CG2021 - 设计文档

> - 2021-2022秋冬，计算机图形学课程项目
> - 简单三维建模及真实感绘制 —— 飞行模拟器
> - 小组成员
>   - 董乙灿，3190105140
>   - 高晨熙：3190103302
>   - 夏霄汉：3190102367

CG2021 - 设计文档

1 基本要求

1.1 基本体素建模

1.1.1 `initBuffers()`绑定buffer

1.1.2 立方体

1.1.3 球

1.1.4 圆柱

1.1.5 圆锥

1.2 三维网格导入导出&材质、纹理显示和编辑

1.2.1 obj文件处理

1.2.1 mtl文件处理

1.3 基本几何变换功能

1.3.1 设置模型矩阵

1.3.2 旋转变换

1.3.3 平移变换

1.3.4 缩放变换

1.4 基本光照明模型

1.4.1 光源

1.4.2 反射类型

漫反射

环境反射

镜面反射

1.5 场景漫游

1.6 Awesomeness指数

1.6.1 体积云实现

云的形状

梯度噪声perlin noise

细胞噪声worly noise

分形布朗运动

三维噪声纹理生成

云的绘制

光线步进法

体积云光照效果

1.6.2 粒子效果

起始位置、结束位置的确定

粒子消散时间的确定

粒子飘动效果

绘制圆形的点

1.6.3 雾化实现

计算物体到相机的距离

获取原色

获取雾色

基于密度的雾

2 额外要求
2.1 漫游时实时碰撞检测
2.2 光照明模型细化 —— 实时阴影
2.2.1 纹理贴图
2.2.2 渲染到纹理
2.2.3 制作阴影纹理
选择光源位置
将深度写入阴影纹理
2.2.4 将阴影纹理贴图到地板上

1 基本要求

1.1 基本体素建模

基本体素的绘制流程为

```
#传入center、size、color等体素参数，得到体素buffer
#其中调用了initBuffers()，绑定buffer
xxxbuffer=initOnexxx(...)
#传入模型矩阵的平移、旋转、缩放等参数，得到体素的modelMatrix
xxx_modelMatrix=setModelMatrix(translation, rotation, scale, modelxrotation, modelyrotation, modelzrotation, center)
#得到体素的viewMatrix和ProjectionMatrix
viewMatrix=setViewMatrix()
projectionMatrix=setProjectionMatrix()
#绘制基本体素
draw(xxxbuffer,xxx_modelMatrix,viewMatrix,ProjectionMatrix)
```

1.1.1 `initBuffers()`绑定buffer

```
function initBuffers(gl, positions, colors, indices, normals) {
    const positionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);

    //为颜色创建缓冲区
    const colorBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);

    // 3.索引缓冲区
    const indexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
    // Now send the element array to GL
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);

    const normalBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(normals), gl.STATIC_DRAW);

```

```
    return {
        position: positionBuffer,
        color: colorBuffer,
        index: indexBuffer,
        normal: normalBuffer,
        indices:indices,
    }
}
```

1.1.2 立方体

立方体由六个面构成，每个面使用两个三角形拼接而成，因此，对于每个面的两个三角形输入三个点的坐标位置，并对这些点分别指定他们的颜色和法向量，再使用indices指定绘制顺序，即可绘制一个立方体。

其中center表示立方体体心的坐标，size表示立方体在x、y、z轴方向上的尺寸，颜色表示立方体整体的颜色

```
  function initOneCube(Program, center, size, color) {
    const positions = [];
    const colors = [];
    const indices = [
      0, 1, 2, 0, 2, 3,
      4, 5, 6, 4, 6, 7,
      8, 9, 10, 8, 10, 11,
      12, 13, 14, 12, 14, 15,
      16, 17, 18, 16, 18, 19,
      20, 21, 22, 20, 22, 23
    ];
    for(var i =0;i<24;i++){
      colors.push(color[0], color[1], color[2], color[3]);
    }
    {
      positions.push(center[0] + size[0] / 2,center[1] + size[1] /
2,center[2] + size[2] / 2);
      positions.push(center[0] - size[0] / 2,center[1] + size[1] /
2,center[2] + size[2] / 2);
      positions.push(center[0] - size[0] / 2,center[1] - size[1] /
2,center[2] + size[2] / 2);
      positions.push(center[0] + size[0] / 2,center[1] - size[1] /
2,center[2] + size[2] / 2);
      positions.push(center[0] + size[0] / 2,center[1] + size[1] /
2,center[2] + size[2] / 2);
      positions.push(center[0] + size[0] / 2,center[1] - size[1] /
2,center[2] + size[2] / 2);
      positions.push(center[0] + size[0] / 2,center[1] - size[1] /
2,center[2] - size[2] / 2);
      positions.push(center[0] + size[0] / 2,center[1] + size[1] /
2,center[2] - size[2] / 2);
      positions.push(center[0] + size[0] / 2,center[1] + size[1] /
2,center[2] + size[2] / 2);
      positions.push(center[0] + size[0] / 2,center[1] + size[1] /
2,center[2] - size[2] / 2);
      positions.push(center[0] - size[0] / 2,center[1] + size[1] /
2,center[2] - size[2] / 2);
      positions.push(center[0] - size[0] / 2,center[1] + size[1] /
2,center[2] + size[2] / 2);
```

```
      positions.push(center[0] - size[0] / 2,center[1] + size[1] /
2,center[2] + size[2] / 2);
      positions.push(center[0] - size[0] / 2,center[1] - size[1] /
2,center[2] + size[2] / 2);
      positions.push(center[0] - size[0] / 2,center[1] - size[1] /
2,center[2] - size[2] / 2);
      positions.push(center[0] - size[0] / 2,center[1] + size[1] /
2,center[2] - size[2] / 2);
      positions.push(center[0] - size[0] / 2,center[1] - size[1] /
2,center[2] + size[2] / 2);
      positions.push(center[0] + size[0] / 2,center[1] - size[1] /
2,center[2] + size[2] / 2);
      positions.push(center[0] + size[0] / 2,center[1] - size[1] /
2,center[2] - size[2] / 2);
      positions.push(center[0] - size[0] / 2,center[1] - size[1] /
2,center[2] - size[2] / 2);
      positions.push(center[0] + size[0] / 2,center[1] - size[1] /
2,center[2] - size[2] / 2);
      positions.push(center[0] + size[0] / 2,center[1] + size[1] /
2,center[2] - size[2] / 2);
      positions.push(center[0] - size[0] / 2,center[1] + size[1] /
2,center[2] - size[2] / 2);
      positions.push(center[0] - size[0] / 2,center[1] - size[1] /
2,center[2] - size[2] / 2);
    }
    const normals = new Float32Array([
      0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
      1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
      0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
      -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, -1.0, 0.0, 0.0,
      0.0, -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, -1.0, 0.0,
      0.0, 0.0, -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, -1.0
    ]);
    const buffers = initBuffers(Program.gl, positions, colors, indices,
normals);
    return buffers;
  }
```

1.1.3 球

球体的绘制可以想象成地球仪上的经纬线，经线和纬线会组成一个方形，将这个方形分成两个三角形。只需要以每条经线和纬线的交点为position绘制这些三角形即可。由于每个片元面积较小，因此可以直接将从球心到每个点的连线当作那个点的法向量。

```
  function initOneBall(Program, center, radius, color) {
    var positions = new Array();
    for (i = 0; i <= 180; i += 1) {//fai
      for (j = 0; j <= 360; j += 1) {//theata
        positions.push(radius * Math.sin(Math.PI * i / 180) *
Math.cos(Math.PI * j / 180) + center[0]);
        positions.push(radius * Math.sin(Math.PI * i / 180) *
Math.sin(Math.PI * j / 180) + center[1]);
        positions.push(radius * Math.cos(Math.PI * i / 180) + center[2]);
      }
    }
```

```
    var colors = new Array();
    for (i = 0; i <= 180; i += 1) {
      for (j = 0; j <= 360; j += 1) {
        colors.push(color[0]);  //R
        colors.push(color[1]);  //G
        colors.push(color[2]);  //B
        colors.push(color[3]);  //Alpha
      }
    }
    var indices = new Array();
    for (i = 0; i < 180; i += 1) {//fai
      for (j = 0; j < 360; j += 1) {//theata
        indices.push(360 * i + j);
        indices.push(360 * i + (j + 1));
        indices.push(360 * (i + 1) + j);
        indices.push(360 * (i + 1) + j + 1);
        indices.push(360 * i + (j + 1));
        indices.push(360 * (i + 1) + j);
      }
    }
    var normals = new Array();
    for (i = 0; i <= 180; i += 1) {//fai
      for (j = 0; j <= 360; j += 1) {//theata
        normals.push(radius * Math.sin(Math.PI * i / 180) * Math.cos(Math.PI
* j / 180));
        normals.push(radius * Math.sin(Math.PI * i / 180) * Math.sin(Math.PI
* j / 180));
        normals.push(radius * Math.cos(Math.PI * i / 180));
      }
    }
    const buffers = initBuffers(Program.gl, positions, colors, indices,
normals);
    return buffers;
  }
```

