

浙江大学



图形学大程报告

项目 GitHub: <https://github.com/AlanSwift/CGProject>

课程名称:	计算机图形学
姓 名:	沈锴 李孟择 赵宣栋 刘栩威
学 院:	计算机科学与技术学院
专 业:	计算机科学与技术
学 号:	3150102098 3150104805 3150104910 3150102179
指导教师:	童若锋

2018 年 1 月 18 日

目录

1. 功能说明	3
2. 实现细节	4
2.1. obj 读取	4
2.2. obj 导出	6
2.3. obj 模型的处理	7
2.4. 场景漫游及视角切换	8
2.5. 游戏逻辑	9
2.6. 基本图形的显示	10
2.7. 阴影	12
2.8. 阴影渲染	14
1) 阴影失真	14
2) 采样过多	15
3) 锯齿	15
4) 多光源阴影	15
2.9. camera	16
2.10. 光照	17
2.11. 碰撞检测	18
1) 半平面交	19
2) OBB 算法	20
2.12. 波浪效果	22
2.13. 粒子系统	27
1) 爆炸系统	28
2) 火焰系统	30
3. 流程展示图片	34
4. 感想	37

1. 功能说明

我们游戏设计的目的是为了模拟飞机在城市中的飞行和投射导弹进行轰炸的过程。一个飞机在城市中漫游，然后可以在某个时间发射导弹，导弹或者飞机撞到建筑物时，会产生爆炸的效果。我们所采用的是上海的城市模型，然后在城市的周围是海洋的水波效果，在水波之外是天空盒，为远山和水面。

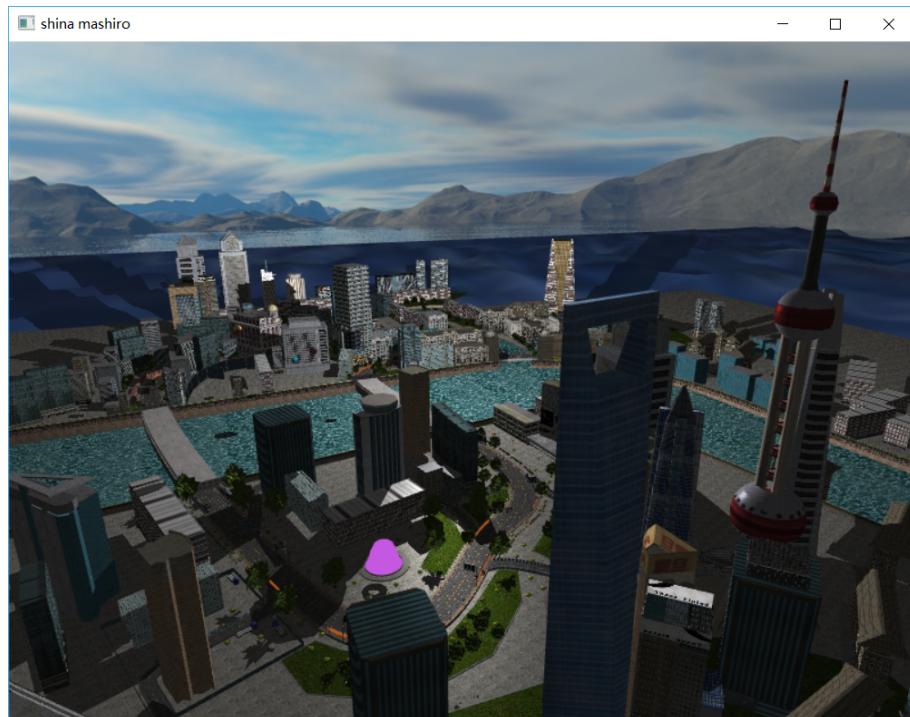


Figure 1 城市整体图

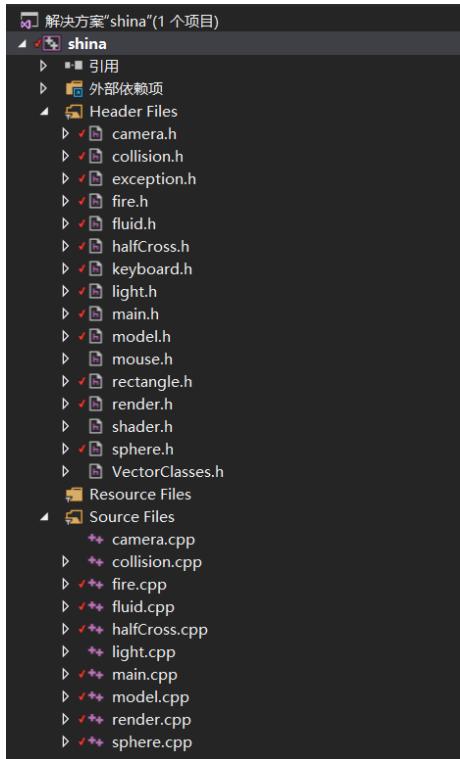


Figure 2 代码架构

2. 实现细节

2.1. obj 读取

我们需要写一个对 **obj** 进行读入的函数。为了写这个函数需要先了解 **obj** 的格式。**obj** 里面主要有以下几种信息：

1. **v** 开头的行表示一个顶点，有下 **x,y,z** 三个坐标。
2. **vt** 开头的行表示纹理的坐标位置，同样有 **x,y,z** 三个坐标（部分 **obj** 中可能仅有 **x,y** 两个坐标，但是为了通用，我们需要写一个都能读取的函数）。
3. **vn** 开头的行表示表示顶点的法线方向，有 **x,y,z** 三个坐标。
4. **f** 开头的行表示一个面，有 **v, v/vt, v/vt/vn** 和 **v//vn** 三种格式，这里的 **v, vt, vn** 都是相应的索引。一般一个 **f** 开头的行会有三组这样的索引，形成一个三角形，但是在有些情况下，也可以有超过三个，这样就是把它们当成一系列三角形的组合读入。

