

Silent Forest

Silent Forest

项目要求及实现

 基本要求

 额外要求

游戏概览

 游戏画面

 游戏操作

实现重点

 Obj导入与贴图导入

 Obj导入

 贴图导入

 材质编辑

 相机类与第三人称视角

 地面纹理显示

 地势构建

 高度变化

 光源设计

 点光源

 聚光光源

 多光源

 实时阴影

 创建ShadowMap

 建立光源视角，生成ShadowMap

 利用阴影贴图计算遮挡关系，生成阴影

 碰撞检测

 与地面碰撞检测

 与树的碰撞检测

 雾效与SkyBox

 雾效的实现

 SkyBox的实现

运行方法与环境

 运行环境

 运行方法

小组成员

参考资料

 工程实现参考

 项目资源来源

项目要求及实现

基本要求

- 基于OpenGL/WebGL，具有基本体素（立方体、球、圆柱、圆锥、多面棱柱、多面棱台）的建模表达能力；
- 具有基本三维网格导入导出功能（建议OBJ格式）；
- 具有基本材质、纹理的显示和编辑能力；
- 具有基本几何变换功能（旋转、平移、缩放等）；
- 基本光照明模型要求，并实现基本的光源编辑（如调整光源的位置，光强等参数）；

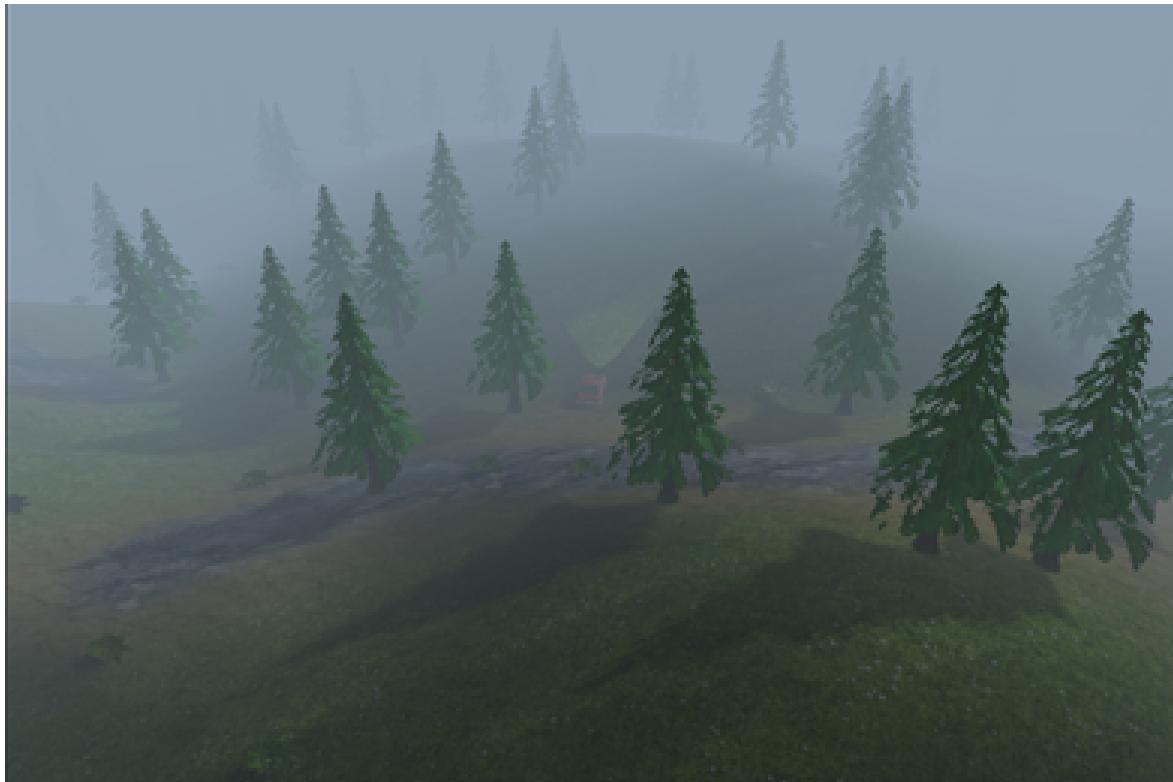
能对建模后场景进行漫游如Zoom In/Out, Pan, Orbit, Zoom To Fit等观察功能。

