i < frag_count; i++) {
        for (j = i + 1; j < frag_count; j++) {
            float depth_i = uintBitsToFloat(fragments[i].w);
            float depth_j = uintBitsToFloat(fragments[j].w);
            if (depth_i > depth_j) {
                uvec4 temp = fragments[i];
                fragments[i] = fragments[j];
                fragments[j] = temp;
                vec4 temp_ = worldPos[i];
                worldPos[i] = worldPos[j];
                worldPos[j] = temp_;
                temp_ = normal[i];
                normal[i] = normal[j];
                normal[j] = temp_;
            }
        }
    }
}
```

```
vec4 calculate_final_color(int frag_count, vec3 base_color) {
    if (frag_count == 0) // no transparent object
        return vec4(0.0f);
    vec3 final_color = base_color;

    float blender = 1; // transparent rate, not non-transparent rate
    for (int i = frag_count - 1; i >= 0; i--) {
        vec4 color = unpackUnorm4x8(fragments[i].y);
        vec3 viewDirection = worldPos[i].xyz - uniViewPosition;
        float cos_theta = dot(normal[i].xyz, viewDirection) / (length(normal[i].xyz) * length(viewDirection));
        float sin_theta = sqrt(1 - cos_theta * cos_theta);
        sin_theta = 0.0f;
        float a_reformed = color.a + (1.0f - color.a) * sin_theta;

        final_color = mix(color.xyz, final_color,  /*blender * */1 - a_reformed);
        blender *= 1.0f - a_reformed;
    }

    //blender = 0.9f;
    return vec4(final_color, 1.0f - blender);
}
```

```
void main() {
    int frag_count;
    frag_count = build_local_fragment_list(texture(gDepthID, textureCoord).r);
    sort_fragment_list(frag_count);
//  _gTransColor = calculate_final_color(frag_count, texture(gAlbedoSpec, textureCoord).rgb);
//  vec3 color;
    _gTransColor = calculate_final_color(frag_count, vec3(0.0f));
    vec3 color = _gTransColor.rgb;
    //_gAlbedoSpec = vec4(color, 1.0f);
    //_gAlbedoSpec = vec4(frag_count * 1.0f / MAX_FRAGMENTS);
    _gPosition = texture(gPosition, textureCoord).xyz;
    _gNormal = texture(gNormal, textureCoord).xyz;
    _gDepthID = texture(gDepthID, textureCoord).rgba;
    if (frag_count != 0) {

        if (normal[0].w == 1) { //water
            //linear fog
            float deltaDepth = texture(gDepthID, textureCoord).r - uintBitsToFloat(fragments[0].w);
            _gTransColor = vec4(mix(color, vec3(165 / 255.9, 236 / 255.9, 250 / 255.9), _gTransColor.a),  1 - 1 / exp(deltaDepth));
////            _gTransColor = vec4(vec3(165 / 255.9, 236 / 255.9, 250 / 255.9), 1 - 1 / exp(deltaDepth));

            _gPosition = worldPos[0].xyz;
            _gNormal = normal[0].xyz;
            _gDepthID = vec4(vec3(uintBitsToFloat(fragments[0].w)), normal[0].w);
            //TODO
        }
    }
}
```

2.6 纹理映射

纹理映射是在 GBuffer 的光照 Pass 完成的：

```
// 从G缓冲中获取数据
float specularStrength = 0.5;
float roughness = texture(gAlbedoSpec, textureCoord).a;
vec3 fragPos = texture(gPosition, textureCoord).rgb;
vec3 originalColor = texture(gAlbedoSpec, textureCoord).rgb;
vec3 normal = texture(gNormal, textureCoord).rgb;
```

之后这些颜色信息、法线信息等将与光照模型一起进行后续计算

2.7 光照模型

CPU 中定义 Light 类型，具有漫反射、环境光等参数：

```
class Light {
public:
    glm::vec3 LightPos = glm::vec3(1.0f);
    glm::vec3 LightColor = glm::vec3(1.0f);

    glm::mat4 modelMat = glm::mat4(1.0f);

    GLuint VAOs[NumLightVAO]{};
    GLuint Buffers[NumLightBuffer]{};
    GLuint Program = 0;

    glm::vec3 ambient = glm::vec3(0.0f);
    glm::vec3 diffuse = glm::vec3(0.0f);
    glm::vec3 specular = glm::vec3(0.0f);

    float constant = 0.0f;
    float linear = 0.0f;
    float quadratic = 0.0f;

    void translate(const glm::mat4& translate) { this->modelMat = translate * modelMat; }
    void setAmbient(glm::vec3 ambient) { this->ambient = ambient; }
    void setDiffuse(glm::vec3 diffuse) { this->diffuse = diffuse; }
```

在最后的 GBuffer 光照渲染中，将 Lighjt 传入：

```
struct Light {
    vec3 position;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;

    float constant;
    float linear;
    float quadratic;
};

uniform float uniNear;
uniform float uniFar;

uniform Light light;
uniform vec3 lightColor;
uniform vec3 viewPos;
uniform mat4 lightSpaceMatrix;
```

然后用这些光照信息进行光照计算：

```
//part1
vec3 ambient = light.ambient * lightColor;
//part2
vec3 lightDir = normalize(light.position - fragPos);
float diff = max(dot(lightDir, normal), 0.0f);
vec3 diffuse = max(dot(lightDir, normal), 0.0f) * lightColor;
//part3
vec3 viewDir = normalize(viewPos - fragPos);
vec3 reflectDir = reflect(-lightDir, normal);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 64);
//spec = 0;
vec3 specular = light.specular * spec * lightColor * vec3(1 - roughness);
```

```
//float bias = max(0.005 * (1.0 - dot(normal, lightDir)), 0.001);
float bias = 0.0001;
vec4 FragPosLightSpace = lightSpaceMatrix * vec4(fragPos, 1.0);
float shadow = ShadowCalculation(bias, FragPosLightSpace);
//shadow = 0;
//part6 reflect
vec4 color = vec4(attenuation * originalColor * (ambient + (diffuse + specular) * (1.0 - shadow)), 1.0f);
```

最后进行反射计算：

```
vec3 reflectEffect(vec3 color, vec3 normal, vec2 uv, vec3 worldPos, float attr) {
    if (attr == 1.0) {
        //vec3 normal = normalDecode(texture2D(gnormal, texcoord.st).rg);
        vec3 viewReflectRay = reflect(normalize(worldPos - viewPos), normal);
        //viewReflectRay = normalize(viewReflectRay);
        vec2 uv2 = textureCoord.st * vec2(1280, 768);
        float c = (uv2.x + uv2.y) * 0.25;
        float jitter = mod(c, 1.0);
        float schlick = 0.02 + 0.98 * pow(1.0 - dot(viewReflectRay, normal), 5.0);
        //color = waterRayTarcing(viewPos + normal * (-viewPos.z / uniFar * 0.2 + 0.05), viewReflectRay, color, jitter, schlick);
        color = waterRayTarcing(worldPos + normal * (-worldPos.z / uniFar * 0.2 + 0.05), viewReflectRay, color, 0, schlick);
    }
    return color;
}
```

```
fColor = vec4(reflectEffect(colorTransMixed.rgb, normal, textureCoord, fragPos, texture(gDepthID, textureCoord).a), 1.0f);
fColor.rgb = vec3(1.0) - exp(-fColor.rgb * 2.5);
```

2.8 水面模型

首先渲染 OIT Buffer：

```
uniform float uniNear;
uniform float uniFar;

float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // back to NDC
    return (2.0 * uniNear * uniFar) / (uniFar + uniNear - z * (uniFar - uniNear));
}

void main() {
    //vec4 frag_color = shadeFragment();
    uint index = atomicCounterIncrement(index_counter) + 1;
    //fColor = vec4(vec3(index * 0.1f), 1.0f);
    uint old_head = imageAtomicExchange(head_pointer_image, ivec2(gl_FragCoord.xy), uint(index));
    //uint old_head = imageAtomicExchange(head_pointer_image, ivec2(0.5, 0.5), uint(1280 * 768));

    uvec4 item;
    item.x = old_head;
    item.y = packUnorm4x8(vec4(135 / 255.9, 206 / 255.9, 250 / 255.9, 0.5f));
    item.z = packUnorm4x8(vec4(normal, 1.0f));
    //item.w = floatBitsToUint(depth);
    item.w = floatBitsToUint(LinearizeDepth(gl_FragCoord.z) / uniFar);
    //item.w = floatBitsToUint(LinearizeDepth(depth) / uniFar);

    imageStore(list_buffer, int(index), item);
    imageStore(list_buffer_worldPos, int(index), vec4(worldPos, 1.0f));
    imageStore(list_buffer_normal, int(index), vec4(normal, 1.0f));
}
```

在 OIT Blend Pass 进行进一步的深度判断，确定水面的不透明度：

```
if (frag_count != 0) {

    if (normal[0].w == 1) { //water
        //linear fog
        float deltaDepth = texture(gDepthID, textureCoord).r - uintBitsToFloat(fragments[0].w);
        _gTransColor = vec4(mix(color, vec3(165 / 255.9, 236 / 255.9, 250 / 255.9), _gTransColor.a),  1 - 1 / exp(deltaDepth));
//          _gTransColor = vec4(vec3(165 / 255.9, 236 / 255.9, 250 / 255.9), 1 - 1 / exp(deltaDepth));

        _gPosition = worldPos[0].xyz;
        _gNormal = normal[0].xyz;
        _gDepthID = vec4(vec3(uintBitsToFloat(fragments[0].w)), normal[0].w);
        //TODO
    }
}
```

2.9 天空盒

天空盒是在一般的纹理多边形上作出一些简单改变，主要表现在纹理读取上：

```
{
    GLuint textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        std::string filename = std::string(TexturePath) + faces[i];
        unsigned char* data = stbi_load(filename.c_str(), &width, &height, &nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
                0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data
            );
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << filename << std::endl;
            stbi_image_free(data);
        }
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