1.1.4 圆柱

圆柱由上下的两个底面圆以及侧面组成。圆形的绘制可以使用圆周上的两个点到圆心的三角形拼接起来组成，这些点的法向量为垂直于该圆形平面的向量。侧面的绘制使用上下圆面的圆周上的点绘制三角形并拼接组成，侧面上的点的法向量可以使用上下底面圆心的中点到两个点的直线表示。

```
function initOneCone(Program, center, radius, height, color) {
    const positions = [];
    const colors = [];
    const indices = [];
    const normals = [];
    for (i = 0; i < 360; i += 1) {//圆上的点
        positions.push(center[0] + radius * Math.cos(2 * Math.PI * i /
360.0));
        positions.push(center[1] + radius * Math.sin(2 * Math.PI * i /
360.0));
        positions.push(center[2]);
        colors.push(color[0], color[1], color[2], color[3]);
        normals.push(0.0, 0.0, -1.0);
    }
    positions.push(center[0], center[1], center[2]);//圆心，360
    colors.push(color[0], color[1], color[2], color[3]);
```

```
    normals.push(0.0, 0.0, -1.0);
    for (i = 0; i < 359; i += 1) {//圆面
        indices.push(i, i + 1, 360);
    }
    indices.push(360, 359, 0);
    for (i = 0; i < 360; i = i + 1) {
        //360次顶点，361~720
        positions.push(center[0], center[1], center[2] + height);//顶点
        colors.push(0.0, 0.0, 0.0, 1.0);//顶点弄成黑的
        //360个顶点法向量
        var vec1 = [radius * Math.cos(2 * Math.PI * i / 360.0), radius * Math.sin(2 * Math.PI * i / 360.0), -height];
        var vec2 = [radius * Math.cos(2 * Math.PI * (i + 1) / 360.0), radius * Math.sin(2 * Math.PI * (i + 1) / 360.0), -height];
        var vec3 = [vec1[1] * vec2[2] - vec1[2] * vec2[1], vec1[2] * vec2[0] - vec1[0] * vec2[2], vec1[0] * vec2[1] - vec1[1] * vec2[0]];
        normals.push(vec3[0], vec3[1], vec3[2]);
    }
    for (i = 0; i < 360; i = i + 1) {
        //圆上的点，721~1080
        positions.push(center[0] + radius * Math.cos(2 * Math.PI * i / 360.0));
        positions.push(center[1] + radius * Math.sin(2 * Math.PI * i / 360.0));
        positions.push(center[2]);
        colors.push(color[0], color[1], color[2], color[3]);
        //360个圆面上点的法向量
        var vec1 = [radius * Math.cos(2 * Math.PI * i / 360.0), radius * Math.sin(2 * Math.PI * i / 360.0), -height];
        var vec2 = [radius * Math.cos(2 * Math.PI * (i + 1) / 360.0), radius * Math.sin(2 * Math.PI * (i + 1) / 360.0), -height];
        var vec3 = [vec1[1] * vec2[2] - vec1[2] * vec2[1], vec1[2] * vec2[0] - vec1[0] * vec2[2], vec1[0] * vec2[1] - vec1[1] * vec2[0]];
        normals.push(vec3[0], vec3[1], vec3[2]);
    }
    for (i = 361; i < 720; i += 1) {
        indices.push(i, i + 360, i + 361);
    }
    indices.push(720, 1080, 721);
    const buffers = initBuffers(Program.gl, positions, colors, indices, normals);
    return buffers;
}
```

1.1.5 圆锥

圆锥由底面圆与侧面组成。圆形的绘制可以使用圆周上的两个点到圆心的三角形拼接起来组成，这些点的法向量为垂直于该圆形平面的向量。侧面的绘制使用圆周上的每两个点到圆锥顶点的三角形拼接组成，侧面上点的法向量为绘制该三角形使用的两条片的向量叉乘。

```
function initOneCone(Program, center, radius, height, color) {
    const positions = [];
    const colors = [];
    const indices = [];
    const normals = [];
    for (i = 0; i < 360; i += 1) {//圆上的点
```

```
        positions.push(center[0] + radius * Math.cos(2 * Math.PI * i / 360.0));
        positions.push(center[1] + radius * Math.sin(2 * Math.PI * i / 360.0));
        positions.push(center[2]);
        colors.push(color[0], color[1], color[2], color[3]);
        normals.push(0.0, 0.0, -1.0);
    }
    positions.push(center[0], center[1], center[2]);//圆心，360
    colors.push(color[0], color[1], color[2], color[3]);
    normals.push(0.0, 0.0, -1.0);
    for (i = 0; i < 359; i += 1) {//圆面
        indices.push(i, i + 1, 360);
    }
    indices.push(360, 359, 0);
    for (i = 0; i < 360; i = i + 1) {
        //360次顶点，361~720
        positions.push(center[0], center[1], center[2] + height);//顶点
        colors.push(0.0, 0.0, 0.0, 1.0);//顶点弄成黑的
        //360个顶点法向量
        var vec1 = [radius * Math.cos(2 * Math.PI * i / 360.0), radius * Math.sin(2 * Math.PI * i / 360.0), -height];
        var vec2 = [radius * Math.cos(2 * Math.PI * (i + 1) / 360.0), radius * Math.sin(2 * Math.PI * (i + 1) / 360.0), -height];
        var vec3 = [vec1[1] * vec2[2] - vec1[2] * vec2[1], vec1[2] * vec2[0] - vec1[0] * vec2[2], vec1[0] * vec2[1] - vec1[1] * vec2[0]];
        normals.push(vec3[0], vec3[1], vec3[2]);
    }
    for (i = 0; i < 360; i = i + 1) {
        //圆上的点，721~1080
        positions.push(center[0] + radius * Math.cos(2 * Math.PI * i / 360.0));
        positions.push(center[1] + radius * Math.sin(2 * Math.PI * i / 360.0));
        positions.push(center[2]);
        colors.push(color[0], color[1], color[2], color[3]);
        //360个圆面上点的法向量
        var vec1 = [radius * Math.cos(2 * Math.PI * i / 360.0), radius * Math.sin(2 * Math.PI * i / 360.0), -height];
        var vec2 = [radius * Math.cos(2 * Math.PI * (i + 1) / 360.0), radius * Math.sin(2 * Math.PI * (i + 1) / 360.0), -height];
        var vec3 = [vec1[1] * vec2[2] - vec1[2] * vec2[1], vec1[2] * vec2[0] - vec1[0] * vec2[2], vec1[0] * vec2[1] - vec1[1] * vec2[0]];
        normals.push(vec3[0], vec3[1], vec3[2]);
    }
    for (i = 361; i < 720; i += 1) {
        indices.push(i, i + 360, i + 361);
    }
    indices.push(720, 1080, 721);
    const buffers = initBuffers(Program.gl, positions, colors, indices, normals);
    return buffers;
}
```

1.2 三维网格导入导出&材质、纹理显示和编辑

模型由obj格式文件提供，加载模型即对obj进行解析。

1.2.1 obj文件处理

对于o或g, 新建object进行存储；对于v，vt， vn，直接读取点坐标信息并存储；对于f，需要进行进一步处理：

首先将坐标索引、纹理坐标索引、法向量索引vi, ti, ni信息进行解析与存储，之后将非三个点的面划分成一个一个三角形：

```
            if(face.vIndices.length > 3){
                var n = face.vIndices.length - 2;
                var newVIndices = new Array(n * 3);
                var newTIndices = new Array(n * 3);
                var newNIndices = new Array(n * 3);
                for(var i=0; i < n; i++){
                  newVIndices[i * 3 + 0] = face.vIndices[0];
                  newVIndices[i * 3 + 1] = face.vIndices[i + 1];
                  newVIndices[i * 3 + 2] = face.vIndices[i + 2];
                  newTIndices[i * 3 + 0] = face.tIndices[0];
                  newTIndices[i * 3 + 1] = face.tIndices[i + 1];
                  newTIndices[i * 3 + 2] = face.tIndices[i + 2];
                  newNIndices[i * 3 + 0] = face.nIndices[0];
                  newNIndices[i * 3 + 1] = face.nIndices[i + 1];
                  newNIndices[i * 3 + 2] = face.nIndices[i + 2];
                }
                face.vIndices = newVIndices;
                face.tIndices = newTIndices;
                face.nIndices = newNIndices;
            }
```

此处，由于webGL在web端的文件加载是异步进行的，所以在初始化相关Buffer时需要保证纹理图片已加载完毕。在这里，我们将纹理的加载和mtl文件的加载从中分离出来，单独进行，以确保加载正确。