额外要求

- 漫游时可实时碰撞检测
- 光照明模型细化，实现实时阴影

游戏概览

游戏画面



游戏操作

- WS控制前后
- AD控制车头转动
- 鼠标左键控制视角
- 滚轮实现缩放
- 空格控制开关灯

实现重点

Obj导入与贴图导入

Obj导入

分别导入v、vt、vn、f等相关数据，存入对应的数组中，经过处理之后送入函数生成模型。

```
ModelData loadObj(const char* fileName) {
    ifstream file(fileName);
    string line;
    std::vector<std::vector<std::string>> f_index;
    while (getline(file, line)) {
        if (line.substr(0, 2) == "v ") {
            glm::vec3 vertex;
            istringstream s(line.substr(2));
            s >> vertex.x;
            s >> vertex.y;
            s >> vertex.z;
            Vertex v = Vertex(vertices.size(), vertex);
            vertices.push_back(v);
        }
        else if (line.substr(0, 3) == "vt ") {
            glm::vec2 texture;
            istringstream s(line.substr(2));
            s >> texture.x;
            s >> texture.y;
            textures.push_back(texture);
        }
        else if (line.substr(0, 3) == "vn ") {
            glm::vec3 normal;
            istringstream s(line.substr(2));
            s >> normal.x;
            s >> normal.y;
            s >> normal.z;
            normals.push_back(normal);
        }
        else if (line.substr(0, 2) == "f ") {
            vector<string> words = parser(line, " ");
            vector<string> v1 = parser(words[1], "/");
            vector<string> v2 = parser(words[2], "/");
            vector<string> v3 = parser(words[3], "/");
            f_index.push_back(v1);
            f_index.push_back(v2);
            f_index.push_back(v3);
        }
    }
}
```

```

    for(int i = 0; i < f_index.size(); i++) {
        processVertex(f_index[i]);
    }
    removeVertices();
    float *verticesArray = new float[vertices.size()*3];
    float *texturesArray = new float[vertices.size()*2];
    float *normalsArray = new float[vertices.size()*3];
    float furthest = convertDataToArrays(verticesArray, texturesArray,
normalsArray);
    int* indicesArray = convertIndicesToArray();
    ModelData data = ModelData(verticesArray, texturesArray, normalsArray,
indicesArray, furthest, vertices.size(), indices.size());
    return data;
}

```

贴图导入

使用stb_image库导入贴图。

```

unsigned int loadCubeMap(string faces[], int length)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < length; i++)
    {
        unsigned char *data = stbi_load(faces[i].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: " << faces[i]
<< 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);

    return textureID;
}

```

材质编辑

通过修改导入模型与贴图的各项参数编辑和显示物体的材质。

```
class Texture {
public:
    int textureID;
    float shineDamper = 1;
    // 反射率
    float reflectivity = 0;
    // 透明度选择开关
    int hasTransparency = 0;
    // 自发光
    int useFakeLighting = 0;
};
```



相机类与第三人称视角

相机始终对准小车，并跟随小车进行移动。使用鼠标可以调整视角，使用WASD控制小车和相机的移动，使用滚轮进行缩放。

```

void move(Terrain terrain){
    flushmouse();
    calculatePitch(terrain);
    calculateAngleAroundPlayer();
    float horizontalDistance = (float) (distanceFromPlayer *
cos(pitch*2*Pi/360.0));
    float verticalDistance = (float) (distanceFromPlayer *
sin(pitch*2*Pi/360.0));
    calculateCameraPosition(horizontalDistance, verticalDistance);
    yaw = 180 - (player->ry + angleAroundPlayer);
}

```

地面纹理显示

使用BlendMap进行多纹理贴图显示，很好的进行了地面的显示，其中主要思路是根据一张有四种颜色的图片进行贴图绑定，不同的颜色区域绑定不同的纹理贴图，并且通过颜色的交织来实现多种纹理的混杂。具体实现的着色器代码如下：

```

vec4 blendMapColor = texture(blendMap, pTexCoord);
float backTextureAmount = 1 - (blendMapColor.r + blendMapColor.g +
blendMapColor.b);
vec2 tiledCoords = pTexCoord * 40.0;
vec4 backgroundTextureColor = texture(backgroundTexture, tiledCoords) *
backTextureAmount;
vec4 rTextureColor = texture(rTexture, tiledCoords) * blendMapColor.r;
vec4 gTextureColor = texture(gTexture, tiledCoords) * blendMapColor.g;
vec4 bTextureColor = texture(bTexture, tiledCoords) * blendMapColor.b;

vec4 totalColor = backgroundTextureColor + rTextureColor + gTextureColor +
bTextureColor;