5. `usemtl` 开头的行代表使用新的材质，后面的字符串是材质的名字，材质的定义可以在.mtl 文件中找到。
6. `mtllib` 声明了这个 `obj` 对应的 `mtl` 文件，可以在其中找到材质的定义，一般这个声明会在 `obj` 的开头。
7. `g` 代表了一个组，一个组是物体的一个组成部分。
8. `#` 代表注释。

mtl 中主要有以下几种信息。

1. `newmtl` 开头的行代表一种新的材料，后面接材料的名称。`Kd r g b` 代表物体的漫反射系数
2. `Ka r g b` 代表物体对环境光的系数。
3. `Ks r g b` 代表物体的镜面反射系数。
4. `Ka_map` 代表了这种材质对应的纹理信息，后面是一张图片的路径。

由这些信息，我们可以写出相应的函数进行读取。

首先，我们定义了类的信息。有一个 `Material` 类，其中包含了材质的信息。

```
class Material {
public:
    GLuint texName;
    std::string name; // 材质名字
    glm::vec3 kd, ka, ks; // 材质信息
    Bitmap src; // 图片信息
    bool is_src; // 是否有图片，有则使用图片画，否则用材质画
```

其中的 `Bitmap` 为图片的信息，因为图片的格式很多，我们不能一一了解格式的读取方式，因此对图片的读取我们直接采用了 MFC 封装好的 `Cimage` 类。

此外，我们还定义了一个 `Group` 类，一个 `Group` 类中有一个材料信息，以及用这个材料画的点的位置坐标。

```
class Group {
public:
    vector<float> pos;
    vector<float> coord;
    vector<float> color;
    vector<float> normal;
    Material material;
```

最外层是一个 `texture` 类，这个类中，我们用了一个 `vector<Group>` 来记录所有 `obj` 中的点的信息，以及一个 `hide` 变量表示是否隐藏，一个 `model` 向量来记录位移，旋转，缩放等信息。

在读取时，我们先读取这行的第一个字符，判断相应的操作：

- (1) 如果是 `v` 开头的，分别将三个坐标 `push` 到相应的临时的 `vector` 中。
- (2) 如果是 `m` 开头的，则读取 `mtl` 文件的路径，打开 `mtl` 文件，进行对 `mtl` 文件的读取。每读取到 `newmtl` 时，将一个新的 `Group` 给 `push` 到类的 `vector<Group>` 中，并且后面读取的材质信息都是这个类的。
- (3) 如果读取到 `u`，则对材质的名字在所有 `group` 的材质间进行匹配，匹配到了之后将 `active` 变量设置为该 `group` 的索引。后面的点全部加入都这个 `group` 中。
- (4) 如果读取到的是 `f`，则对检查是四种模式中的哪一种，这一行的读取都按照这个模式进行，先将前三个点的信息 `push` 到对应 `group` 的 `vector<pos>` 中，（这里需要注意 `f` 的最小索引是 1，并且一个 `f` 代表了一个点的 `x,y,z` 坐标，因此一个 `push` 需要按照 `x,y,z` 进行三个 `push`）。如果当前的模式没有 `vt`，则 `push` 三个 0 进入，如果当前的模式没有 `vn`，则根据三个点的坐标计算平面的法线方向，将这个法线方向作为三个点的法线方向。在读完三个顶点之后再次按照模式检查是否还有信息，若读到新的信息，则用一个 `while` 循环进行循环读取，这里读取的时候，每读取一次，都需要加入一个新的三角形。

2.2. `obj` 导出

由于我们读入的时候信息根据材质分成了很多 `group`，因此我们在导出 `obj` 的时候，是根据一个 `group` 导出的，一个 `obj` 可能会导出多个 `obj` 和 `mtl` 信息，分别命名为 `name_1`, `name_2`, ..., `name_n`。

因此我们的 `write` 函数是 `group` 类里面的成员函数。在导出 `obj` 的时候，用一个 `for` 循环对所有 `group` 进行导出即可。

`group` 类的导直接对其中的 `v,vt,vn` 信息进行成行打印，而 `f` 就可以写成 `1/1/1, 2/2/2, ...n/n/n` 的形式。由于一个 `group` 对应一个 `material`，因此 `mtl` 文件只有一

个材质，命名为 Default。根据 material 中的材质信息按行填写即可。最后，如果有图片信息，则根据读入的图片数组将该数组重新转换为图片即可。

在 write 函数需要一个 name 的参数，这个导出的 obj 会被命名为 name.obj，mtl 文件会被命名为 name.mtl，图片信息为 name.jpg（后缀取决于原先图片的形式）