1.2.1 mtl文件处理

mtl文件的处理相对简单，只需要将各项材质信息读取出来即可，在外部传出进行处理。

```
    switch(command){
        //comment
        case '#':
            continue;
        //mtl
        case 'newmtl':
          var materialname = sp.getWord();
          var material  = new Material(materialname);
          mtl.mtls.push(material);
          currentmaterial = material;

            // currentmaterial.name = sp.getWord();
            continue;
        //反射指数
        case 'Ns':
            currentmaterial.Ns = sp.getFloat();
            continue;
```

```
        //环境反射
        case 'Ka':
            currentmaterial.Ka = sp.getRGB();
            continue;
        //漫反射
        case 'Kd':
            currentmaterial.Kd = sp.getRGB();
            continue;
        //镜面反射
        case 'Ks':
            currentmaterial.Ks = sp.getRGB();
            continue;
        //放射光
        case 'Ke':
            currentmaterial.Ke = sp.getRGB();
            continue;
        //折射值描述
        case 'Ni':
            currentmaterial.Ni = sp.getFloat();
            continue;
        //照明度
        case 'illum':
            currentmaterial.illum = sp.getInt();
            continue;
        // 渐隐指数
        case 'd':
            currentmaterial.d = sp.getFloat();
            continue;
        //为漫反射指定文件
        case 'map_Kd':
            // sp.getWord();sp.getWord();
            // var texname = sp.getWord();
            // mtl.texture = loadTexture(gl, path + texname);
            continue;
        default:
            continue;
```

1.3 基本几何变换功能

- 几何变换的基本原理已经多次接触，此处不再赘述
- 实现几何变换的主要对象为飞机和碰撞球，通过键盘交互修改几何变换参数的值，从而实现基本几何变换
- 注意几何变换需要符合基本物理规律，防止出现缩放穿模等现象

1.3.1 设置模型矩阵

- `setModelMatrix()`代码如下
  - 使用`mat4`矩阵类
  - 传入平移矩阵、旋转矩阵和缩放矩阵，以及模型的方向修正矩阵
  - 调用`mat4`的三种方法完成变换，返回最终的模型矩阵

```
// 设置模型矩阵，translation为模型的平移，rotation为模型的旋转，modelrotation，modelrotation用于模型的方向修正
```

```
    function setModelMatrix(translation, rotation, scale, modelxrotation, modelyrotation, modelzrotation, center) {
        const modelMatrix = mat4.create();
        mat4.translate(modelMatrix,     // destination matrix
            modelMatrix,                // matrix to translate
            translation);               // amount to translate
        mat4.rotate(modelMatrix,        // destination matrix
            modelMatrix,                // matrix to rotate
            rotation.rad,               // amount to rotate in radians
            rotation.axis);             // axis to rotate around (Z)
        if(center){
            var center0 = [];
            center0[0] = -center[0];
            center0[1] = -center[1];
            center0[2] = -center[2];
            mat4.translate(modelMatrix, // destination matrix
                modelMatrix,            // matrix to translate
                center);                // amount to translate
            mat4.scale(modelMatrix,     // destination matrix
                modelMatrix,            // matrix to rotate
                scale);                 // axis to rotate around (Z)
            mat4.translate(modelMatrix, // destination matrix
                modelMatrix,            // matrix to translate
                center0);               // amount to translate

        }
        if (modelxrotation)
            mat4.rotate(modelMatrix,    // destination matrix
                modelMatrix,            // matrix to rotate
                modelxrotation.rad,     // amount to rotate in radians
                modelxrotation.axis);   // axis to rotate around (Z)
        if (modelyrotation)
            mat4.rotate(modelMatrix,    // destination matrix
                modelMatrix,            // matrix to rotate
                modelyrotation.rad,     // amount to rotate in radians
                modelyrotation.axis);   // axis to rotate around (Z)
        if (modelzrotation)
            mat4.rotate(modelMatrix,    // destination matrix
                modelMatrix,            // matrix to rotate
                modelzrotation.rad,     // amount to rotate in radians
                modelzrotation.axis);   // axis to rotate around (Z)
        return modelMatrix;
    }
```

1.3.2 旋转变换

- 本次大程中使用到旋转变换的典例为飞机的爬升、俯冲和转弯
- 为了游戏控制性，我们给旋转变换设置了最大旋转角度，且用户松开按键后会自动转回初始状态
- 俯冲&爬升：飞机绕X轴转动，关键代码如下
    - `modelxrotation` 表示绕X轴旋转

```
function handleKeyDown(event) {
    if (String.fromCharCode(event.keyCode) == "S") {        // 俯冲
        planeIsRotating = true;
        if (modelxrotation.rad > minRotatingAngle)
                modelxrotation.rad -= del;
    }
    else if (String.fromCharCode(event.keyCode) == "W") {   // 爬升
        planeIsRotating = true;
        if (modelxrotation.rad < maxRotatingAngle)
            modelxrotation.rad += del;
    }
}
```

- 转弯：和上述实现类似，只是旋转轴改为Z轴，改变`modelzrotation`，细节不再贴出

1.3.3 平移变换

- 主要使用平移变换的是场景中碰撞球的分布和飞机的左右移动，此处以飞机左右移动为例

```
function handleKeyDown(event) {
    if (String.fromCharCode(event.keyCode) == "A") {        // 飞机向左的旋转效果
        translation[0] -= del * 1.2;
    }
    else if (String.fromCharCode(event.keyCode) == "D") {   // 飞机向右的旋转效果
        translation[0] += del * 1.2;
    }
}
```

- 直接修改飞机的平移矩阵`translation`的x分量即可实现左右平移

1.3.4 缩放变换

- 使用时间参数`now`调整对碰撞球的缩放变换

```
sinball_scale[0] = 1.5 + 0.3 * Math.sin(now);
sinball_scale[1] = 1.5 + 0.3 * Math.sin(now);
sinball_scale[2] = 1.5 + 0.3 * Math.sin(now);

var sinball_modelMatrix = setModelMatrix(nochange_translation, nochange_rotation, sinball_scale, null, null, null, ballCenter[i]);
drawTexture(Program, ballBuffer[i], sinball_modelMatrix, viewMatrix, projectionMatrix, ballSet[i].type, lightDirection);
```

- 使用上文提到的`setModelMatrix()`函数设置球体的模型矩阵，实现缩放变换

1.4 基本光照明模型

游戏中实现了基本光照模型，使用环境光+漫反射光+镜面反射光的基本模型。

1.4.1 光源

光源类型使用平行光。模拟太阳的光照效果。对于平行光源，需要给出光线方向：

```
    uniform vec3 uLightDirection;
```

此外使用环境光,模拟真实环境中的非直射光（由光源发出后经过墙壁或其他物体反射后的光）

本游戏中对于环境光的实现较为简单，各方向光强一致，只需定义颜色：

```
    uniform vec3 uAmbientLight; // 环境光
```

1.4.2 反射类型

漫反射

漫反射的反射光在各个方向上是均匀的，其反射光的强度收到入射角的影响。将入射角定义为θ，漫反射光的颜色可以根据下式计算得到：

$$diffuseColor = lightColor \times basicColor \times \theta$$

`lightColor`指的是平行光的颜色

环境反射

环境反射中，反射光的方向可以认为是入射光的反方向。环境反射光的颜色可以如下计算：

$$ambientLight = lightColor \times basicColor$$

`lightColor`指的是环境光的颜色

镜面反射

镜面反射采取`Microfacet-Based Models`,使用`Cook-Torrance`方法

$$f(i,o) = \frac{F(i,h)G(i,o,h)D(h)}{\pi(vn)(ln)}$$

$D(h)$:法线分布函数(Normal Distribution Function)，其代表了所有微观角度下微小镜面法线的分布情况，粗糙表面法线分布相对均匀，光滑表面法线分布相对集中

$$D = \frac{\exp\left((nh)^2 - 1\right)}{\pi(roughness)^4(nh)^6}$$

$F(i,h)$:菲涅尔方程，描述了物体表面在不同入射光角度下反射光线所占的比率

$$F = (1 - vh)^{fresnel}$$

$G(i,o,h)$:几何函数，描述了微平面自遮挡的属性。当一个平面相对比较粗糙的时候，平面表面上的微平面有可能挡住其他的微平面从而减少表面所反射的光线。

$$G = min(1, \frac{2(hn)(vn)}{vn}, \frac{2(hn)(ln)}{vn})$$

```
    float cookTorrance(vec3 viewDirection,
        vec3 lightDirection,
        vec3 vNomral,
        float roughness,
        float fresnel){
```

```
          float VdotN = max(dot(viewDirection, vNomral), 0.0);
          float LdotN = max(dot(lightDirection, vNomral), 0.000001);
          vec3 h = normalize(viewDirection + lightDirection);

          //Geometric term, Gb = 2(nh)(nv)/vh, Gc = 2(nh)(nl)/lh, G = min(1, Gb, Gc)
          float NdotH = max(dot(h, vNomral), 0.0);
          float VdotH = max(dot(h, viewDirection), 0.0);
          float LdotH = max(dot(h, lightDirection), 0.0);
          float G = min(1.0, min(2.0 * NdotH * VdotN / VdotH, 2.0 * NdotH * LdotN / LdotH));

          //Distribution term
          float D = exp((NdotH * NdotH - 1.0) / (roughness * roughness * NdotH * NdotH)) / (3.14159265 * roughness * roughness * NdotH * NdotH * NdotH * NdotH);

          //Fresnel term
          float F = pow(1.0 - VdotN, fresnel);

          return max(G * F * D / max(3.14159265 * VdotN * LdotN, 0.000001), 0.0);
      }
```