然后需要设定 Cube Texture 的坐标：

```
float skyboxVertices[] = {
    // positions
    -1.0f,  1.0f, -1.0f,
    -1.0f, -1.0f, -1.0f,
     1.0f, -1.0f, -1.0f,
     1.0f, -1.0f, -1.0f,
     1.0f,  1.0f, -1.0f,
    -1.0f,  1.0f, -1.0f,

    -1.0f, -1.0f,  1.0f,
    -1.0f, -1.0f, -1.0f,
    -1.0f,  1.0f, -1.0f,
    -1.0f,  1.0f, -1.0f,
    -1.0f,  1.0f,  1.0f,
    -1.0f, -1.0f,  1.0f,

     1.0f, -1.0f, -1.0f,
     1.0f, -1.0f,  1.0f,
     1.0f,  1.0f,  1.0f,
     1.0f,  1.0f,  1.0f,
     1.0f,  1.0f, -1.0f,
     1.0f, -1.0f, -1.0f,
```

```
glGenVertexArrays(NumVAO, this->VAOs);
glCreateBuffers(NumBuffer, this->Buffers);

glNamedBufferStorage(this->Buffers[ArrayBuffer], sizeof(skyboxVertices), skyboxVertices, 0);

glBindVertexArray(this->VAOs[VAO]);
glBindBuffer(GL_ARRAY_BUFFER, this->Buffers[ArrayBuffer]);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);
glEnableVertexAttribArray(0);
```

2.11 纹理残留

首先进行手动曲面细分：

```
FitTexturedPlane::FitTexturedPlane(glm::vec3 position, GLfloat points[], GLint sizeofPoints, GLuint indices[], GLint sizeofIndices, ShaderInfo
    :ViewObject(ViewObjectEnum::FitTextureObject)
{
    //手动进行第一次细分 可以修改
    GLfloat* pointGen = new GLfloat[5 * 4 * part * part];
    for (int i = 0; i < part; i++)
        for (int j = 0; j < part; j++)
        {//for element
            // x y z s t
            pointGen[5 * 4 * (i * part + j) + 5 * 0 + 0] = (points[2 * 5 + 0] - points[0 * 5 + 0]) / part * i + points[0 * 5 + 0];
            pointGen[5 * 4 * (i * part + j) + 5 * 0 + 1] = (points[2 * 5 + 1] - points[0 * 5 + 1]) / part * i + points[0 * 5 + 1];
            pointGen[5 * 4 * (i * part + j) + 5 * 0 + 2] = (points[1 * 5 + 2] - points[0 * 5 + 2]) / part * j + points[0 * 5 + 2];
            pointGen[5 * 4 * (i * part + j) + 5 * 0 + 3] = (points[2 * 5 + 3] - points[0 * 5 + 3]) / part * i + points[0 * 5 + 3];
            pointGen[5 * 4 * (i * part + j) + 5 * 0 + 4] = (points[1 * 5 + 4] - points[0 * 5 + 4]) / part * j + points[0 * 5 + 4];

            // x y z s t
            pointGen[5 * 4 * (i * part + j) + 5 * 1 + 0] = (points[2 * 5 + 0] - points[0 * 5 + 0]) / part * (i + 1) + points[0 * 5 + 0];
            pointGen[5 * 4 * (i * part + j) + 5 * 1 + 1] = (points[2 * 5 + 1] - points[0 * 5 + 1]) / part * (i + 1) + points[0 * 5 + 1];
            pointGen[5 * 4 * (i * part + j) + 5 * 1 + 2] = (points[1 * 5 + 2] - points[0 * 5 + 2]) / part * j + points[0 * 5 + 2];
            pointGen[5 * 4 * (i * part + j) + 5 * 1 + 3] = (points[2 * 5 + 3] - points[0 * 5 + 3]) / part * (i + 1) + points[0 * 5 + 3];
            pointGen[5 * 4 * (i * part + j) + 5 * 1 + 4] = (points[1 * 5 + 4] - points[0 * 5 + 4]) / part * j + points[0 * 5 + 4];

            // x y z s t
            pointGen[5 * 4 * (i * part + j) + 5 * 2 + 0] = (points[2 * 5 + 0] - points[0 * 5 + 0]) / part * i + points[0 * 5 + 0];
            pointGen[5 * 4 * (i * part + j) + 5 * 2 + 1] = (points[2 * 5 + 1] - points[0 * 5 + 1]) / part * i + points[0 * 5 + 1];
            pointGen[5 * 4 * (i * part + j) + 5 * 2 + 2] = (points[1 * 5 + 2] - points[0 * 5 + 2]) / part * (j + 1) + points[0 * 5 + 2];
            pointGen[5 * 4 * (i * part + j) + 5 * 2 + 3] = (points[2 * 5 + 3] - points[0 * 5 + 3]) / part * i + points[0 * 5 + 3];
```

然后对该曲面特定地建立一个 framebuffer 用于存储残留的信息：

```
glGenVertexArrays(NumFitPlaneVAO, this->VAOs);
glCreateBuffers(NumFitPlaneBuffer, this->Buffers);

glNamedBufferStorage(this->Buffers[FitPlaneArrayBuffer], 5 * 4 * part * part * sizeof(GLfloat), pointGen, 0);

glBindVertexArray(this->VAOs[FitPlaneVAO]);
glBindBuffer(GL_ARRAY_BUFFER, this->Buffers[FitPlaneArrayBuffer]);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat), 0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_TRUE, 5 * sizeof(GLfloat), (void*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(1);

glNamedBufferStorage(this->Buffers[FitTextureArrayBuffer], sizeofPoints, points, 0);
glNamedBufferStorage(this->Buffers[FitTextureElementBuffer], sizeofIndices, indices, 0);

glBindVertexArray(this->VAOs[FitTextureVAO]);
glBindBuffer(GL_ARRAY_BUFFER, this->Buffers[FitTextureArrayBuffer]);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->Buffers[FitTextureElementBuffer]);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat), 0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_TRUE, 5 * sizeof(GLfloat), (void*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(1);
```

```
glGenFramebuffers(NumFitFrameBuffer, FrameBuffers);
glBindFramebuffer(GL_FRAMEBUFFER, FrameBuffers[FitFrameBuffer]);

glGenTextures(1, &Textures[FitHeightTexture]);
glBindTexture(GL_TEXTURE_2D, Textures[FitHeightTexture]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB32F, FitTextureW, FitTextureH, 0, GL_RGB, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, Textures[FitHeightTexture], 0);

GLuint attachments[1] = { GL_COLOR_ATTACHMENT0 };
glDrawBuffers(1, attachments);

unsigned int rboDepth;
glGenRenderbuffers(1, &rboDepth);
glBindRenderbuffer(GL_RENDERBUFFER, rboDepth);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, FitTextureW, FitTextureH);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, rboDepth);
// finally check if framebuffer is complete
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    std::cout << "Framebuffer not complete!" << " " << glCheckFramebufferStatus(GL_FRAMEBUFFER) << std::endl;
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

在 Shader 中进行第二次自动细分：

```
#version 450 core

layout(vertices = 4)out;

in vec3 fragPos[];
in vec2 textureCoord[];
out vec2 textureCoord_CTS_ETS[];
out vec3 fragPos_CTS_ETS[];

void main() {
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
    fragPos_CTS_ETS[gl_InvocationID] = fragPos[gl_InvocationID];
    textureCoord_CTS_ETS[gl_InvocationID] = textureCoord[gl_InvocationID];

    float t = 64;

    // 下面设置细分曲面程度
    gl_TessLevelInner[0] = t;      // 内部划分3条垂直区域，即内部新增2列顶点
    gl_TessLevelInner[1] = t;      // 内部划分4条水平区域，即内部新增3行顶点

    gl_TessLevelOuter[0] = t;      // 左边2条线段
    gl_TessLevelOuter[1] = t;      // 下边3条线段
    gl_TessLevelOuter[2] = t;      // 右边4条线段
    gl_TessLevelOuter[3] = t;      // 上边5条线段
}
```

```
//}

float u = gl_TessCoord.x;
float omu = 1 - u;
float uniV = gl_TessCoord.y;
float omv = 1 - uniV;
//fragPos =
//   (omu * omv * gl_in[0].gl_Position +
//   u * omv * gl_in[1].gl_Position +
//   u * uniV * gl_in[2].gl_Position +
//   omu * uniV * gl_in[3].gl_Position).xyz;

fragPos_ET_GS =
     (omu * omv * fragPos_CTS_ETS[1] +
          u * omv * fragPos_CTS_ETS[0] +
          u * uniV * fragPos_CTS_ETS[2] +
          omu * uniV * fragPos_CTS_ETS[3]).xyz;
textureCoord_ET_GS =
     (omu * omv * textureCoord_CTS_ETS[1] +
          u * omv * textureCoord_CTS_ETS[0] +
          u * uniV * textureCoord_CTS_ETS[2] +
          omu * uniV * textureCoord_CTS_ETS[3]);
//fragPos.y = -fragPos.y;
```

然后用一个额外的 Program 进行单独的 HeightMap 更新：

```
void main() {
    float coe = 0.5;
    float base = 1;
    float baseLine = min((sqrt(textureCoord.x * coe) * sqrt(textureCoord.y * coe) * sqrt((1 - textureCoord.x) * coe)
        * sqrt((1 - textureCoord.y) * coe)) * 16, base);
    if (rst == 1) {
        heightMapNew = baseLine;
    }
    else {

          float temp = texture(heightMap, textureCoord.xy).r;
          float R2 = pow(radius, 2);
          float D2 = pow(radius * 5, 2);
          float deltaX2 = pow(fragPos.x - uniObjPos.x, 2);
          float deltaZ2 = pow(fragPos.z - uniObjPos.z, 2);
          float delta1 = radius - sqrt(R2 - deltaX2 - deltaZ2);
          float delta2 = radius * 5 - sqrt(D2 - deltaX2 - deltaZ2);
```