2.3. obj 模型的处理

我们从网上找到飞机和导弹的 3dmax 模型，一般找到的飞机的模型都带有轮子，我们将轮子删除。导入城市模型后，根据城市的大小调整飞机和导弹的大小。因为飞机要实现从公路上起飞的效果，我们在软件中将飞机和导弹调整到城市的公路上。一切完成后，导出飞机和导弹的 obj。

城市模型在网上找到的很多都存在着问题，或者是没有纹理，或者是与后续的操作配合不起来。经过多次的寻找，我们终于找到了一个合适的城市模型。对这些模型稍作处理，通过软件找到高楼的 8 个点，成片的低矮楼房在软件中找到总的八个点，就可以继续进行后续的操作了。

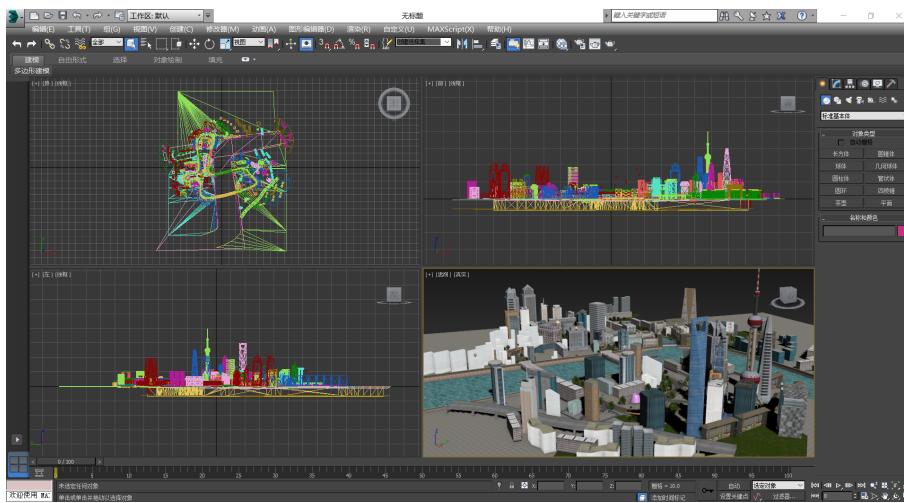


Figure 3 使用 3Dmax 软件处理 obj 模型

2.4. 场景漫游及视角切换

监听键盘的按键，当按下对应的键的时候，对相机做前后左右上下的移动。鼠标用于控制相机的观察方向变换。相机通过旋转矩阵，实现观察方向的变换。每次鼠标点击屏幕的时候，鼠标当前的点击位置会被记录下来。鼠标点击后，拖动的时候，这个位置会连续变化，这个时候根据位置的变化旋转视角就可以了。但是这里存在一个问题，当用户第一次点击屏幕的时候，如果和上一次的位置有较大的偏差，相机的观察方向就会突变到意想不到的位置。为了解决这个问题，我们在程序中对鼠标前后两个位置坐标的变化做判断，如果大于某一个值，就舍弃这次的相机旋转，并记录想当前的鼠标位置。经过测试，这个方法很好的解决了问题。同时，我们在游戏的过程中可以通过按 z 键来进行第一和第三视角的切换，第一视角设定为在飞机和导弹的身后。