镜面反射颜色计算：

$$specularColor = lightColor \times basicColor * cookTorrance$$

1.5 场景漫游

- 使用`W A S D Q E`控制飞机本身，鼠标控制视角变换
- 飞机控制
  - `Q E`：飞机加速/减速，直接修改`speed`变量
  - `A D`：飞机左转/右转，修改飞机模型矩阵中的`translation`和`rotation`部分，使得飞机绕Z轴旋转的同时沿着Y轴左右移动
  - `W S`：飞机俯冲/爬升，修改飞机模型矩阵中的`rotation`部分，使得飞机绕着Y轴旋转
  - 为了优化飞机控制的操作感，给所有的旋转操作（转弯和爬升俯冲）加上阈值，设定最大旋转角度并在旋转结束后恢复成原来的角度
  - 具体代码实现如下

```
function handleKeyDown(event) {
    if(airCrash) return;    // 飞机已坠毁，禁用控制
    if (String.fromCharCode(event.keyCode) == "Q") {        // 加速，model和view同步
        if(speed < 24) speed += del;    // 飞机最大速度
    }
    else if (String.fromCharCode(event.keyCode) == "A") {   // 飞机向左的旋转效果
        planeIsRotating = true;
        translation[0] -= del;
        if(rotation.rad < 0.7)
            rotation.rad += del / 2;
        rotation.axis = [0, 0, 1];
    }
```

```
    else if (String.fromCharCode(event.keyCode) == "E") {   // 减速，model和view同步
        if (speed > del) {
            speed -= del;
        }
    }
    else if (String.fromCharCode(event.keyCode) == "D") {   // 飞机向右的旋转效果
        planeIsRotating = true;
        translation[0] += del;
        if(rotation.rad > -0.7)
            rotation.rad -= del / 2;
        rotation.axis = [0, 0, 1];
    }
    else if (String.fromCharCode(event.keyCode) == "S") {   // 飞机向下
        planeIsRotating = true;
        if(modelxrotation.rad > 1.3)
        modelxrotation.rad -= del / 2;
        nochange_translation[1] += del;
    }
    else if (String.fromCharCode(event.keyCode) == "W") {   // 飞机向上
        planeIsRotating = true;
        if(modelxrotation.rad < 2.2)
            modelxrotation.rad += del / 2;
        nochange_translation[1] -= del;
    }
}
```

- 视角变换 - 移动视线
  - 根据鼠标位置的移动，计算视线方向，并修改视点、UP向量和目标点
  - 具体代码实现如下

```
function handleMouseDown(event) {
    mouseDown = true;
    lastMouseX = event.clientX;
    lastMouseY = event.clientY;
}
function handleMouseUp() {
    mouseDown = false;
}
function handleMouseOut() {
    mouseDown = false;
}
function handleMouseMove(event) {
    if (!mouseDown) {
        return;
    }
    var newX = event.clientX;
    var newY = event.clientY;
    var deltaX = newX - lastMouseX;
    var deltaY = newY - lastMouseY;
    var radx = 0.0, rady = 0.0;
    const viewRotationMatrix = mat4.create();

```

```
    //旋转角度radx，横向拖动时的变化，将其转化到弧度
    radx = -1 * (2 * Math.PI) * deltaX * 0.5 / cw;
    //旋转角度rady，纵向拖动时的变化，将其转化到弧度
    rady = (2 * Math.PI) * deltaY * 0.5 / ch;
    //获取视点，up向量,和目标点三个三维向量确立视角坐标系
    var vec3_eye = vec3.fromValues(eye[0], eye[1], eye[2]);
    var vec3_up = vec3.fromValues(up[0], up[1], up[2]);
    var vec3_target = vec3.fromValues(target[0], target[1],
target[2]);
    //求出视线方向
    var vec3_eye2target = vec3.create();
    vec3.subtract(vec3_eye2target, vec3_eye, vec3_target);
    //求出横向旋转的方向，即y方向绕其旋转
    var vec3_rotation_y = vec3.create();
    vec3.cross(vec3_rotation_y, vec3_eye2target, vec3_up);
    var rotation_y = [vec3_rotation_y[0], vec3_rotation_y[1],
vec3_rotation_y[2]];
    //x方向绕up旋转，通过rotate变换绕两方向旋转得出变换矩阵
    mat4.rotate(viewRotationMatrix,
        viewRotationMatrix,
        radx,
        up);
    mat4.rotate(viewRotationMatrix,
        viewRotationMatrix,
        rady,
        rotation_y);

    //对视点和up向量进行旋转变换
    vec3.transformMat4(vec3_eye, vec3_eye, viewRotationMatrix);
    vec3.transformMat4(vec3_up, vec3_up, viewRotationMatrix);

    eye[0] = vec3_eye[0]; eye[1] = vec3_eye[1]; eye[2] =
vec3_eye[2];
    up[0] = vec3_up[0]; up[1] = vec3_up[1]; up[2] = vec3_up[2];
    //mat4.lookAt(viewMatrix, eye, target, up);
    lastMouseX = newX;
    lastMouseY = newY;
}
```

- 视角变换 - Zoom in/out/To Fit
  - 根据鼠标滚轮，沿着视线方向改变视点位置，实现视界的放大/缩小
  - 具体代码实现如下

```
function onMouseWheel(event) {
    var distance = Math.pow(eye[0] - target[0], 2) +
Math.pow(eye[1] - target[1], 2) + Math.pow(eye[2] - target[2], 2);
    distance = Math.sqrt(distance);
    if ((event.deltaY > 0 && distance < 15) || (event.deltaY < 0 &&
distance > 3)) {
        eye[0] = eye[0] + (eye[0] - target[0]) / distance *
event.deltaY * 0.001;
        eye[1] = eye[1] + (eye[1] - target[1]) / distance *
event.deltaY * 0.001;
        eye[2] = eye[2] + (eye[2] - target[2]) / distance *
event.deltaY * 0.001;
        sum_deltaY = sum_deltaY + event.deltaY;
```

```
    }
}

function handleKeyDown(event) {
    ...
    else if (String.fromCharCode(event.keyCode) == "Z") {//Zoom to fit
        var distance = Math.pow(eye[0] - target[0], 2) + Math.pow(eye[1] - target[1], 2) + Math.pow(eye[2] - target[2], 2);
        distance = Math.sqrt(distance);
        eye[0] = eye[0] - (eye[0] - target[0]) / distance * sum_deltaY * 0.001;
        eye[1] = eye[1] - (eye[1] - target[1]) / distance * sum_deltaY * 0.001;
        eye[2] = eye[2] - (eye[2] - target[2]) / distance * sum_deltaY * 0.001;
        sum_deltaY = 0.0;
    }
}
```

1.6 Awesomeness指数

1.6.1 体积云实现

云的形状

梯度噪声perlin noise

梯度噪声通过片元的位置和梯度向量得到噪声值，生成具有连续性的纹理

1. 划分网格：对三维空间进行划分，生成三维网格
2. 生成梯度向量。对每个网格的8个顶点生成随机梯度向量

```
    //生成随机梯度向量
    vec3 random_perlin(vec3 p) {
        p = vec3(
                dot(p,vec3(127.1,311.7,69.5)),
                dot(p,vec3(269.5,183.3,132.7)),
                dot(p,vec3(247.3,108.5,96.5))
                );
        return -1.0 + 2.0*fract(sin(p)*43758.5453123);
    }
```

3. 根据网格的八个顶点的梯度向量与当前片元点P的距离，赋权进行点积计算

```
        float a = dot(random_perlin(i),s);
        float b = dot(random_perlin(i + vec3(1, 0, 0)),s - vec3(1, 0,
0));
        float c = dot(random_perlin(i + vec3(0, 1, 0)),s - vec3(0, 1,
0));
        float d = dot(random_perlin(i + vec3(0, 0, 1)),s - vec3(0, 0,
1));
        float e = dot(random_perlin(i + vec3(1, 1, 0)),s - vec3(1, 1,
0));
        float f = dot(random_perlin(i + vec3(1, 0, 1)),s - vec3(1, 0,
1));
        float g = dot(random_perlin(i + vec3(0, 1, 1)),s - vec3(0, 1,
1));
        float h = dot(random_perlin(i + vec3(1, 1, 1)),s - vec3(1, 1,
1));
```

4. 进行平滑插值

```
        // Smooth Interpolation
        vec3 u = smoothstep(0.,1.,s);
        // 根据八个顶点进行插值
        return mix(mix(mix( a, b, u.x),
                    mix( c, e, u.x), u.y),
                mix(mix( d, f, u.x),
                    mix( g, h, u.x), u.y), u.z);
```

细胞噪声worly noise

细胞噪声生成晶胞状纹理，用来丰富云的块状结构和蓬松感

根据voronoi图原理，空间被分为多个块状，每个块状有一个中心点称为特征点。空间中的任一点，到其所在区域的特征点的距离相比到其他区域特征点的距离最小。

1. 生成网格，与perlin noise相同
2. 每个网格生成特征点，该特征点在网格内是随机的

```
    vec3 random_worly(vec3 p) {
        p = vec3(
                dot(p,vec3(127.1,311.7,69.5)),
                dot(p,vec3(269.5,183.3,132.7)),
                dot(p,vec3(247.3,108.5,96.5))
                );
        return fract(sin(p)*43758.5453123);
    }
```