```

可以看到，最终的纹理颜色将会是背景与三张贴图的混合。



地势构建

高度变化

地形的基本贴图是一张平面，为了实现地形的起伏变化，我们采用了HeightMap设置地形高度。HeightMap是一张利用噪声生成的贴图，我们将其导入后根据纹理坐标读取对应位置的高度值，设置地形高度，从而将二维的地形拓展为三维地势。

```
// Terrain.h
unsigned char* data = stbi_load(heightMap.c_str(), &width, &height, &nChannels,
0);
...
for (int i = 0; i < VERTEX_COUNT; i++) {
    for (int j = 0; j < VERTEX_COUNT; j++) {
        float tHeight = getHeight(j, i, height, width, data);
        vertices[vertexPointer * 3 + 1] = tHeight;
    }
}
```



光源设计

除了最基本的Phong光照模型之外，我们还实现了点光源、聚光光源与多光源，并且光源的强度、位置和照射方向均可改变。

点光源

在平行光源的基础上，设置光源位置与衰减系数。光照方向通过点坐标减光源坐标计算得出，光强随着距离衰减。

```
float distance = length(toLightVector[i])/5;
float attFactor = attenuation[i].x+(attenuation[i].y*distance)+  
(attenuation[i].z*distance*distance);
```

聚光光源

在点光源的基础上，加入打光方向和光照的界限。若光源指向物体的向量与光的方向的夹角超过限制，则不会显示这道光。

```
float theta = dot(lightDir, normalize(-lightDirection[i]));
if(theta < lightCutoff[i]){
    totalDiff += (bright * lightColor[i])/attFactor;
    totalSpec += (dampedFactor * reflectivity * lightColor[i])/attFactor;
}
```

多光源

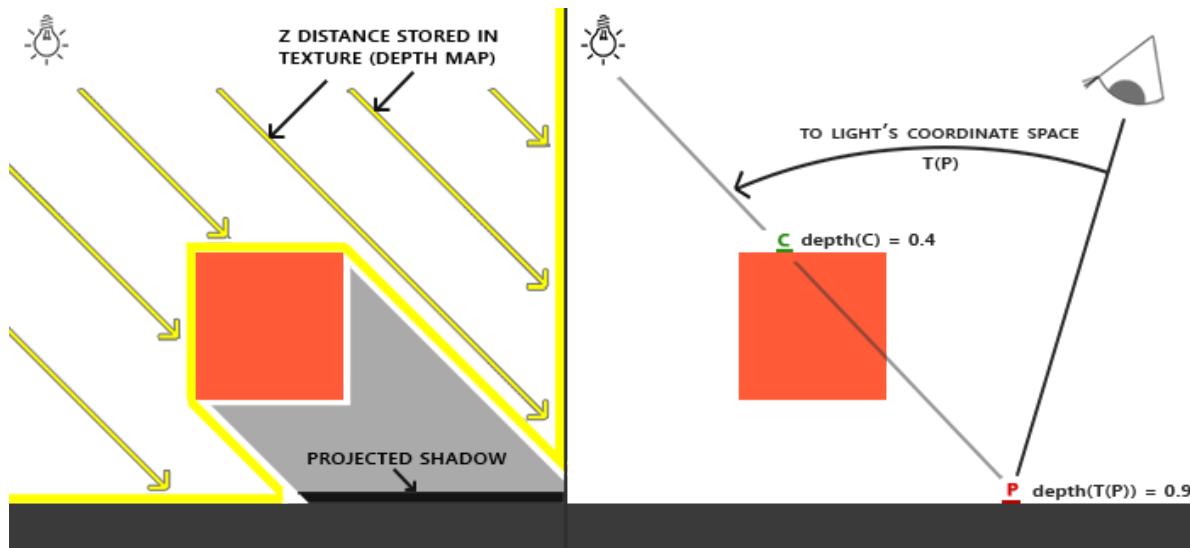
多光源的实现基于着色器的修改，允许传入 `MAX_SIZE` 的光源数，并在着色器中对所有光源逐一进行计算，得到最终的光照效果。光源的添加位于 `CGProj.cpp` 中，默认位置为(0,0,0)，默认白光。

```
// Shader.fs
uniform vec3 lightPos[4];
...
for(int i=0; i<4; i++){
    toLightVector[i] = lightPos[i] - worldPos.xyz;
}
...
```