```
            if (delta1 <= baseLine) {
                if (dot(uniObjVel.xz, fragPos.xz - uniObjPos.xz) >= 0) {
                    heightMapNew = min(delta1, temp);
                }
                else heightMapNew = temp;
            }
            else if (delta2 <= baseLine) {
                if (dot(uniObjVel.xz, fragPos.xz - uniObjPos.xz) >= 0) {
                    float linjie = radius * 5 - sqrt(D2 - R2 + pow(radius - baseLine, 2));
                    float x = (delta2 - linjie) / (baseLine - linjie);
                    if (temp < baseLine-0.1)
                        heightMapNew = temp;
                    else
                        heightMapNew = max(temp, baseLine + 2 * log(6 * x + 1.5) / (6 * x + 1.5) - 0.54);
                }
                else heightMapNew = temp;
            }
            else {
                float volosity = min(abs(baseLine - temp), 0.999);
                volosity = -(volosity - 0) * (volosity - 1) * 0.001;
                heightMapNew = (baseLine - temp) * volosity + temp;
            }
    }
```

2.12 游戏场景

每个游戏场景设计为四种状态：Idle， Hint, Play 与 Success，以及自动生成空气墙等功能：

```
class GameSceneBase
{
public:
    enum class GameState { IdleState = 0, HintState, PlayState, SuccessState };


    GameSceneBase();
    void Rotate(const GLfloat& angle, const glm::vec3& axis);
    void Translate(const glm::vec3& displacement);
    void Scale(const glm::vec3& scaler);
    bool CheckSuccess() const;
    GameSceneBase::GameState GetState() const;
    virtual void Update() = 0;
    virtual void Idle() = 0;
    virtual void Hint() = 0;
    virtual void Play() = 0;
    virtual void Success() = 0;
protected:
    // generate air wall, vertices should be (x1, z1, x2, z2, ...) in anticlockwise order
    std::vector<std::shared_ptr<GameObject>> _objects;
    GameState _state = GameState::IdleState;
    void GenerateAirWall(const std::vector<GLfloat>& vertices);

    float _lastTime;
};
```

地面小岛谜题额外增加石柱：

```
class DesertScene final : public GroundGameScene
{
public:
    DesertScene(const GLfloat& width = 300.0f, const GLfloat& height = 300.0f);
    virtual void Idle();
    virtual void Hint();
    virtual void Play();
    virtual void Success();
    virtual void Update();

    enum class BarState { IdleBar, ActiveBar, WrongBar, SuccessBar};

private:

    // generate random target list and puzzle bars
    void GenerateRandomList(const int& num);
    void GenerateBars(const int& num);
    std::shared_ptr<FitTexturedPlane> _areaGround;
    std::vector<std::shared_ptr<GameObject>> _puzzleBars;
    void ChangeBarState(const int& index, const BarState& state);
    std::vector<int> _targetList;
    std::vector<int> _userList;
    std::array<bool, 256> _activeBarMap{0};
    GLfloat _groundWidth;
    GLfloat _groundHeight;
    int _startShowIndex = 0;
```

飞机谜题增加半透明圆环：

```
class PlaneGameScene final : public GameSceneBase
{
public:
    PlaneGameScene(const int& ringNum = 16);
    int GetRingNumber() const;
    virtual void Update();
    virtual void Idle();
    virtual void Hint();
    virtual void Play();
    virtual void Success();

private:
    void GenerateRings(const int& ringNum = 16);
    std::vector<std::shared_ptr<GameObject>> _rings;
    std::shared_ptr<GameObject> _well;
    std::array<bool, 256> _goaledRings{ 0 };
    int _goaledNumber = 0;
};
```

具体实现代码过于冗长，此处略去

2.13 屏幕截取

用 glReadPixels 可以简单实现：

```
    constexpr int components = 3;
    constexpr int BMP_Header_Length = 54;
    int lengthW = (width * components + 3) / 4 * 4;
    std::cout << "Screen Shotting..." << std::endl;

    std::string&& realFilename = (std::string(ScreenShotPath) + filename);
    std::cout << "\tScreen Shot Filename = " << realFilename << std::endl;
    char * imageBuf = (char*)malloc(lengthW * height * sizeof(char));
    if (!imageBuf)
        throw("ScreenShot:: Memory NOT ENOUGH");

    glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
    glReadPixels((GLint)0, (GLint)0, (GLint)width, (GLint)height,
        GL_RGB, GL_UNSIGNED_BYTE, imageBuf);

    // reverse the image

    for (int i = 0; i < height / 2; i++)
        for (int j = 0; j < lengthW; j++)
        {
            int pos = i * lengthW + j;
            int spos = (height - 1 - i) * lengthW + j;
            swap(imageBuf[pos], imageBuf[spos]);
        }

    // write the image into file
    stbi_write_jpg(realFilename.c_str(), width, height, 3, imageBuf, 0);
```

2.14 用户交互

对于键盘交互，我们简单地通过一个按键回调函数来实现。回调函数通过 key 以及 action 两个参数来判断用户的行为，并由此改变响应变量的值。

```
if (key == GLFW_KEY_SPACE && action == GLFW_RELEASE)
{
    key_space_pressed = false;
}

if (key == GLFW_KEY_R && action == GLFW_PRESS) {
    key_r_pressed = true;
    printf("Key R\n");
}
if (key == GLFW_KEY_Q && action == GLFW_PRESS) {
    key_q_pressed = true;
    printf("Key Q\n");
}
if (key == GLFW_KEY_H && action == GLFW_PRESS) {
    key_h_pressed = true;
    printf("Key H\n");
}

if (key == GLFW_KEY_W && action == GLFW_PRESS) {
    key_w_pressed = true;
    printf("Key W\n");
}
```

相比之下，鼠标交互略显复杂。按照我们的设计，游戏要依据用户是否按下鼠标左键拥有两套交互逻辑。因此，我们先要给鼠标按键绑定一个回调函数，并让它改变相应变量的值：

```
void Interaction::MouseButtonCallback(GLFWwindow* window, int button, int action, int mods)
{
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS) {
        left_button_pressed = 1;
        left_button_pressed_just = 1;
        printf("GET LEFT BUTTON PRESS\n");
    }

    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_RELEASE) {
        left_button_pressed = 0;
        printf("GET LEFT BUTTON RELEASE\n");
    }

    if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS) {
        right_button_pressed = 1;
        right_button_pressed_just = 1;
        printf("GET RIGHT BUTTON PRESS\n");
    }

    if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_RELEASE) {
        right_button_pressed = 0;
        printf("GET RIGHT BUTTON RELEASE\n");
    }
}
```

然后，我们在鼠标的回调函数中根据这些变量实现两套交互逻辑：

```
if (left_button_pressed) {
    // modify camera view dir
    if (left_button_pressed_just) {
        lastX = xposf;
        lastY = yposf;
        left_button_pressed_just = 0;
    }
    if (mouseReverseFlag)
        camera.ProcessLeftMouseMovement(tmp_xoffset, tmp_yoffset);
    else
        camera.ProcessLeftMouseMovement(-tmp_xoffset, -tmp_yoffset);
}
else if (right_button_pressed) {
}
else {
    // modify game obj dir
    // will be consumed by the loop in ViewWorld.cpp
    xoffset = tmp_xoffset;
    yoffset = tmp_yoffset;
    // below: not used
    yaw += xoffset;
    pitch += yoffset;
    glm::vec3 new_front;
```

接下来还有一个问题：鼠标回调函数的刷新率是低于游戏画面的刷新率的。为了确保移动一次鼠标后游戏内的物体/视角只会被移动一次，我们设计了一个类似于生产者-消费者模型的接口，确保鼠标移动的 offset 只会被游戏主循环读取一次：

```
GLfloat Interaction::ReadXoffset()
{
    GLfloat ret = xoffset;
    xoffset = 0;
    return ret;
}

GLfloat Interaction::ReadYoffset()
{
    GLfloat ret = yoffset;
    yoffset = 0;
    return ret;
}
```

2.15 镜头操作

镜头的状态由以下几个向量决定，分别表示镜头的位置、朝向、上方方向、右侧方向（由前两者外积计算得出）。由它们还可以衍生出欧拉角等更为直观的镜头参数。对镜头的操作本质上也都是改变这几个向量。

```
// camera Attributes
glm::vec3 Position;
glm::vec3 Front;
glm::vec3 Up;
glm::vec3 Right;
```

```
glm::mat4 GetViewMatrix()
{
    return glm::lookAt(Position,
}
```

我们的镜头支持软跟踪。当镜头的朝向改变时，Front 向量并不会立刻被设成新的朝向，而是由旧的朝向和新的朝向进行一个线性组合。迭代的帧数越多，镜头朝向与目标朝向差别越小，它移动的速度也就越缓慢。这样，可以让游戏的主角不总是被锁定在屏幕的正中央、但又不会偏离太多，让画面更有动态交互感。

```
void SetFrontDir(glm::vec3 in_front, bool hard = false)
{
    if (hard) {
        Front = in_front;
    }
    else {
        // smooth transition
        Front = (1 - FOLLOW_SPEED) * Front + FOLLOW_SPEED * in_front;
    }
    updateCameraVectors();
    this->updateGlobalData();
}
```

2.16 碰撞箱

碰撞箱在读入模型时自动生成，是最小的能包裹整个模型的长方体。同时生成的还有最小的能包裹模型的球体，称其半径为“碰撞半径”。在处理碰撞时，先通过两个模型间的碰撞半径进行一次快速筛选。如果模型中心间的距离大于二者碰撞半径之和，就意味着碰撞不可能发生：