3. 针对当前片元点p，计算出距离点p最近的特征点v，将点p到点v的距离记为F1

```
     float noise_worley(vec3 st) {
         // Tile the space
         vec3 i_st = floor(st);
         vec3 f_st = fract(st);
         float min_dist = 1.;
         for (int i = -1; i <= 1; i++) {
             for (int j = -1; j <= 1; j++) {
                 for (int k = -1; k <= 1; k++) {
```

```
                    vec3 neighbor = vec3(float(i),float(j),float(k));
                    vec3 point = random_worly(i_st + neighbor);
                    float d = length(point + neighbor - f_st);
                    min_dist = min(min_dist,d);
                }
            }
        }
        return pow(min_dist,2.);
    }
```

分形布朗运动

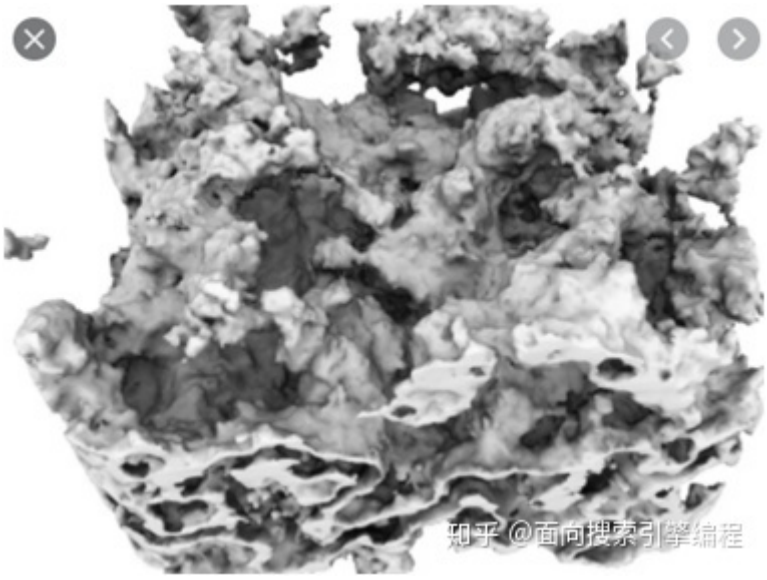
分形布朗运动将不同频率和振幅的噪声进行插值计算，使得云的细节更加丰富。

```
    float noise_perlin_fbm(vec3 p) {
        float f = 0.0;
        p = 2. * p;
        float a = 2.;
        for (int i = 0; i < 5; i++) {
            f += a * noise_perlin(p);
            p = 2.0 * p;
            a /= 2.;
        }
        return f;
    }
    float noise_worley_fbm_abs(vec3 p) {
        float f = 0.0;
        float a = 0.5;
        for (int i = 0; i < 6; i++) {
            f += a * abs(noise_worley(p)-.5);
            p = 2. * p;
            a /= 2.;
        }
        return f;
    }
```

三维噪声纹理生成

根据两种噪声方法，在三维空间中各点生成噪声，噪声值由两噪声混合而成。每个点的噪声值代表该点处的云的密度。

三维噪声示意图：（选自RayMarching实时体积云渲染入门(上) - 知乎 (zhihu.com)）



使用噪声时，将其进行插值混合。首先获得两种低频率噪声的分形，作为云的整体效果基础，之后获得两噪声的分形布朗运动模型，对二者进行混合得到最终效果。

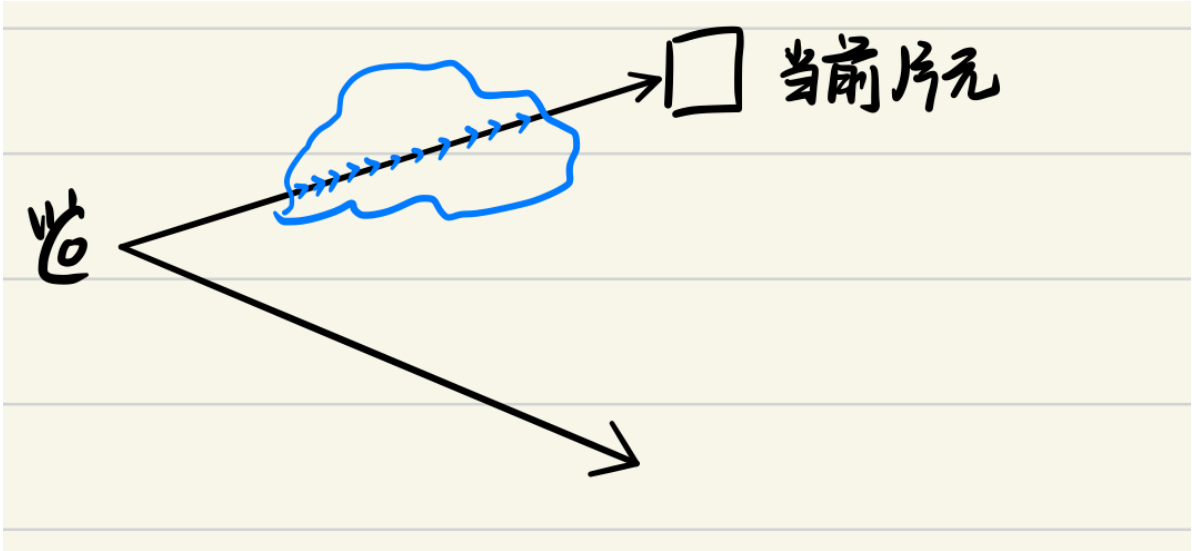
```
        float g1 = 0.5+0.5*noise_perlin( q*0.3 );
        float g2 = 0.5+0.5*noise_worley( q*0.3 );

        float f1, f2;
        f1 = noise_perlin_fbm(q);
        f2 = noise_worley_fbm_abs(q * oct);

        //f1 * a - 0.75, a越大，起伏细节越多
        f1 = mix( f1*0.3-0.75, f1, g1*g1 ) + 0.3;
        //f2 * a - 0.75, a越大，边缘细节越多
        f2 = mix( f2*0.1-0.75, f2, g2*g2 ) + 0.2;

        float f = mix(f1, f2, 0.9);
```