Figure 3 第一视角

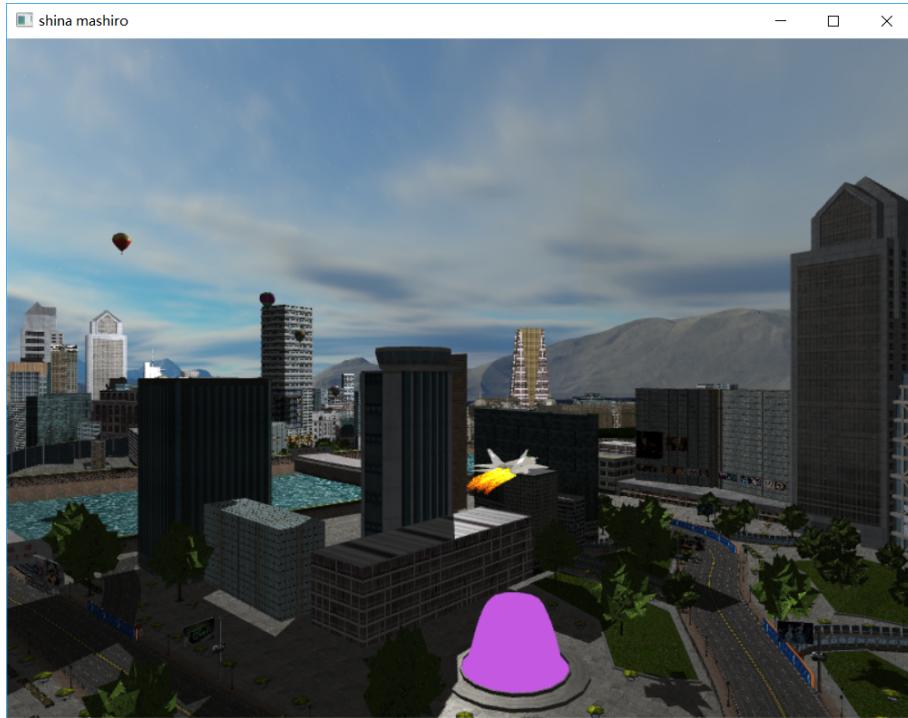


Figure 4 第三视角

2.5. 游戏逻辑

我们的游戏逻辑是这样的：首先进入相机的漫游，按 ASDWQE 可以上下左右前后漫游整个城市，然后我们可以移动相机，对于相机和飞机进行碰撞检测，如果相机和飞机碰撞的话，视角切换到第一视角。或者我们可以按键盘 0 键，直接进入飞机起飞的阶段。

飞机起飞过程中，首先是进入自动起飞阶段，飞机调整前后仰角和飞行速度和高度，然后进入自主控制飞行的阶段。在自主飞行阶段，可以通过按键设置飞机的飞行和移动，按空格键之后发射导弹，在导弹发射之后，可以通过按键控制导弹的飞行，直到导弹和楼碰撞为止。流程图如下：

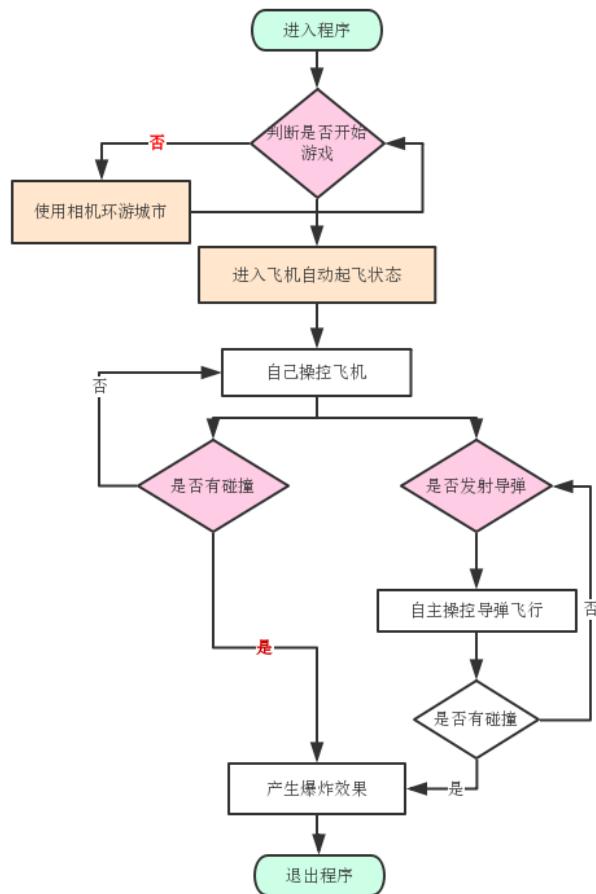


Figure 5 游戏流程图

2.6. 基本图形的显示

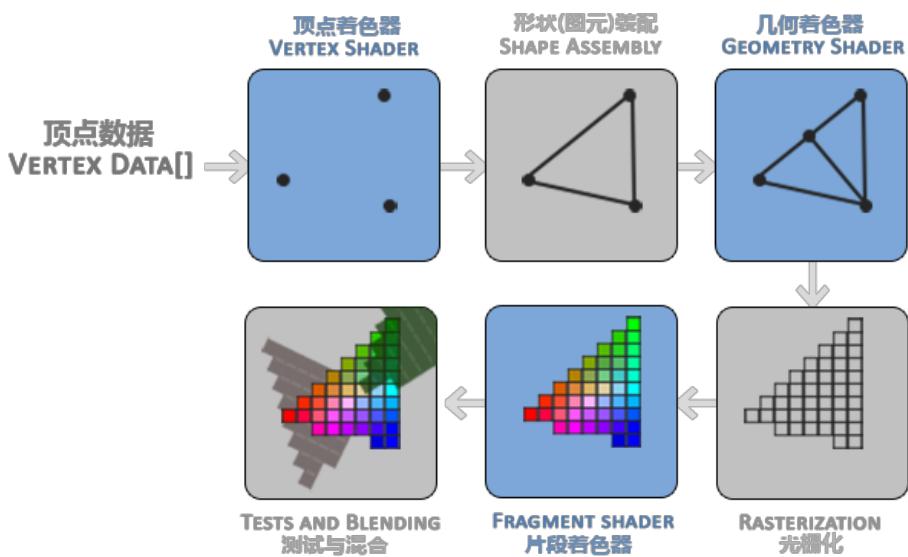
对于可编程管线，能够显示的图形只有点、线、三角形。我们的工程中所有的形状都由基本的三角形组成。

在 OpenGL 中，任何事物都在 3D 空间中，而屏幕和窗口却是 2D 像素数组。这导致 OpenGL 的大部分工作都是关于把 3D 坐标转变为适应你屏幕的 2D 像素。3D 坐标转为 2D 坐标的处理过程是由 OpenGL 的图形渲染管线(Graphics Pipeline，大多译为管线，实际上指的是一堆原始图形数据途经一个输送管道，期间经过各种变化处理最终出现在屏幕的过程)管理的。图形渲染管线可以被划分为两个主