实时阴影

实时阴影的整体实现流程大致分为如下几步：创建帧缓冲，创建ShadowMap -> 建立光源视角，利用着色器生成阴影贴图 -> 利用阴影贴图计算遮挡关系，生成阴影。



创建ShadowMap

我们在此处参考了[2]中的帧缓冲和shadowmap创建方式（虽然其他教程看了一下也都差不多），与原有帧缓冲和材质不同的是，ShadowMap禁止了Draw和Read功能，并将材质属性设置为深度测试。

```
glGenFramebuffers(1, &depthMapFBO);
// create depth texture
glGenTextures(1, &depthMap);
 glBindTexture(GL_TEXTURE_2D, depthMap);
 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);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

// attach depth texture as FBO's depth buffer
 glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
 depthMap, 0);
 glDrawBuffer(GL_NONE);
 glReadBuffer(GL_NONE);
```

建立光源视角，生成ShadowMap

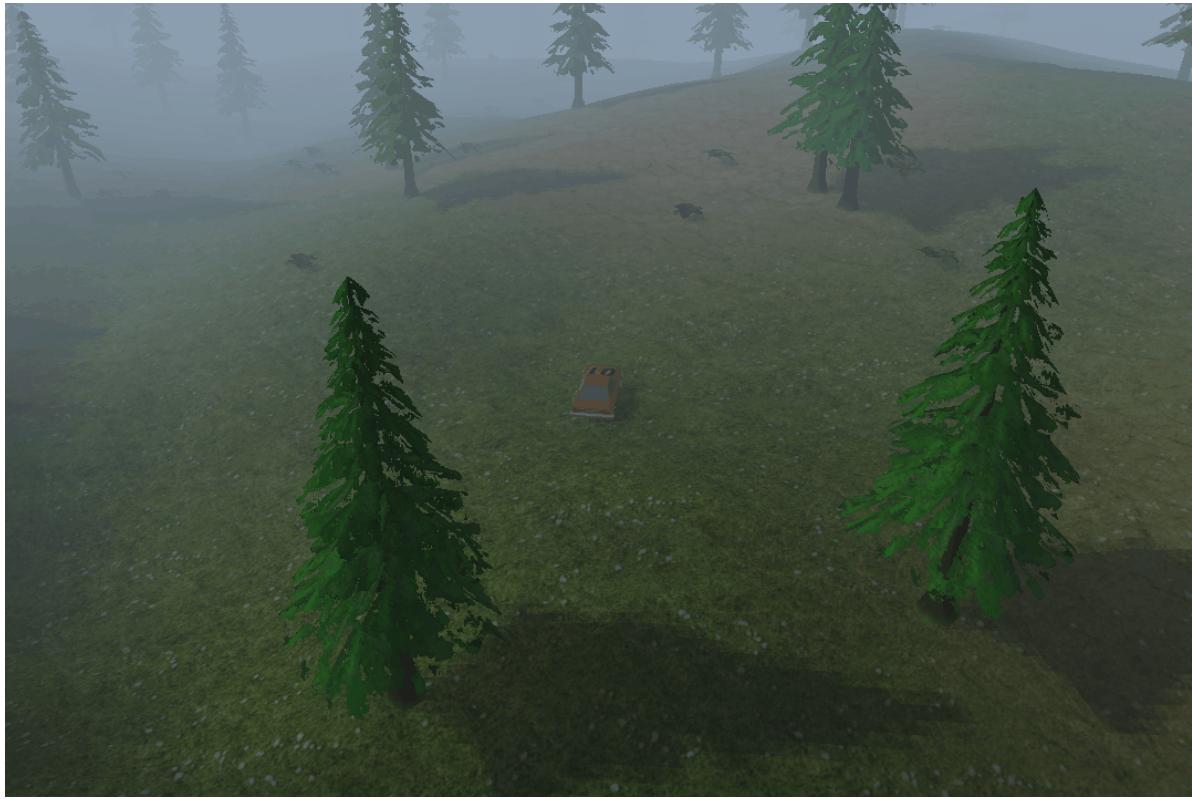
由于找到的教程中框架与我们的工程框架不相符，后面的相关代码都是我们自行构建的。主要涉及 `ShadowEntityRender.h` 和 `ShadowShader.h` 两个类，前者负责调用 `ShadowShader` 实现物体在 `ShadowMap` 上的渲染，后者负责通过 `depth.fs` 和 `depth.vs` 两个 `glsl` 文件生成专用的阴影着色器。`ShadowEntityRender` 的渲染操作在 `MasterRender.h` 的 `ShadowRender()` 函数中执行。