到此为止，获得云的基本形状

云的绘制

光线步进法

光线步进法从相机位置出发，沿光线方向步进，对每一步进位置的云的密度进行累加，即为需要绘制的云的颜色，将其绘制在片元上。密度由噪声函数计算得到。

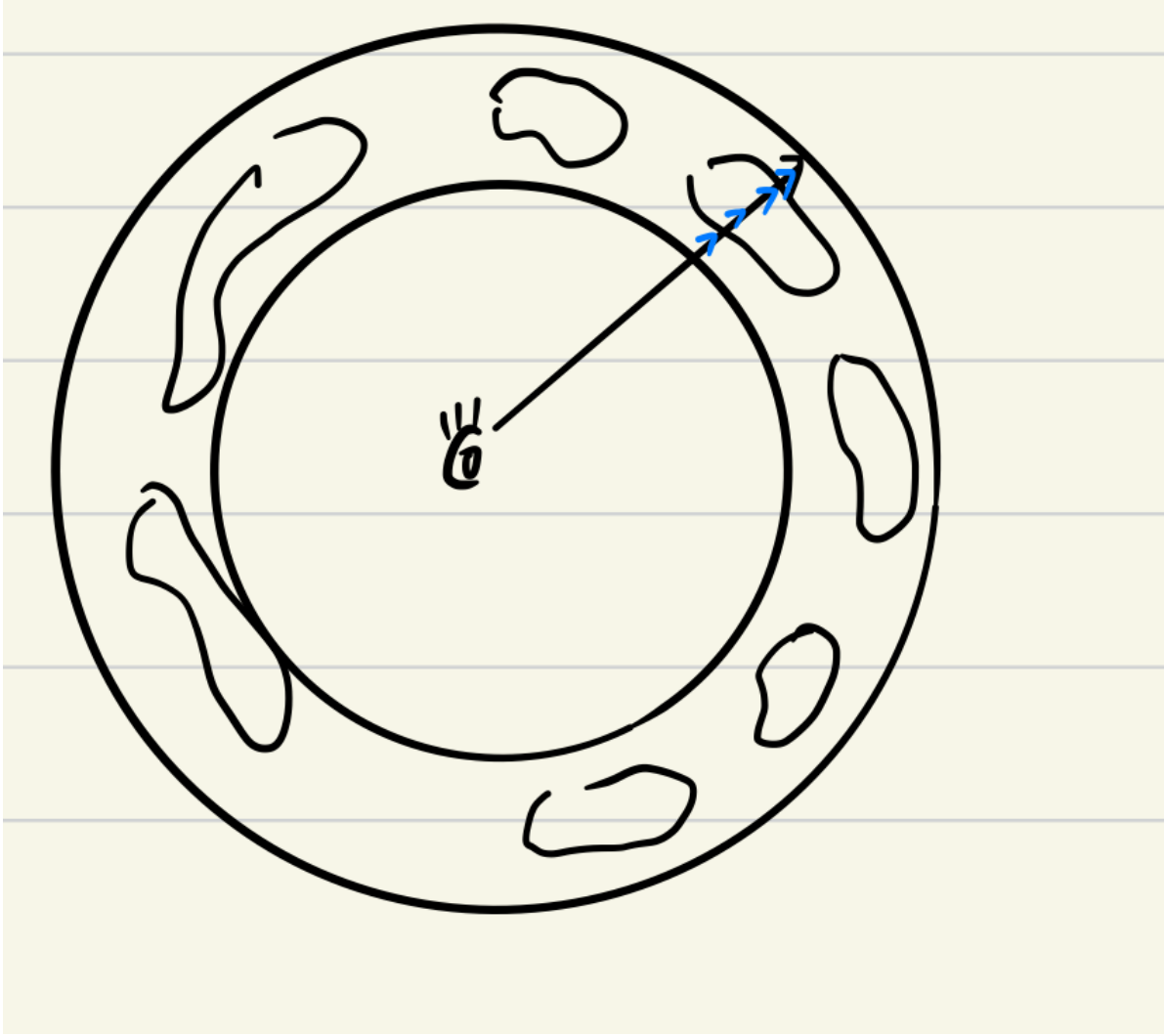


在绘制时，需要使用透明混合：

$$color = bgColor * (1 - cvColor.a) + cvColor$$

$(1 - cvColor.a)$为不透明度，不透明度乘以背景色，最后叠加需要透明的物体的颜色即可进行透明绘制。

光线步进法需要规定云的绘制范围，在本次作业中为了兼顾性能和效果，步进范围为以视点为中心的球壳范围内



而半径与视点位置有关，以实现飞机上升时云的远近变化。

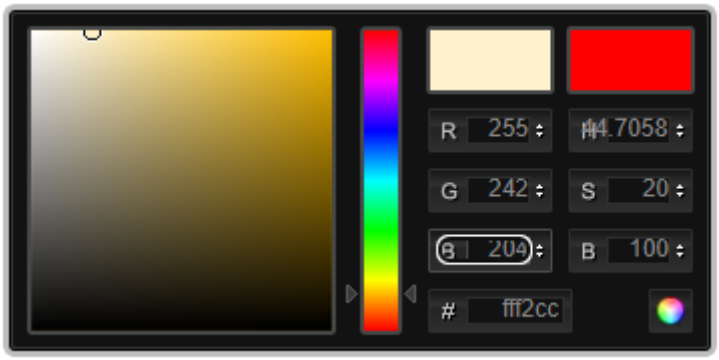
体积云光照效果

体积云的光照分为两个部分：

1. 基础颜色

   基础颜色由累计密度决定，云层越厚则越暗。规定云的基础颜色：

```
vec3(1.0,0.95,0.8)   云的亮部基础颜色
vec3(0.25,0.3,0.35)  云的暗部基础颜色
```



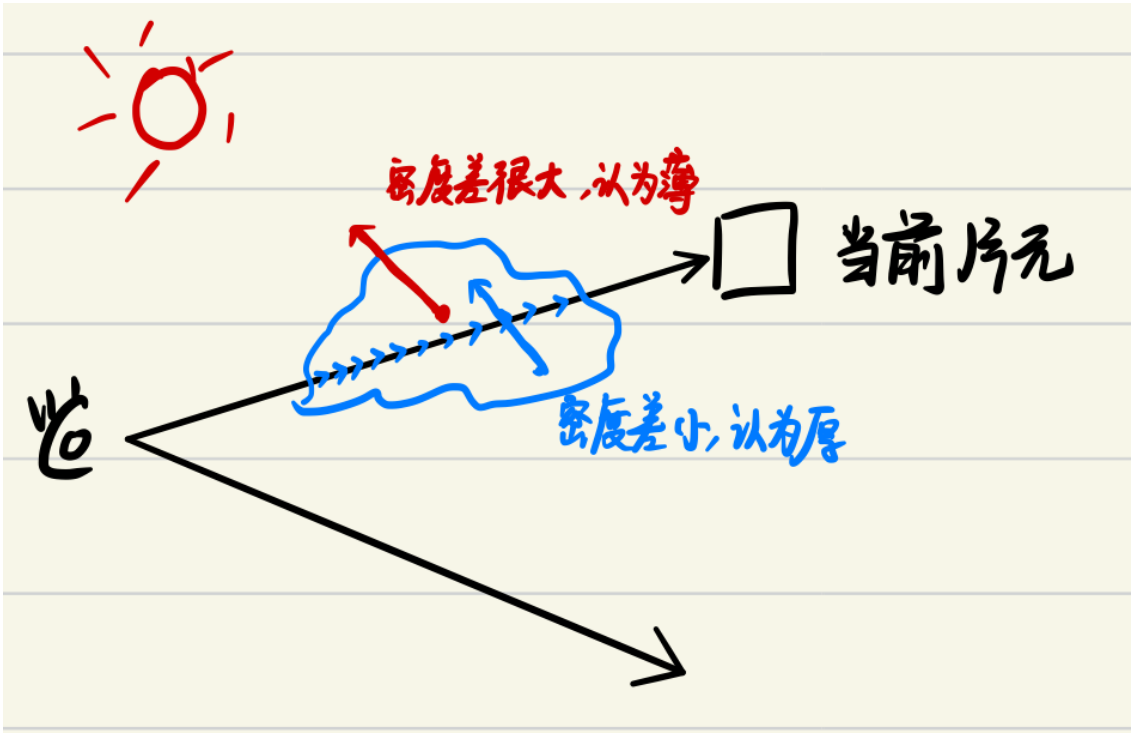
基础颜色部分根据云的密度对基础颜色进行线性混合：

```
vec4  col = vec4(mix(vec3(1.0,0.95,0.8), vec3(0.25,0.3,0.35), den), den);
```

此时云具备层次感。

2. 光照

对于云内一点的亮度确定，需要知道该点与光源之间的云层厚度。根据云层光照颜色的二值性，沿光线方向采样，计算两点之间的密度差，以此粗略估计云层厚度。密度差很大的，说明云层较薄，颜色较亮；密度差小的，说明云层较厚，颜色较暗。



根据厚度，加入光照颜色：

```
float dif = clamp((den - map(ro, pos+0.3 * sundir, oct)) / 0.3, 0.0, 1.0 );
vec3  lin = vec3(0.65, 0.65, 0.75) * 1.1 + 0.8 * vec3(1.0, 0.6, 0.3) * dif;
```

最终实现代码：

对于无云的区域，加入no_cloud进行判断。若有一整段距离无云，则判断其邻域均无云。

对于步进距离，较近处需要高精度，而远处不需要很高精度，所以根据距离对步进长度进行变化。

```
    vec4 raymarch( in vec3 ro, in vec3 rd, in vec3 bgcol){

        // 确认需要ray march的区域
        float tmin, tmax;

        tmin = abs(ro.y) + 0.1;
        tmax = tmin + 2.;

        float t = tmin;

        // raymarch loop
        vec4 sum = vec4(0.0);
        int no_cloud = 0;
        for( int i=0; i<50*kDiv; i++ )
        {
           // step size
           if(no_cloud >= 50)
            break;
           float dt = max(0.3,0.1*t/float(kDiv));

           float oct = 1.;

           //cloud
           vec3 pos = ro + t*rd;
           float den = map(ro, pos,oct);
           if( den > 0.01 ){ // if inside
               // do lighting
               float dif = clamp((den - map(ro, pos+0.3 * sundir, oct)) / 0.3, 0.0, 1.0 );
               vec3  lin = vec3(0.65, 0.65, 0.75) * 1.1 + 0.8 * vec3(1.0, 0.6, 0.3) * dif;
               vec4  col = vec4(mix(vec3(1.0,0.95,0.8), vec3(0.25,0.3,0.35), den), den);
               col.xyz *= lin;
               // fog
               col.xyz = mix(col.xyz, bgcol, 1.0 - exp2(-0.003 * t * t));
               // composite front to back
               col.w    = min(col.w * 8.0 * dt,1.0);
               col.rgb *= col.a;
               sum += col*(1.0-sum.a);
           }
           else{
               no_cloud += 1;
           }
           // advance ray
           t += dt;
```

```
            // until far clip or full opacity
            if( t>tmax || sum.a>0.99 ) break;
        }
        no_cloud = 0;
        return clamp(sum, 0.0, 1.0);
    }
```

##

1.6.2 粒子效果

粒子效果实际上为多个点像素的在一定范围内的随机变化，包含了起始位置、结束位置、粒子消散时间、颜色等。

起始位置、结束位置的确定

使用`(Math.random() * 2 - 1`得到了[-1,1]的随机数，乘以缩放系数后加上中心点的坐标值，即获得了起始、结束位置的坐标

粒子消散时间的确定

每一个粒子的消散时间为[0,1]乘speed，到达lifetime后，粒子消散

```
function initParticle(Program, center, color, num, startscale, endscale, speed) {
    var start = new Array();
    var end = new Array();
    var lifetime = new Array();
    var indices = new Array();
    var colors = new Array();
    for (i = 0; i <= num; i += 1) {
        lifetime.push(Math.random() * speed);
        start.push((Math.random() * 2 - 1) * startscale + center[0]);
        start.push((Math.random() * 2 - 1) * startscale + center[1]);
        start.push((Math.random() * 2 - 1) * startscale + center[2]);
        end.push((Math.random() * 2 - 1) * endscale + center[0]);
        end.push((Math.random() * 2 - 1) * endscale + center[1]);
        end.push((Math.random() * 2 - 1) * endscale + center[2]);
        colors.push(color[0]);
        colors.push(color[1]);
        colors.push(color[2]);
        colors.push(color[3]);
        indices.push(i);
    }
    const buffers = initParticleBuffers(Program.gl, start, end, lifetime, colors, indices);
    return buffers;
}
```