要部分：第一部分把你的 3D 坐标转换为 2D 坐标，第二部分是把 2D 坐标转变为实际的有颜色的像素。

图形渲染管线其实是一种流水线，它接收一组 3D 坐标，然后转变成为 2D 像素的输出。它被划分成为几个阶段，按照流水的形式连接。因为高度流水化，因此具有非常多的核心，每一个核心都会运行叫 **Shader** 的程序完成计算。

具体的过程可以看下图：



顶点着色器把每一个单独的顶点作为输入，把坐标转换成齐次坐标，完成基本的矩阵变换。

顶点着色器的输出会经过图元装配将输出传递给几何着色器，几何着色器可以根据输入产生非常多的顶点，其输出会被传入光栅化阶段，被映射为相应的像素，传入片段着色器。

片段着色器会计算一个像素的最终颜色。

最终输出会经过 **alpha** 测试和混合阶段，并经过深度测试。

因此我们需要配置两个必选着色器（顶点、片段）和一个可选着色器（几何）。

Texture 类的显示是整个工程最复杂的部分。首先我们只关心基本的显示。

基本的显示只要把坐标、纹理、纹理坐标输入即可。

首先绑定缓冲区：

```
glBindVertexArray(graph.material.vao);
```

然后将点的信息传入 buffer:

```
glBindBuffer(GL_ARRAY_BUFFER, graph.material.positionBufferHandle);
glBufferData(GL_ARRAY_BUFFER, graph.pos.size() * 4, graph.getPos(),
GL_STATIC_DRAW);
 glEnableVertexAttribArray(0);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), NULL);
 glBindBuffer(GL_ARRAY_BUFFER, graph.material.coordBufferHandle);
 glBufferData(GL_ARRAY_BUFFER, graph.coord.size() * 4, graph.getCoord(),
GL_STATIC_DRAW);
 glEnableVertexAttribArray(1);
 glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), NULL);
```

然后着色器就可以收到数据。

顶点着色器: 首先转换成其次坐标，然后左乘变换矩阵并乘 camera 的变换矩阵和投影矩阵。

```
v_Position = (u_modelMatrix * vec4(a_Position, 1.0)).xyz;
gl_Position = u_projection * u_view * vec4(v_Position, 1.0);
```

纹理部分也很容易:

```
gl_FragColor = vec4(texture2D(u_textureMap, v_Coord).rgb, 1.0);
```

2.7. 阴影

阴影的原理是 z-Buffer。

它的原理是从光源的位置看出去，将能看到的东西都点亮，如果没有被看到，就贴一个黑色的纹理。

关键在于如何计算 z-Buffer，我使用的是深度贴图，将深度信息存储在一个纹理中。做法是先产生一个帧缓冲对象，然后创建一个 2D 纹理，提供给帧缓冲的深度缓冲。

然后将顶点全部传进 shader，渲染一遍。比较有意思的是如何产生从光源看出去的矩阵，做法是和 camera 一样，使用 glm 的函数，可以产生从光源开始的

`lookat` 矩阵，然后乘上透视投影矩阵，就能将世界坐标变换到光源所在的坐标系。

做法如下：

```
lightProjection = glm::perspective(60.f, (float)SHADOW_WIDTH /
(float)SHADOW_HEIGHT, 8.f, 10000.0f);

lightView = glm::lookAt(light->pos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));

light->lightMatrix = lightProjection * lightView;
```

然后渲染一下深度贴图即可。

```
glBindFramebuffer(GL_FRAMEBUFFER, light->frameHandle);

glClear(GL_DEPTH_BUFFER_BIT);

	glBindBuffer(GL_ARRAY_BUFFER, graph.material.spositionBufferHandle);
glBufferData(GL_ARRAY_BUFFER, graph.pos.size() * 4, graph.getPos(),
GL_STATIC_DRAW);
```

深度贴图需要一个专门的着色器。

顶点着色器只做坐标变换：

```
gl_Position = u_lightMatrix * u_modelMatrix * vec4(a_Pos, 1.0);
```

片段着色器是空的：

```
void main()
{
}
```

但实际上这里有一句隐藏的代码：

```
gl_FragDepth = gl_FragCoord.z;
```

它会自己去更新深度缓冲。然后经过渲染，这个深度贴图的每一个像素都会填充一个最近的 z 值，我们渲染真正的图形的时候只要去深度贴图中拿坐标查一个深度贴图的 r 即可。

2.8. 阴影渲染

阴影渲染正如上一节所写的，需要按照坐标去深度贴图中查找 z 值，和当前 z 做一个对比，如果超过了，就将其设置成阴影。

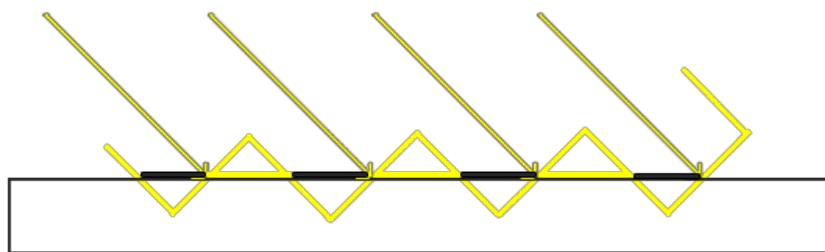
```
float currentDepth = projCoords.z;
```