```
bool GameObject::collisionPossible(GameObject& obj) {
    // fast collision dectection
    // treat all objects as balls
    glm::vec3 coord = this->getPosition();
    glm::vec3 coord2 = obj.getPosition();
    GLfloat dist_sq = (coord[0] - coord2[0]) * (coord[0] - coord2[0])
        + (coord[1] - coord2[1]) * (coord[1] - coord2[1])
        + (coord[2] - coord2[2]) * (coord[2] - coord2[2]);
    // printf("thisR: %.1f    thatR: %.1f    dist: %.1f", hitRadius, h
    return sqrt(dist_sq) <= (hitRadius + obj.hitRadius);
}
```

接下来，我们再用碰撞箱进行判断：

```cpp
bool GameObject::collision(GameObject& obj) {
    if(!collisionPossible(obj)) return false;// fast detection
    // determine if this is in obj
    glm::vec4 coord = glm::vec4(getPosition(), 1);
    coord = glm::inverse(obj.viewObj->GetM()) * coord;
    coord /= coord[3];
    /*std::cout << "coord ";  printVec(coord);
    std::cout << "min ";  printVec(minVertexCoord);
    std::cout << "max ";  printVec(maxVertexCoord);*/
    for (int i = 0; i < 3; i++)
        if (!(coord[i] > obj.minVertexCoord[i] && coord[i] < obj.maxVertexCoord[i]))
            return false;
    // determine if obj is in this
    // not implemented
    return true;
}
```

如果二者确实发生了碰撞，按照物体的 fixed 属性，我们有两种处理方案。如果 fixed 为真，就撤销导致碰撞发生的变换；否则，将被撞物体按照作用量最小的方向弹开。

```cpp
if (collision(*ptr)) { // fast dectection
    if (ptr->fixed) {
        // undo translation
        this->viewObj->Translate(-vec);
        // set the plane velocity to zero
        this->setVelocity(glm::vec3(0.0f));
        return;
    }
    else {
        static const GLfloat delta = 1e-3f;
        int minStepNeeded = 500;
        glm::vec3 optimizedDir = { 0, 0, delta };
        for(GLfloat x = -delta; x <= delta; x+= delta / 2)
            for (GLfloat y = -delta; y <= delta; y += delta / 2)
                for (GLfloat z = -delta; z <= delta; z += delta / 2) {
                    glm::vec3 deltaDir = { x, y, z };
                    int steps = 1;
                    ptr->translate(deltaDir * GLfloat(1));
                    while (collision(*ptr) && steps < minStepNeeded) {// 倍增
                        ptr->translate(deltaDir * GLfloat(steps));
                        steps *= 2;
                    }
                    ptr->translate(-deltaDir * GLfloat(steps));
                    if (steps < minStepNeeded) {
                        minStepNeeded = steps;
                        optimizedDir = deltaDir;
                    }
                }
        ptr->translate(optimizedDir * GLfloat(minStepNeeded));
        // std::cout << "minsteps = " << minStepNeeded << "\n";
    }
```

2.17 实时 CG

CG 通过硬编码调用变换函数实现。每当画面刷新时，调用一次 CG 的绘制函数并更新一帧。具体的绘制代码在此省略，以下是 CG 绘制函数的主要结构：

```
void UpdateTakeoffCG() {
    static int tick = 0;
    static const int tickPerSec = 60;
    // percentage of CG displayed
    // begin = 0.0, finished = 1.0
    GLfloat progress = GLfloat(tick) / (10.0f * tickPerSec);
    if (progress > 1.0) {
        tick = 0;
        Interaction::displayTakeOffCGFlag = false;
        airplane->setPower(1.0);
        airplane->setVelocity(airplane->localFront * 10.0f);
        return;
    }
    tick++;

    // initialize
    if (tick == 1) {
```

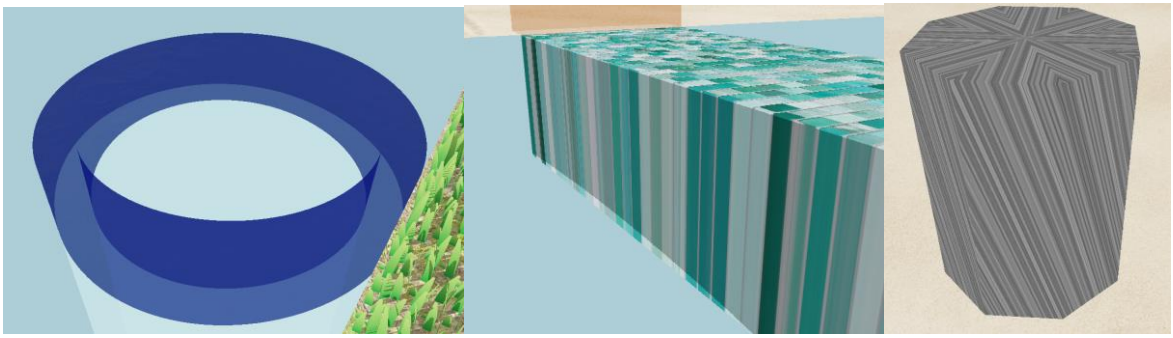
```
    // update airplane
    if(progress > 0.2f && progress < 0.7f)
        airplane->changePitch(0.2f * progress);
    else if (progress > 0.7f) {
        airplane->changePitch(-0.2f * (1.0f -progress));
    }
    airplane->translate(airplane->localFront * 20.0f * (