粒子飘动效果

确定粒子的起始、结束位置，以及消散时间后，便可在着色器中计算像素的实时位置

```
tt=alifetime-utime;
pos.xyz=astart+(tt*(aend-astart));
pos.w=1.0;
```

绘制圆形的点

默认情况下，着色器绘制的点像素为立方体，在粒子效果方面看起来就很Minecraft，所以在片元着色器判断每一个绘制点与像素点的距离，舍弃超过0.5像素距离的点，即可得到圆形点

```
float d = distance(gl_PointCoord, vec2(0.5, 0.5));
if(d < 0.5) {// Radius is 0.5
    gl_FragColor.xyz = v_Color.rgb;
    gl_FragColor.w = outlifetime;
}
else discard;
```

1.6.3 雾化实现

雾化的实现，基本就是在着色器中使用某些从相机位置计算的深度或者距离来使颜色或多或少的成为雾色。

在着色器中可以写为

```
gl_FragColor = mix(originalColor, fogColor, fogAmount);
```

将原色和雾色按比例混合

想要绘制基于密度的雾，需要原色和雾色，以及物体到相机的距离来计算雾色的比例。

计算物体到相机的距离

```
vposition = v_Position.xyz;
float fogDistance = length(vposition);
```

获取原色

使用采样器获取纹理贴图的纹理坐标对应的颜色即可

```
vec4 v_Color=texture2D(uSampler,v_TextCoord);
```

获取雾色

雾色的获取稍有些困难，网上的各种教程均将背景色设为灰色，然后将物体的颜色与灰色混合，以达到边缘逐步雾化的效果。但这实际上是一种伪雾化，毕竟在实际的游戏中，背景色不可能是一成不变的颜色，比如本游戏中就使用了天空盒，而天空盒的颜色由天空盒贴图决定。因此，得到天空盒的颜色并将其作为雾色，便能使用插值函数实现雾化效果。

我们注意到，以当前视角看向物体，视点与物体的连线与天空盒的交点，便是在当前状态下，该物体的背景色。

因此，可以使用当前视点的位置和物体的位置，计算出视线的方向。对视线向量做归一化后，其结果与天空盒的世界坐标是一致的，因此，可以直接使用`CubeSampler`对天空盒的贴图进行采样，便得到了真实的背景色

```
vec3 viewDirection = normalize(uEyePosition - vposition); //通过uEyePosition和vpositon算出视线
vec3 TextureCoord = viewDirection;  //天空盒对应的纹理坐标数值上跟视线一样
vec4 fogcolor = textureCube(uCubeSampler, normalize(TextureCoord));  // 背景色
```

使用背景色与灰色进行插值，便得到了雾色的效果

```
float n=0.3;
vec4 grey;
grey=vec4(n, n, n,1.0);
fogcolor = mix(fogcolor,grey,0.5);
```

基于密度的雾

综上，得到了雾色、原色、物体与视点的距离后，便可计算得到实际绘制的颜色，实现雾化效果

```
#define LOG2 1.442695
float fogDistance = length(vposition);
float fogAmount = 1.0 - exp2(-uFogDenisty * uFogDenisty * fogDistance * fogDistance * LOG2);
fogAmount = clamp(fogAmount, 0.0, 1.0);
fogAmount = max(0.4,fogAmount);
vec4 color=mix(v_Color, fogcolor, fogAmount);
```

2 额外要求

2.1 漫游时实时碰撞检测

- 鉴于只需要检测飞机和球体，将所有飞机有可能碰撞到的球体放入`ballSet`集合中，每一帧进行一次碰撞检测
- 采用AABB包围盒（Axis-aligned bounding box）进行碰撞检测，采用包围盒可以简化物体运动过程中的碰撞检测，而AABB包围盒每个面都平行于坐标平面，因此判断两个包围盒是否发生碰撞仅需要判断3个轴方向的交叠部分大小是否大于预定的阈值，如果交叉部分大于阈值则发生碰撞
- 具体的碰撞检测代码如下

```
 class crashObj {
        constructor(id, center, dx, dy, dz, type) {
            this.id = id;
            this.center = center;
            this.dx = dx;       // dx, dy, dz: 球体大小
            this.dy = dy;
            this.dz = dz;
            this.type = type;   // type = 0, 爆炸球; type = 1, 得分球
            this.exist = true;
        }
        check(px, py, pz) {     // px, py, pz: 飞机的大小
            return (Math.abs(this.center[0] + nochange_translation[0] - translation[0]) < px + this.dx) && (Math.abs(this.center[1] + nochange_translation[1] - translation[1]) < py + this.dy) && (Math.abs(this.center[2] + nochange_translation[2] - translation[2]) < pz + this.dz);
        }
    }
```

```
function checkCollision() {
        if (airCrash) return;
        // traversal
        var px = Math.abs(2.5 * planeSize * Math.cos(rotation.rad));
        var py = Math.abs(planeSize * Math.sin(rotation.rad));
```

```
        for (let i = 0; i < ballSet.length; i++) {
            // crash
            if (ballSet[i].check(px, py, 1.5 * planeSize) &&
ballSet[i].exist) {
                if(ballSet[i].type == 1) {      // 得分球
                    change_score();
                    ++score;
                    scoreObjIndex = i;
                    ballSet[i].exist = false;
                    break;
                }
                else if(ballSet[i].type == 0) {  // 爆炸球
                    change_score();
                    scoreObjIndex = i;
                    ballSet[i].exist = false;
                    change_crash();
                    boom_time = 0.001;
                    rotation = new Rotation(0, [0, 0, 1]);
                    modelxrotation = new Rotation(Math.PI / 2, [1, 0,
0]);
                    speed = 1.0;
                    break;
                }
            }
        }
    }
```

- 如果碰撞到爆炸球，调用`change_crash()`引发飞机和球体的爆炸；如果碰撞到的是得分球，调用`change_score()`方法引发球体爆炸并增加游戏得分

2.2 光照明模型细化 —— 实时阴影

实时阴影的原理实际为纹理贴图，本程序中我们通过实时绘制一张“影子”的纹理，并将其贴图在地板上，以达到实时阴影的效果。

2.2.1 纹理贴图

在物体被渲染时，对于每一个像素，检查被投影的纹理是否在范围内，如果在范围内，就从被投影的纹理中采样相应的颜色，如果不在范围内，就从自身的颜色或从其他的纹理中采样颜色。纹理的颜色是通过使用纹理坐标进行查找的，纹理坐标把一个纹理映射到了物体上。

```
const vsSource=`
    attribute vec4 a_position;
    attribute vec2 a_texcoord;

    uniform mat4 u_matrix;

    varying vec2 v_texcoord;

    void main() {
    // 将位置和矩阵相乘
    gl_Position = u_matrix * a_position;
    // 传递纹理坐标到片断着色器
    v_texcoord = a_texcoord;
    }
```

```
`;
const fsSource=`
    precision mediump float;

    // 从顶点着色器中传入的值
    varying vec2 v_texcoord;

    // 纹理
    uniform sampler2D u_texture;

    void main() {
        gl_FragColor = texture2D(u_texture, v_texcoord);
    }
`;
```

2.2.2 渲染到纹理

创建一个纹理贴图，再创建一个新的帧缓冲区，并和这个纹理绑定，此后每次调用 `gl.clear`，`gl.drawArrays`，或 `gl.drawElements` 时，WebGL都会渲染到纹理上而不是画布上。

```
function initFramebufferObject(gl) {
    var framebuffer, texture, depthBuffer;
    // Create a framebuffer object (FBO)
    framebuffer = gl.createFramebuffer();
    // Create a texture object and set its size and parameters
    texture = gl.createTexture(); // Create a texture object
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, OFFSCREEN_WIDTH, OFFSCREEN_HEIGHT, 0, gl.RGBA, gl.UNSIGNED_BYTE, null);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
    // Create a renderbuffer object and Set its size and parameters
    depthBuffer = gl.createRenderbuffer(); // Create a renderbuffer object
    gl.bindRenderbuffer(gl.RENDERBUFFER, depthBuffer);
    gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16, OFFSCREEN_WIDTH, OFFSCREEN_HEIGHT);
    gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
    gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D, texture, 0);
    gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT, gl.RENDERBUFFER, depthBuffer);
    framebuffer.texture = texture; // keep the required object
    // Unbind the buffer object
    gl.bindFramebuffer(gl.FRAMEBUFFER, null);
    gl.bindTexture(gl.TEXTURE_2D, null);
    gl.bindRenderbuffer(gl.RENDERBUFFER, null);
    return framebuffer;
}
```