最近的 $depth$ 获取并不是简单的查点，否则得到的阴影会有失真等一系列问题，按照 `opengl` 的教程，我解决了以下的问题：

1) 阴影失真

因为阴影贴图受限于解析度，在距离光源比较远的情况下，多个片元可能从深度贴图的同一个值中去采样。图片每个斜坡代表深度贴图一个单独的纹理像素。你可以看到，多个片元从同一个深度值进行采样。

虽然很多时候没问题，但是当光源以一个角度朝向表面的时候就会出问题，这种情况下深度贴图也是从一个角度下进行渲染的。多个片元就会从同一个斜坡的深度纹理像素中采样，有些在地板上面，有些在地板下面；这样我们所得到的阴影就有了差异。因为这个，有些片元被认为是在阴影之中，有些不在，由此产生了图片中的条纹样式。



解决方法是用一个最近的 $depth$ 减去一个 $bias$:

```
currentDepth - bias
```

这个 $bias$ 我设置成 0.005 ，这个值虽然很小，但可以纠正绝大部分的阴影失真。

2) 采样过多

使用 z-buffer 算法有一个问题，就是它将所有‘不可见的物体’都认为是在阴影中，不管它是不是真的不可见。（我们使用了透视矩阵左乘了坐标，限制了一定区域是可见的，脱离这个区的点虽然是不可见的，但是不在阴影中）。因此需要对 z 特判：

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

3) 锯齿

z-buffer 还有一个问题，就是它是按照像素分的，其深度值只有像素点个，但是屏幕的点是不定的，这个信息不对成很可能造成不连续，也就是锯齿。

简单的处理方法是平均采样：

```
vec2 sTextureSize = 1.0 / textureSize(u_shadowMap[idx], 0);

for(int x = -1; x <= 1; ++x) {
    for(int y = -1; y <= 1; ++y) {
        float pcfDepth = texture(u_shadowMap[idx], projCoords.xy + vec2(x, y)
* sTextureSize).r;

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

然后就可以得到相应的 shadow 值，如何和别的参数混合呢：方法如下：

```
vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * fragColor;
```

4) 多光源阴影

多光源阴影扩展比较容易，只要对每一个光源都产生一个深度贴图，然后对于每一个光源都渲染一遍，产生每一个光源的 shadow 值，做一个叠加即可。可以想象，如果每一个物体都在阴影区，就是最黑的，其次是第二黑这样。

```

for(int i = 0; i < u_lightNum; i++){
    lighting += calcLight(i);
}

```

2.9. camera

摄像机是一个假想的设备，实际上所有的场景的渲染都是基于 camera 的位置和方向。

首先是 camera 的位置的确定。Camera 定位采用的三维极坐标系，因此需要三个参数，也就是 r, theta, phi。

```

GLfloat theta, phi;
GLfloat factor;

```

观察者坐标系的确定需要两个参数：camera 的观察方向，camera 的 upper 方向。为了方便，我还是用了 eye 表示 camera 的方向，毕竟极坐标系是对于用户不友好的。因此需要同时维护 eye 和极坐标参数。

1) camera 的 move

Camera 的 move 是将二维的移动映射到三维空间，我规定，只映射 xz，y 不变，因此是一个简单的转换：

这个函数接受两个参数：

```
void moveCamera(GLfloat deltaX, GLfloat deltaY);
```

实现如下：

```

eye.x -= (deltaX*sin(theta) + deltaY*cos(theta));
eye.z -= ( -deltaX*cos(theta) + deltaY*sin(theta));

```

2) camera 的 rotate

Rotate 的过程也是将二维的变换映射到三维，规定观察方向是不变的，也就是 eye+dir 是不变的。

然后 `theta` 加上 `x` 的移动, `phi` 加上 `y` 的移动, 最后更新一下 `eye` 和 `dir` 即可。

```
vec3 look = eye + dir;//not change

theta += rx;

phi = inner<float>(0.01, (float)(PI - 0.01f), phi - ry);

//phi = inner<float>(0.001f, (float)(PI / 2 - 0.001f), phi + ry);

//change axis

eye.x = look.x + factor*sin(phi)*cos(theta);

eye.z = look.z + factor*sin(phi)*sin(theta);

eye.y = look.y + factor*cos(phi);

dir = look - eye;
```

3) camera 的 zoom

`Zoom` 的过程只要将 `factor` 更新一下即可, 但是为了一些不必要的麻烦, 我规定了一下 `zoom` 的上下界, 然后更新一下 `eye` 和 `dir` 即可。

```
GLfloat preValue = factor;

factor = inner<float>(0.001, 2147483647, factor*dx);