    // update camera
    glm::vec3 cameraPos = Interaction::camera.GetViewPos
    if (progress < 0.3f) {
        cameraPos += airplane->getLeftDir() * 0.3f;
        Interaction::camera.SetPosition(cameraPos);
    }
    else if (progress < 0.5f) {
```

static 变量 tick 表示当前正在绘制的帧数。当绘制第一帧时，会执行一些初始化的操作（例如，设定相机的初始位置）；当最后一帧绘制完毕之后，将一个 flag 设成 false 通知上层函数，并在将画面的控制权交还给用户之前，会执行一些善后工作。在绘制其他帧时，则按照更新逻辑更新画面中的物体与镜头。

3 功能测试

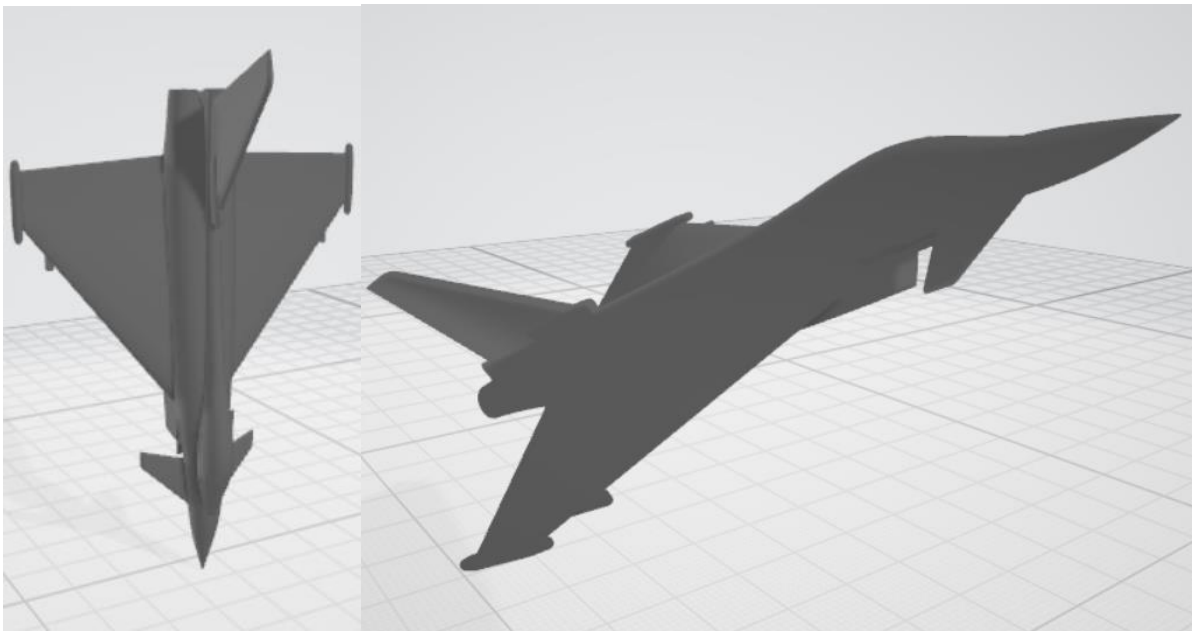
3.1 体素测试

以下分别是圆柱、立方体、棱柱。



3.2 三维网格导入导出测试

系统会从 Obj 文件读入模型，并在按下 F2 键后输出经过变换后的模型。

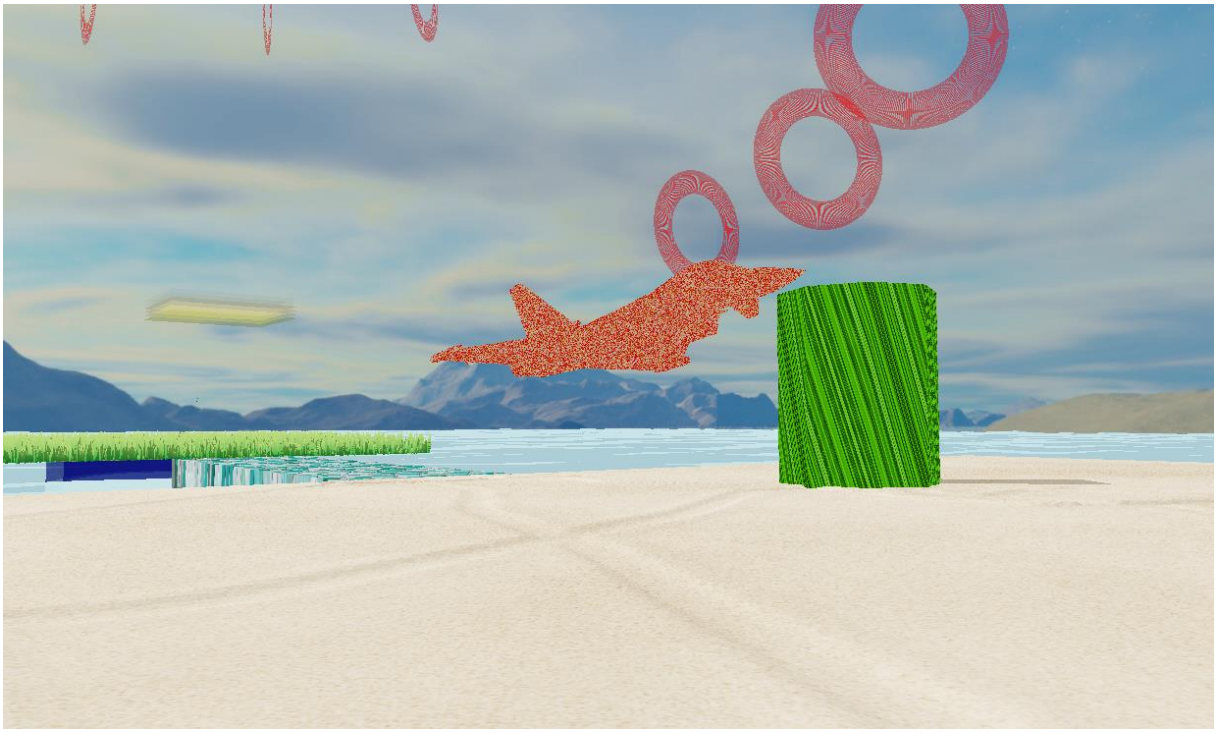