利用阴影贴图计算遮挡关系，生成阴影

利用ShadowMap生成最后阴影的过程主要在Terrain相关类中实现。

- 向TerrainShader中导入ShadowMap
- 利用ShadowMap计算遮挡关系，实现阴影效果，同时利用PCF进行阴影优化

```
// TShader.fs
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // perform perspective divide
    vec3 shadowCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // transform to [0,1] range
    shadowCoords = shadowCoords * 0.5 + 0.5;
    // get closest depth value from light's perspective
    float closestDepth = texture(shadowMap, shadowCoords.xy).r;
    // get depth of current fragment from light's perspective
    float currentDepth = shadowCoords.z;
    // check whether current frag pos is in shadow
    float bias = 0.001;
    //float shadow = currentDepth - bias > closestDepth ? 0.7 : 1.0;
    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, shadowCoords.xy + vec2(x, y) *
texelSize).r;
            shadow += currentDepth - bias > pcfDepth ? 0.6 : 0.0;
        }
    }
    shadow /= 9.0;
    // keep the shadow at 0.0 when outside the far_plane region of the light's
frustum.
    if(shadowCoords.z > 1.0)
        shadow = 0.0;
    return shadow;
}
```



碰撞检测

与地面碰撞检测

与地面的碰撞检测主要实现两点：一是保持小车高度与地面高度一致，二是小车的底面与地面平齐。

第一点的实现较为简单，只需要让小车的y坐标与地面高度始终相等即可。

对于第二点，我们利用附近地面的高度，可以大致计算出小车横向与纵向上的地面坡度，并根据坡度设置小车绕x轴和z轴旋转的角度，就可以大致保持小车与地面平齐。

```
Xslope =(terrainHeight-terrain.getHeightOfTerrain(position.x-sin(radians(this->ry))*10,
    position.z- cos(radians(this->ry))*10))/(sqrt(sin(radians(this->ry))*sin(radians(this->ry))*100+cos(radians(this->ry))*cos(radians(this->ry))*100));
Zslope =(terrainHeight-terrain.getHeightOfTerrain(position.x-sin(radians(this->ry+270)))*5,
    position.z-cos(radians(this->ry+270))*5))/(sqrt(sin(radians(this->ry+270))*sin(radians(this->ry+270))*25+cos(radians(this->ry+270))*cos(radians(this->ry+270))*25));

float Xangle = atan(Xslope)*360/(2*Pi);
float Zangle = atan(Zslope)*360/(2*Pi);

rx = -Xangle*cos(ry*2*Pi/360.0);
rz = -Zangle*fabs(sin(ry*2*Pi/360.0));
```

与树的碰撞检测

实现与树的碰撞检测时，我们尝试了质点检测与AABB检测两种方法。质点检测即为直接判断树的中心与小车中心的距离，若小于一个阈值，则判断两者相撞。AABB检测则是将两者都视为一个长方体的盒子，计算两个盒子是否相交。

质点：

```
bool checkCollision(float x1, float z1, float x2, float z2){  
    return sqrt((x1-x2)*(x1-x2)+(z1-z2)*(z1-z2))<collideSize;  
}
```

AABB：

```
bool checkCollision(float x1, float z1, float x2, float z2){  
    bool collisionX = x1 + xSize1 >= x2 - 1 && x2 + 1 >= x1 - xSize2;  
    bool collisionZ = z1 + zSize >= z2 - 1 && z2 + 1 >= z1 - zSize;  
    return collisionX && collisionZ;  
}
```

由于我们的小车的横轴与纵轴并不一直与x轴和z轴保持一致，AABB碰撞检测的最终效果反而不如质点检测，因此我们最终使用质点检测的方法进行碰撞检测。



雾效与SkyBox

雾效的实现

雾的效果由物体本身颜色与背景色混合实现，使用density代表雾的浓度，gradient代表雾随距离增长的快慢。以density、gradient、distance三者为参数构建函数，计算出对应的能见度visibility。再以visibility为参数将背景色与物体本身颜色混合。

```

visibility = exp(-pow((distance*density),gradient));
visibility = clamp(visibility, 0.0, 1.0);

FragColor = mix(vec4(skyColor,1.0), FragColor, visibility);