GLfloat c = factor/preValue;

eye = eye + dir*(1-c);

cout << factor << endl;

dir = dir*c;
```

2.10. 光照

光照的渲染比较麻烦, 需要考虑的特别仔细。

首先我实现的是局部光照模型。主要结构由 3 个元素组成: 环境(Ambient)、漫反射(Diffuse)和镜面(Specular)光照。

每一个物体可以定义自己的材质, 当描述物体的时候, 我们可以使用 3 种光照元素: 环境光照(Ambient Lighting)、漫反射光照(Diffuse Lighting)、镜面光照

(Specular Lighting) 定义一个材质颜色。通过为每个元素指定一个颜色，我们已经对物体的颜色输出有了精密的控制。现在把一个镜面高光元素添加到这三个颜色里，这是我们所需要的所有材质属性。

最后还有一个情况就是光照纹理，就是我们的实际颜色是查纹理贴图得到的，需要将光照和纹理混合。

首先是获得纹理中真正的颜色：

```
vec3 fragColor = texture2D(u_textureMap, v_Coord).rgb;
```

然后需要把法向转换成单位向量，计算光的方向的向量并单位化，对法向和光的方向进程点乘计算光对当前实际的散射影响，再乘以光的颜色。

```
vec3 fragNormal = normalize(v_Normal);
vec3 lightColor = u_lightInfo[idx].u_lightDiff;
vec3 ambient = u_lightInfo[idx].u_lightAmb;
vec3 lightDir = normalize(u_lightInfo[idx].u_lightPos - v_Position);
float diff = max(dot(lightDir, fragNormal), 0.0);
vec3 diffuse = diff * lightColor;
```

然后是镜面光照。

```
vec3 viewDir = normalize(u_eyePos - v_Position);
vec3 reflectDir = reflect(-lightDir, fragNormal);
float spec = 0.0;
vec3 halfwayDir = normalize(lightDir + viewDir);
spec = pow(max(dot(fragNormal, halfwayDir), 0.0), 64.0);
vec3 specular = spec * u_lightInfo[idx].u_lightSpec * lightColor;
```

最后和阴影混合一下即可：