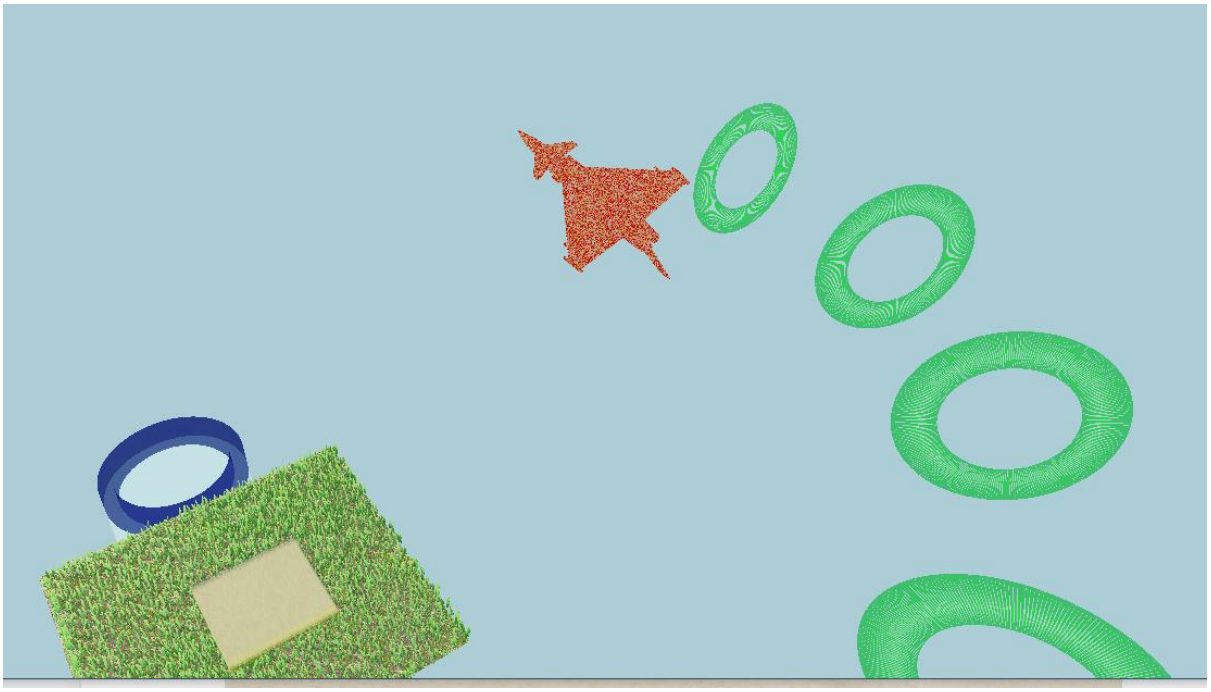
3.3 材质纹理测试

沙地的纹理会根据走过的痕迹而动态变化：

3.4 几何变换测试

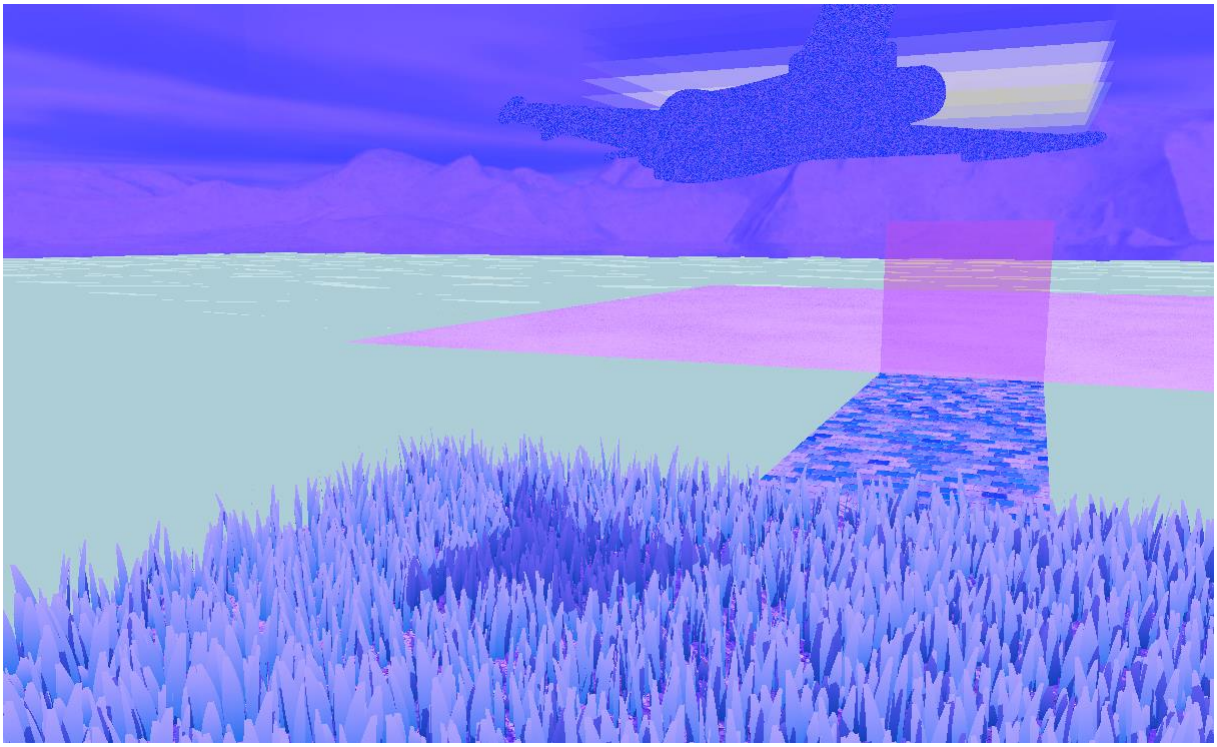
我们在游戏中使用了两段 CG，通过硬编码几何变换来控制镜头和物体的移动。

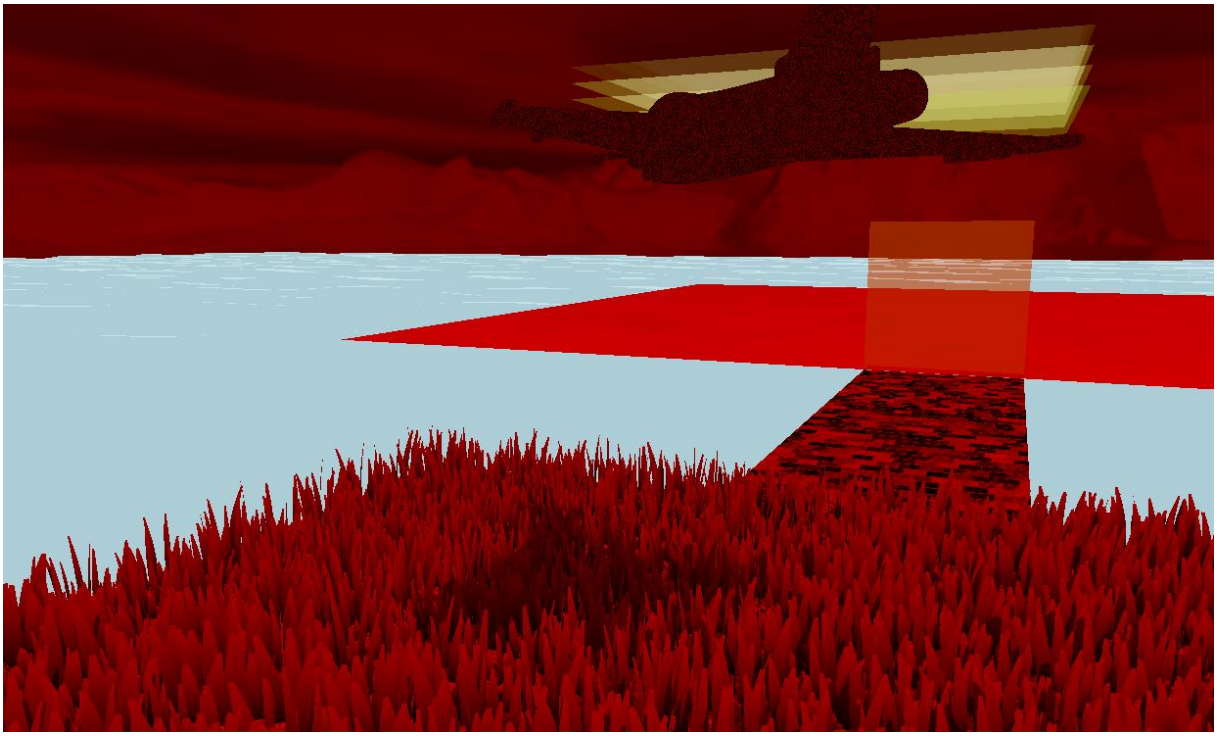




3.5 基本光照测试

按下 F3 后，可以观察场景在不同光源下的表现效果。

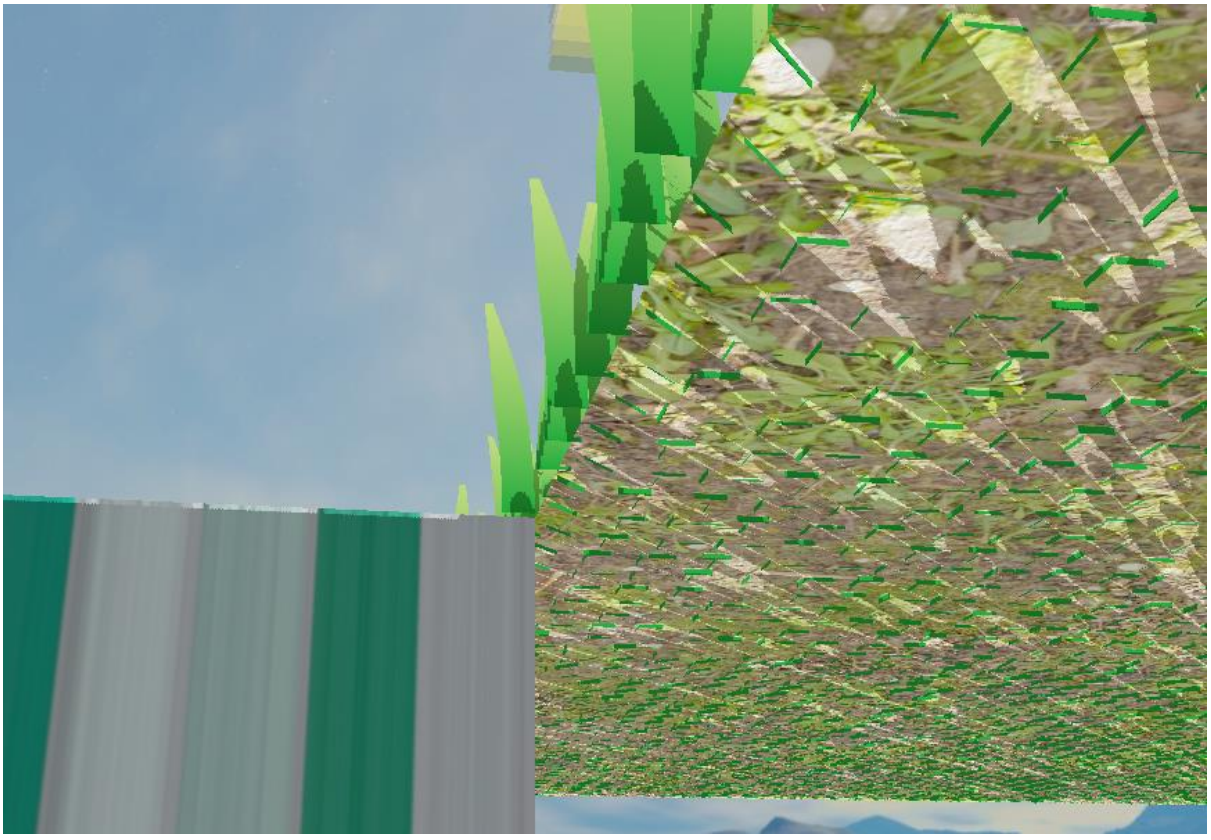




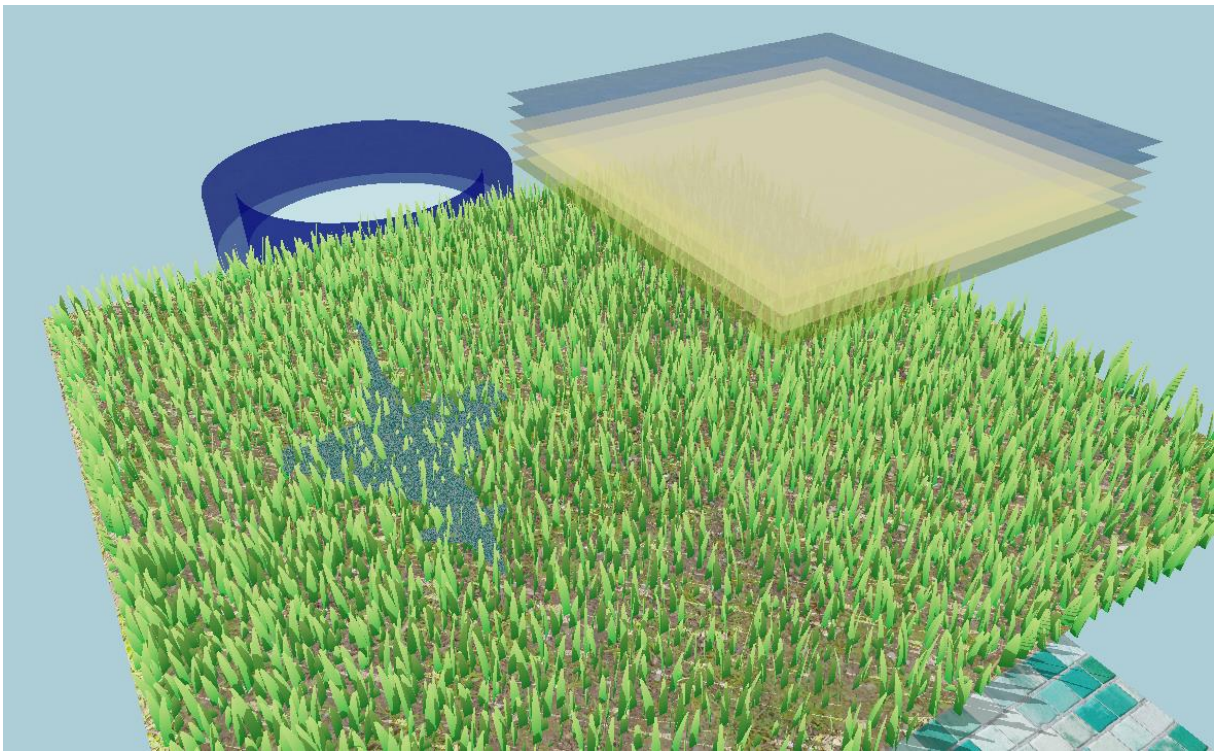
3.6 漫游测试

按下 P 键后，进入漫游模式，可以抵达游戏正常无法抵达的区域、从不同视角观察。

地面之下：



游戏的出生点：



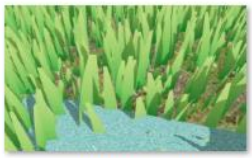
天空关卡：

3.7 屏幕截取测试

PC › Documents › ZJU › CG › CGProject › new_ver › screenshot