```

SkyBox的实现

SkyBox我们使用简单的立方体贴图，将整个场景覆盖起来，并且始终保证摄像机处在SkyBox的中间，读取SkyBox的六张贴图就利用在贴图导入部分所描述的loadTexture函数，这里主要展示它的着色器代码

```

#version 400 core

in vec3 textureCoord;
out vec4 FragColor;

uniform samplerCube cubeMap;
uniform vec3 fogColor;

const float lowerLimit = 0.0;
const float upperLimit = 60.0;

void main(void) {
    vec4 FinalColor = texture(cubeMap, textureCoord);

    float factor = (textureCoord.y - lowerLimit) / (upperLimit - lowerLimit);
    factor = clamp(factor, 0.0, 1.0);
    FragColor = mix(vec4(fogColor, 1.0), FinalColor, factor);
}

```

```

-#version 400 core

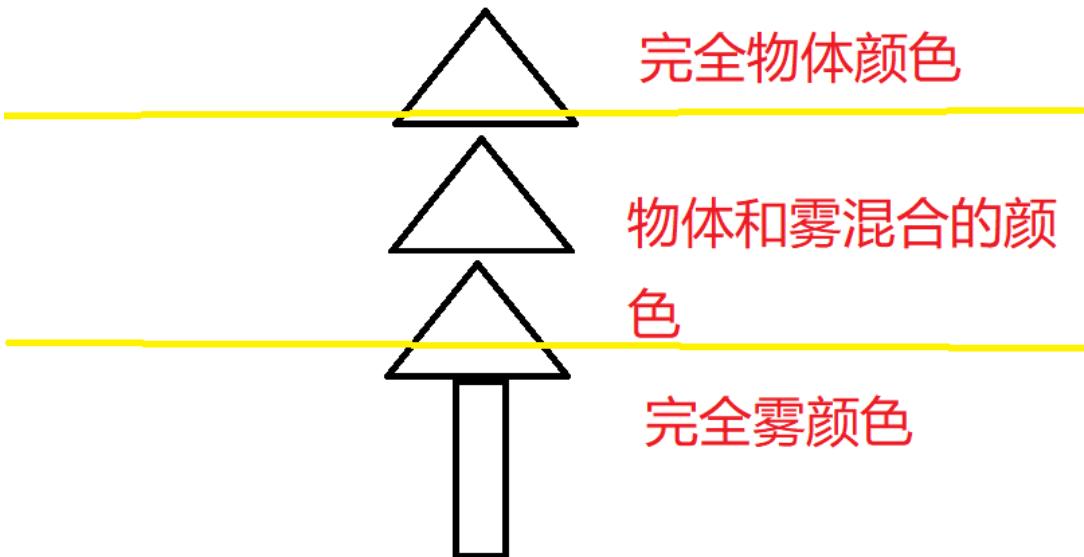
layout (location = 0) in vec3 position;
out vec3 textureCoord;

uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;

void main()
{
    textureCoord = position;
    vec4 pos = projectionMatrix * viewMatrix * vec4(position, 1.0);
    gl_Position = pos.xyww;
}

```

在这里我组与普通天空盒稍有区别，为了使得天空盒与雾效比较配合，我组在绘制颜色时将天空颜色与雾的颜色同时考虑：越是靠近地面的物体，雾越浓（也就是越能表现出雾的颜色），往高处，才会慢慢凸物体，最上层（超过某个高度）则是物体本身的颜色或者天空盒子的颜色。原理如下图所示：(图片来源：<https://codeantenna.com/a/8JOrZV9ExO>)



最终我们可以得到一个比较好的效果



运行方法与环境

运行环境

- Unbuntu 20.04
- GLFW and GLM and GLAD

运行方法

在src目录下执行./run.sh, 其中的内容如下:

```
g++ glad.c stb_image.cpp CGProj.cpp -o CGProj -lglfw3 -lGL -lX11 -lpthread -lXrandr -lXi -ldl
./CGProj
```

小组成员

- 曾帅 3190105729
- 王异鸣 3190102780
- 杨淇森 3190105500

参考资料

工程实现参考

1. glfw基本框架与opengl基础功能(shader/render)参考: <https://learnopengl-cn.github.io/01%20Getting%20started/04%20Hello%20Triangle/>
2. 实时阴影: [LearnOpenGL - Shadow Mapping](#)
3. 地面纹理与BlendMap: <https://www.chinedufn.com/webgl-multitexture-blend-map-tutorial/>
4. SkyBox雾效问题解决: <https://codeantenna.com/a/8jOrZV9ExO>
5. 地形绘制与基本雾: https://lwglibook-cn-translation.readthedocs.io/zh_CN/latest/

项目资源来源

- <http://free3d.com/>
- 谷歌图片