因此，只需要将物体的影子当作普通的绘制一样，绘制在纹理上，即可以在后续的绘制中将影子纹理贴在地板表面了。

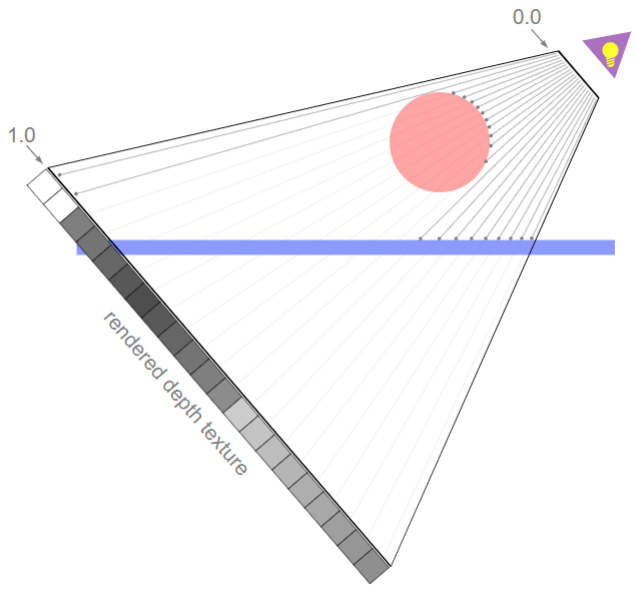
```
var fbo = initFramebufferObject(Program.gl);
Program.gl.bindFramebuffer(Program.gl.FRAMEBUFFER, fbo);// Change the drawing destination to FBO
Program.gl.viewport(0, 0, OFFSCREEN_HEIGHT, OFFSCREEN_HEIGHT);// Set view port for FBO
Program.gl.clear(Program.gl.COLOR_BUFFER_BIT |
Program.gl.DEPTH_BUFFER_BIT);// Clear FBO
drawShadow(Program, objbuffers[0], aircraft_modelMatrix, viewMatrixFromLight, projectionMatrixFromLight);
...//call drawShadow() to draw shadows onto the shadow texture
```

当影子绘制完成后，还需要将绘制区域切换成正常的画布

```
Program.gl.bindFramebuffer(Program.gl.FRAMEBUFFER, null);                //
Change the drawing destination to color buffer
Program.gl.viewport(0, 0, cw, ch);
Program.gl.clear(Program.gl.COLOR_BUFFER_BIT | Program.gl.DEPTH_BUFFER_BIT);
   // Clear color and depth buffer
```

2.2.3 制作阴影纹理

制作阴影纹理的原理是制作一张包含了来自光源视角的深度数据



上图的结果是，球体会得到一个更加接近光源的深度值，而平面会得到一个稍微远离光源的深度值。获得深度信息后，在选择渲染哪个颜色时，就可以从被投影的纹理中进行采样，得到一个采样深度值，然后和当前正在绘制的像素的深度值进行比较。如果当前像素的深度值比采样得到的深度值大，则说明还有其他东西比当前像素更加接近光源。也就是说，某样东西挡住了光源，则该像素是处于阴影中的。

选择光源位置

游戏模拟了太阳的光线，即平行光，但阴影投影时，需要一个与光线方向对应的“点光源”作为视锥的光点，以此绘制阴影贴图。因此，选择光线方向延伸一定距离的位置作为点光源，并设置来自光点的视角矩阵，可以在保持光线方向不变的情况下，模拟视锥的效果，具体代码如下。

```
var LIGHT_X = -lightDirection[0] * light_Scale, LIGHT_Y = -lightDirection[1] * light_Scale, LIGHT_Z = -lightDirection[2] * light_Scale;
```

```
//设置光看的投影矩阵
function setProjectionMatrixFromLight(gl) {
    const fieldOfView = 75 * Math.PI / 180;   // in radians
    const aspect = OFFSCREEN_WIDTH / OFFSCREEN_HEIGHT;
    const zNear = 0.1;
    const zFar = 1000.0;
    const projectionMatrix = mat4.create();
    mat4.perspective(projectionMatrix,
        fieldOfView,
        aspect,
        zNear,
        zFar);
    return projectionMatrix;
}
//设置光看的视角矩阵，根据视点，目标点和上方向确定视角矩阵
function setViewMatrixFromLight(LIGHT_X, LIGHT_Y, LIGHT_Z) {
    //设置view坐标系
    var light_eye = [LIGHT_X, LIGHT_Y, LIGHT_Z];
    var light_target = [0.0, 0.0, 0.0];
    const ViewMatrix = mat4.create();
    // mat4.lookAt(ViewMatrix, light_eye, light_target, up);
    mat4.lookAt(ViewMatrix, light_eye, target, up);
    return ViewMatrix;
}
```

将深度写入阴影纹理

因此，需要从光源视角绘制一次物体，并将深度值写入阴影纹理中

```
const shadow_vsSource = `
attribute vec4 aVertexPosition;

uniform mat4 uProjectionMatrix;  //投影矩阵，用于定位投影
uniform mat4 uViewMatrix;  //视角矩阵，用于定位观察位置
uniform mat4 uModelMatrix;  //模型矩阵，用于定位模型位置

void main() {
  gl_Position = uProjectionMatrix * uViewMatrix * uModelMatrix *
aVertexPosition;
}
`;
const shadow_fsSource = `
precision mediump float;

void main() {
    const vec4 bitShift = vec4(1.0, 256.0, 256.0 * 256.0, 256.0 * 256.0 *
256.0);
    const vec4 bitMask = vec4(1.0/256.0, 1.0/256.0, 1.0/256.0, 0.0);
    vec4 rgbaDepth = fract(gl_FragCoord.z * bitShift);
    rgbaDepth -= rgbaDepth.gbaa * bitMask;
    gl_FragColor = rgbaDepth;
}
`;
```

2.2.4 将阴影纹理贴图到地板上

对于地板，共使用了两张纹理贴图，一张为地板的材质，使用3号采样器，一张为影子贴图，使用7号采样器

先根据地板的纹理坐标对地板的材质进行采样，并将纹理的颜色设为地板的颜色。然后根据地板的像素深度与影子贴图中存储的像素深度比较，若地板的像素深度更深，则地板处于阴影之中，将原先的地板颜色rgb统一乘以0.7，得到影子的效果

```
const plane_vsSource = `
    attribute vec4 a_Position;
    attribute vec4 aNormal; //法向量
    attribute vec2 aTextCoord; //纹理

    uniform mat4 uModelMatrix;  //模型矩阵，用于定位模型位置
    uniform mat4 uProjectionMatrixFromLight;//光源角度的投影矩阵
    uniform mat4 uViewMatrixFromLight;//光源角度的视角矩阵
    uniform mat4 uProjectionMatrix;  //投影矩阵，用于定位投影
    uniform mat4 uViewMatrix;  //视角矩阵，用于定位观察位置
    uniform mat4 uReverseModelMatrix; //模型矩阵的逆转置

    varying vec4 v_PositionFromLight;
    varying vec3 v_Normal;
    varying vec4 v_Position;
    varying vec2 v_TextCoord;

    void main() {
        gl_Position =  uProjectionMatrix * uViewMatrix * uModelMatrix * a_Position;
        v_Normal = normalize(vec3(uReverseModelMatrix * aNormal));
        v_Position = uModelMatrix * a_Position;
        v_TextCoord = aTextCoord;
        v_PositionFromLight =uProjectionMatrixFromLight * uViewMatrixFromLight * uModelMatrix * a_Position;
    }
    `;
const plane_fsSource = `
    precision mediump float;

    uniform sampler2D uShadowMap;
    uniform sampler2D uSampler;

    uniform vec3 uLightColor; //光颜色强度
    uniform vec3 uLightDirection; //光线方向
    uniform vec3 uAmbientLight; // 环境光
    uniform float plane_height; //飞机高度

    varying vec4 v_PositionFromLight;
    varying vec3 v_Normal;
    varying vec4 v_Position;
    varying vec2 v_TextCoord;

    float unpackDepth(const in vec4 rgbaDepth) {
        const vec4 bitShift = vec4(1.0, 1.0/256.0, 1.0/(256.0 * 256.0), 1.0/(256.0 * 256.0 * 256.0));
        float depth = dot(rgbaDepth, bitShift);
        return depth;
```

```
    }

    void main() {
        vec4 v_Color=texture2D(uSampler,v_TextCoord);
        vec3 lightDirection = normalize(uLightDirection);
        //计算cos入射角 当角度大于90 说明光照在背面 赋值为0
        float nDotLight = max(dot(lightDirection, v_Normal), 0.0);
        //计算漫反射光颜色
        vec3 diffuse = uLightColor * v_Color.rgb * nDotLight;
        // 环境反射光颜色
        vec3 ambient = uAmbientLight * v_Color.rgb;
        vec4 tmp_Color = vec4(diffuse + ambient, v_Color.a);

        vec3 shadowCoord =
(v_PositionFromLight.xyz/v_PositionFromLight.w)/2.0 + 0.5;
        vec4 rgbaDepth = texture2D(uShadowMap, shadowCoord.xy);
        float depth = unpackDepth(rgbaDepth);
        float visibility =(shadowCoord.z > depth + 0.0015)? 0.7:1.0;
        // gl_FragColor = vec4(tmp_Color.rgb * visibility, 1.0-
plane_height/50.0);
        gl_FragColor = vec4(tmp_Color.rgb * visibility, 1.0);

    }
    `;
```

以下为实时阴影效果图