IMG_2021-01-17_15-33-29.jpg

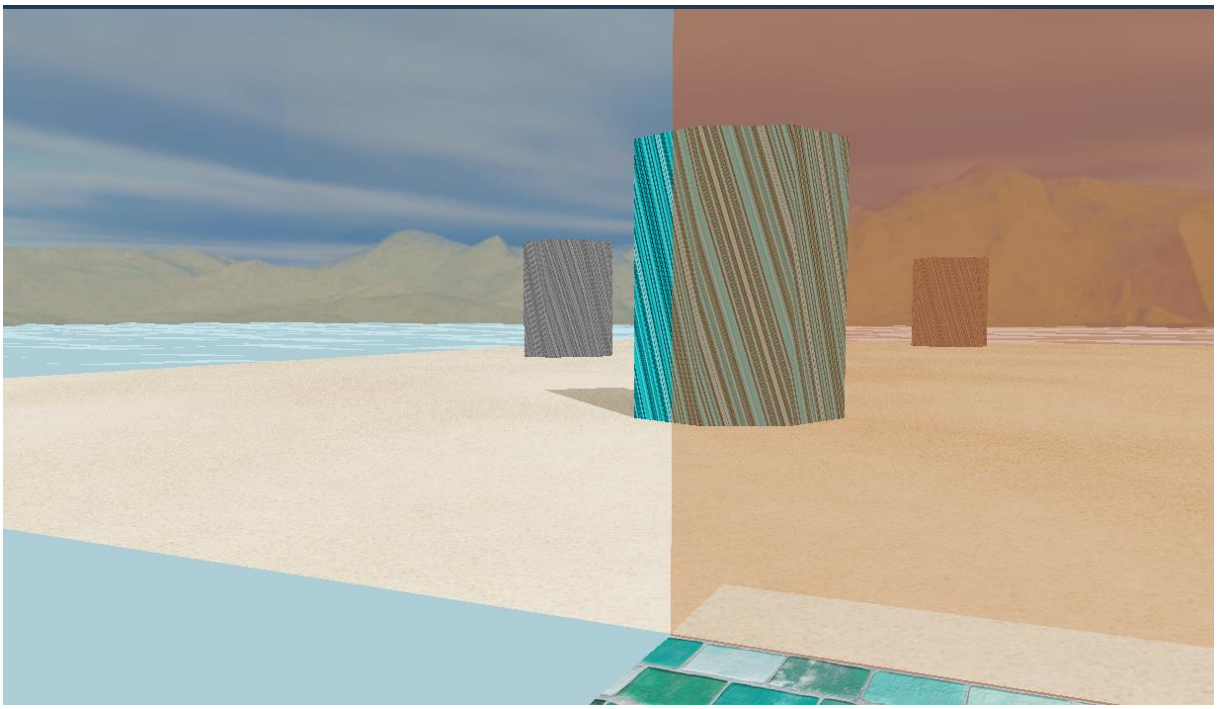

IMG_2021-01-17_15-33-32.jpg


IMG_2021-01-18_20-10-28.jpg

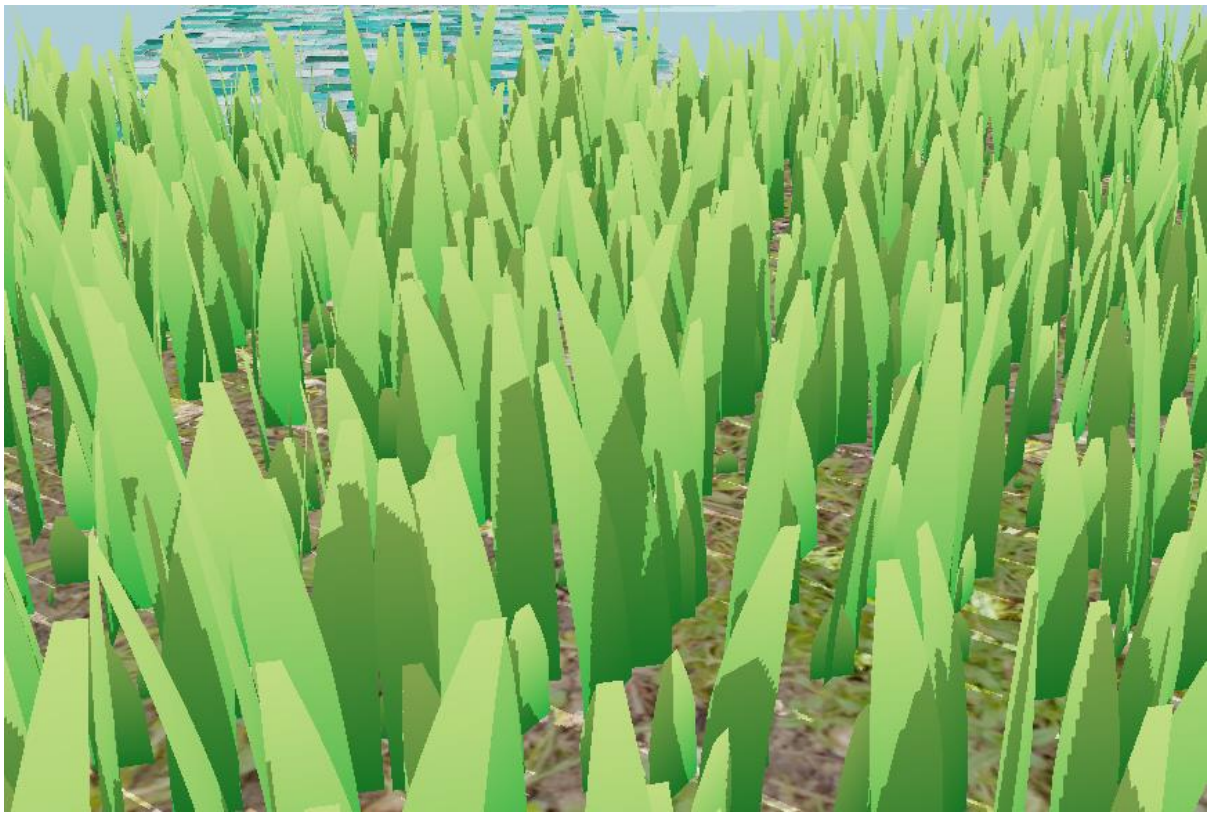

IMG_2021-01-18_20-10-30.jpg

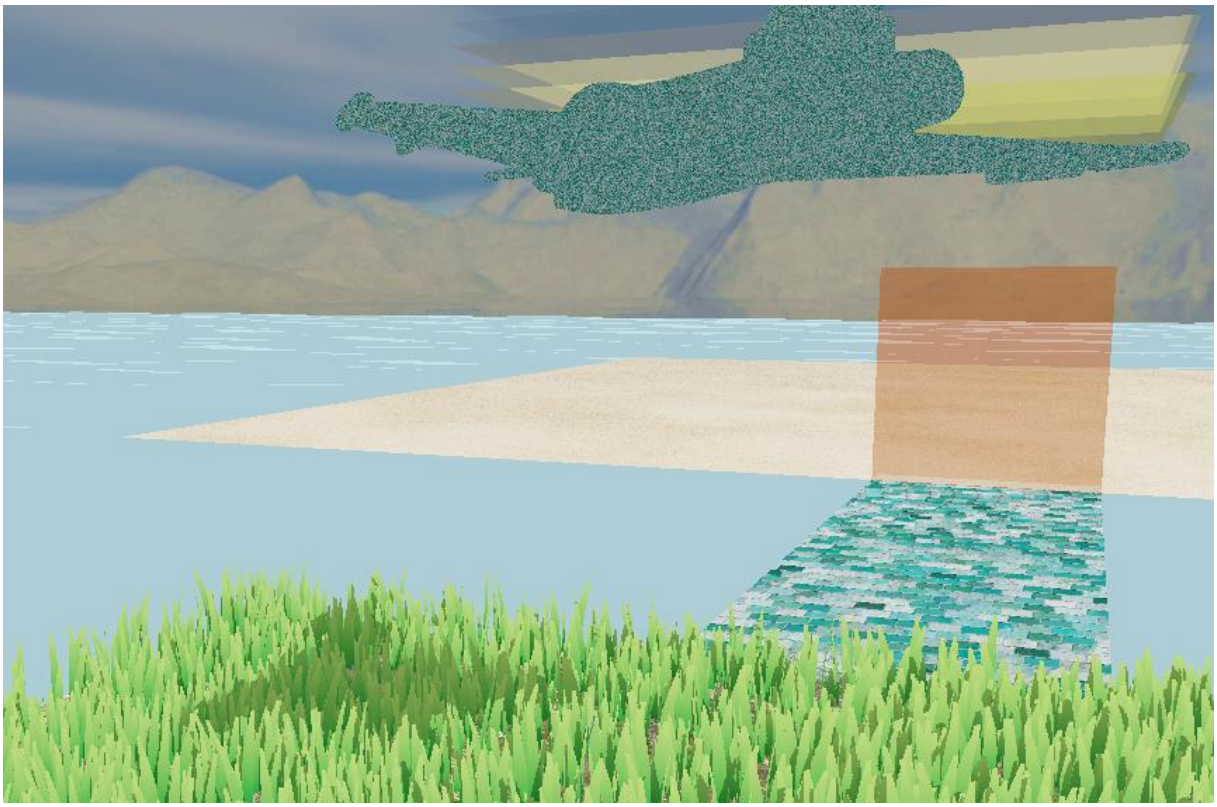
3.8 碰撞测试

所有模型在读入时会自动地生成一个碰撞箱，且所有墙面会自动生成空气墙。此外，地面关卡“接近触发灯柱”的效果天空关卡中“穿过圆环”的效果也都利用了碰撞检测算法。



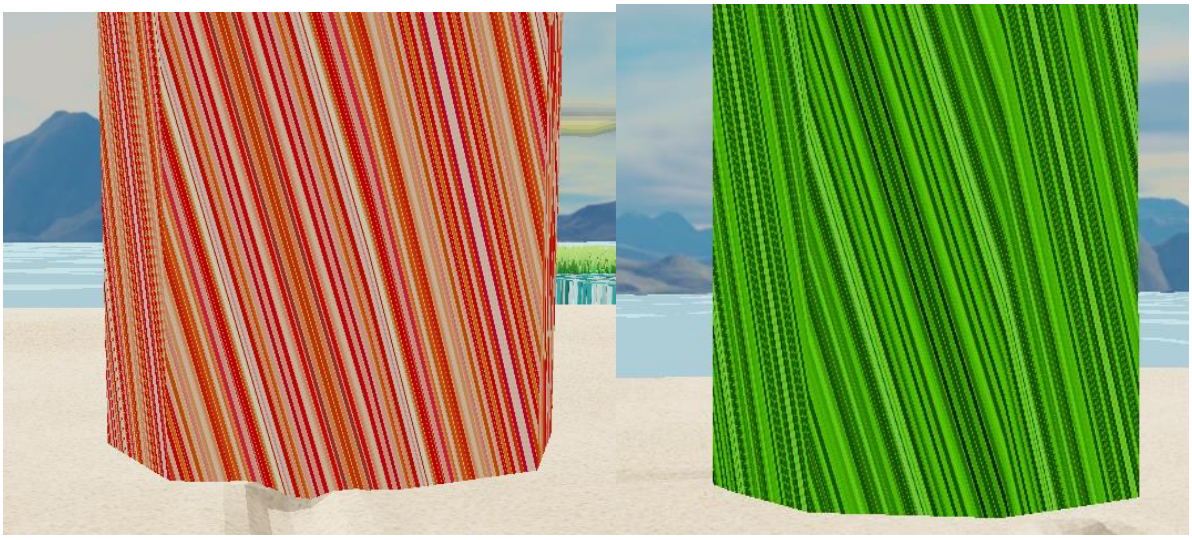
3.9 实时阴影测试





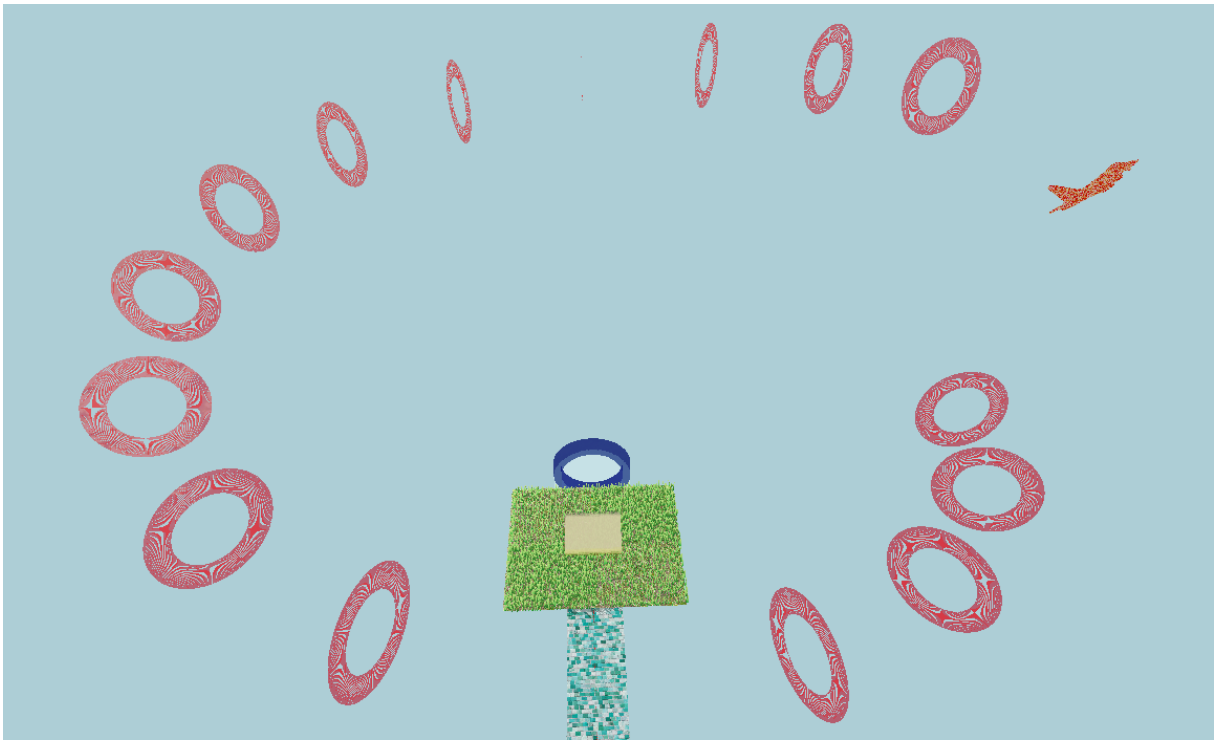
4 系统测试

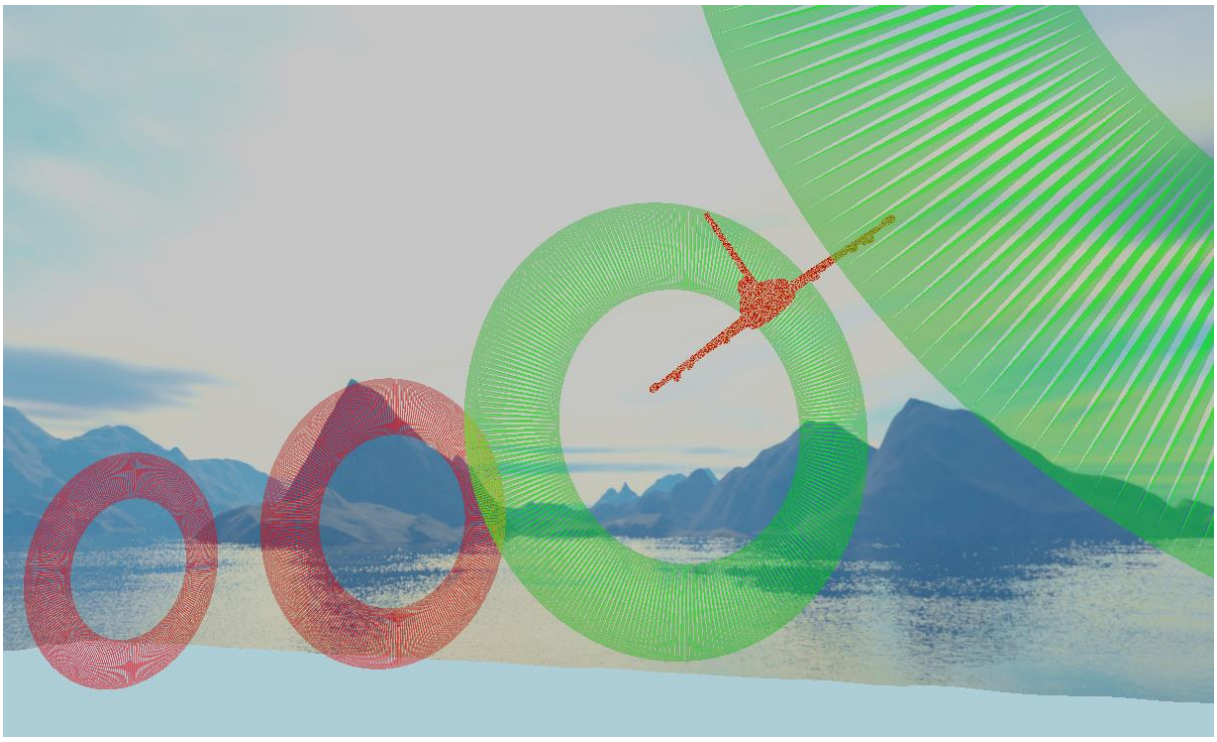
玩家在一个小岛上出生，且小岛四周存在空气墙。在地面关卡，玩家需要记住五个灯柱的闪亮次序，并按次序激活灯柱。如果激活次序正确，灯柱会变为绿色，否则会变为红色：



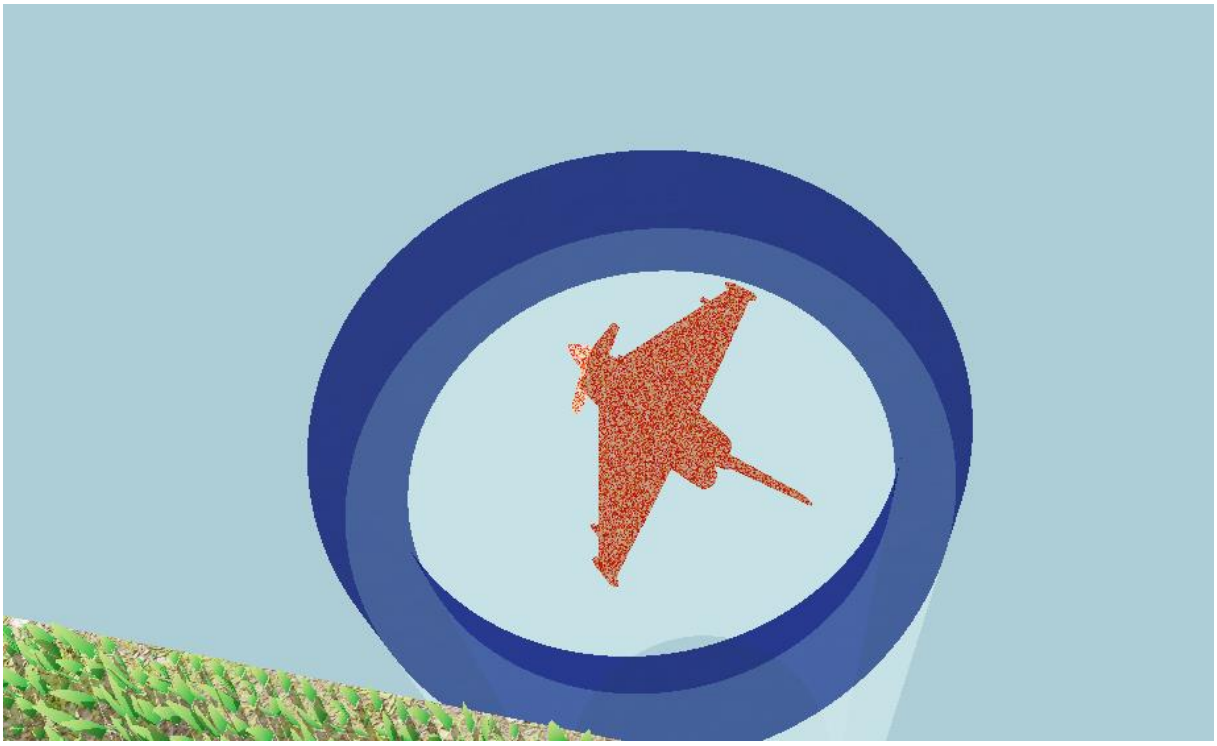
激活正确后，回到飞机旁，飞机会变为红色，且起飞 CG 会被触发：

接下来，在天空关卡中，玩家需要穿过所有的圆环。每个圆环在被穿过后会变为绿色。





所有圆环都被穿过后，会触发第二段 CG，且游戏会被重置：



5 心得

在这次大程序的开发中，我们学到了许多。从底层的渲染管线到顶层的游戏策划，每一个环节都需要大量的阅读以及思考。当然，使用 GLSL 变成给我们带来了比预期困难许多的挑战——CPU 中各种奇特 Buffer 的绑定、GLSL 语言的特殊执行流程以及 GPU 调试的困难性与复杂性。与之相伴的，我们也收获了非常多在课程中未能学到的知识。并且，我们综合运用了一学期中理论课程学到的变换、投影、光照、材质、网格等技术，并在工程实践中将它们有机地结合在了一起——而不是单纯地在 Glut 中使用一个函数，加深了对他们的理解。例如，所有的光照、纹理加载、几何变换等操作我们都是手动在 CPU 中写好绑定，在 GPU 中通过 GLSL 建构渲染管线的方式完成的。

然而，由于工期紧迫，我们还有许多没有实现的技术与设想，例如多光源技术以及更多的游戏关卡。希望如果有机会，我们能在未来选修更多的图形学相关课程，并将游戏完善。

望有所广益。