```
vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * fragColor;
```

2.11. 碰撞检测

碰撞检测我们使用了两种策略：1.对于大面积的建筑使用半平面交算法粗判。
2.对于少数几幢有特点的建筑使用 **obb** 算法进行立体的细判。

1) 半平面交

这个算法首先是将建筑物产生包围盒分割开，这个可以用建模软件很容易的完成。然后我写了一个类专门导入这个 `obj` 但是不会显示，只是将不同的建筑分成一个个的 `group`。

对于每一个 `group` 都生成一个凸包。

a) 凸包生成算法

生成凸包的算法是 `Graham scan` 算法，复杂度是 $O(N \log N)$ 。算法是将所有的点先预处理按照极角排序，然后做下列步骤：

Step1: 选定 x 坐标最小（相同情况 y 最小）的点作为极点，这个点必在凸包上；

Step2: 将其余点按极角排序，在极角相同的情况下比较与极点的距离，离极点比较近的优先；

Step3: 用一个栈 S 存储凸包上的点，先将按极角和极点排序最小的两个点入栈；

Step4: 按序扫描每个点，检查栈顶的两个元素与这个点构成的折线段是否“拐”向右侧（叉积小于等于零）；

Step5: 如果满足，则弹出栈顶元素，并返回 **Step4** 再次检查，直至不满足。将该点入栈，并对其余点不断执行此操作；

Step6: 最终栈中元素为凸包的顶点序列。

b) 碰撞检测思路

我是如何使用半平面交做碰撞检测的呢，首先将所有的点投影到 xz 平面，使用 `graham` 扫描线方法对每一个建筑预处理一个凸包，这一步因为点非常多，因此很耗时。然后将飞机生成一个包围盒。这一步只要在不同的维度找到一个最大点就好了。然后将包围盒存起来。每一次飞机遇到只要将包围盒左乘一个变换矩阵就是新的包围盒，将其投影到 xz 平面，然后做一个扫描线算法生成新的凸包，

由于点只有 8 个，这一步时间很少。最后将这个凸包和其它的所有凸包做一个半平面交，求一下核的面积即可。

c) 半平面交算法

首先需要将凸包分解，分解成一个个的半平面。两辆凸包半平面放在一 `vector` 里面，求一下核面积即可。

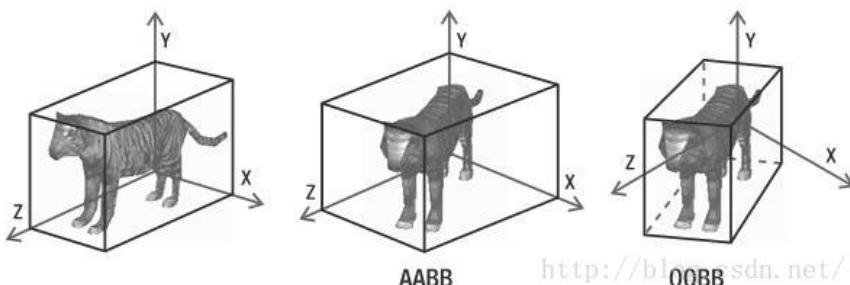
这一步只能形象的描述一下算法的步骤。

其原理是切割一个核。首先还是按照极角排序，用一个双向队列保存决策，每一次新增一个半平面都可能让队首或者队尾的一些半平面变的没用，可以出队列。判断是否由于只要看直线是否在左边即可。这个可以使用叉积去判断。最后全部添加进去之后只要计算一下核的面积，如果是 0，说明是没有碰撞，否则就是有碰撞的。

2) OBB 算法

obb 碰撞检测分为两步，第一步为生成有向包围盒，第二步检测包围盒之间的碰撞。

obb 包围盒与 **AABB** 包围盒不同，**obb** 包围盒是有向包围盒，它可以最大程度上精确碰撞检测：



首先我们需要生成有向包围盒。通过 PCA 获得特征向量，这些特征向量就是 **obb** 包围盒的三个轴。整体流程大致为：计算协方差矩阵，用雅克比方法计算特征向量，施密特正交化得到三个轴，最后确定中心点和求出三个轴方向上的半长。

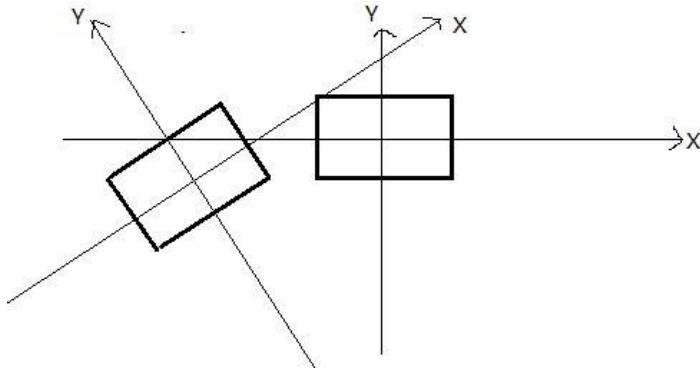
对所有点计算三维的协方差矩阵：

$$A = \begin{bmatrix} cov(x, x) & cov(x, y) & cov(x, z) \\ cov(x, y) & cov(y, y) & cov(y, z) \\ cov(x, z) & cov(y, z) & cov(z, z) \end{bmatrix}$$

主对角线的元素表示变量的方差，如果很大则表示强信号。较大的非主对角线元素表示数据的畸变。

得到协方差矩阵后利用雅克比方法求出特征向量。这些向量就是 **obb** 包围盒的坐标轴。最后对这些特征向量做施密特正交化处理，保证坐标轴之间相互垂直。到这一步，**obb** 的包围盒坐标轴就产生好了。下一步定义中心为所有点取平均。找到每个坐标轴方向上最远的点，这些点就是 **obb** 包围盒每个边的半长。

碰撞检测的时候，两个物体的 **obb** 包围盒都需要提前生成。拿到两个包围盒的有关信息后，将两个包围盒在 6 个坐标轴上，每个坐标轴上都做投影，如果每个轴投影都重叠，则说明发生碰撞。我们以二维模型为例：



这里两个二维的包围盒，将两个包围盒在四个轴上做投影，我们可以直观的看到，如果所有的轴上投影都是重叠的，那么这两个包围盒就相交了，如果只有某几个轴，或者没有轴重叠，则说明两个包围盒没有碰撞。

投影计算很简单，只需要将坐标轴归一化，之后与包围盒顶点点乘就可以得到投影长度了，计算为负则投影在负半轴上。

导弹有 3000 个点，飞机比它要多。飞机和导弹每次运动都需要生成 **obb** 包围盒用于碰撞检测。这样计算量过大，很影响系统的运行速度。为了简化计算，我们在 **3dmax** 中找到飞机和导弹包围盒，记录下 8 个顶点（飞机可以用多个包围盒逼近）。这些顶点在飞机和导弹运动的时候，将坐标乘上他们的变换矩阵就可以随着他们一起运动。每次生成包围盒的时候，只用这 8 个点（飞机要更多），计算 **obb** 包围盒就可以了。

对城市中楼的处理也和这个类似，在软件中找到楼的 8 个点，根据这 8 个点生成 **obb** 包围盒。

2.12. 波浪效果

我们的波浪效果是基于网格绘制的，通过物理模拟波动方程的方法来计算网格各个点的位置，达到动态的效果。在物理学中，我们知道二维波动方程为：

$$\frac{\partial^2 z}{\partial t^2} = c^2 \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right)$$

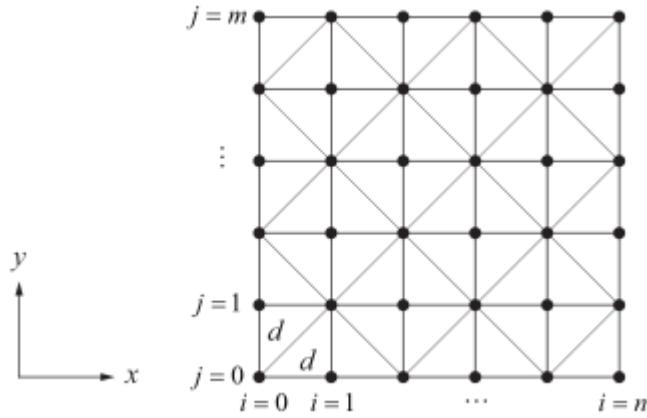
考虑到水面波的平均振幅应该会有一个阻尼衰减，添加一个阻尼系数，得到水面波的方程：

$$\frac{\partial^2 z}{\partial t^2} = c^2 \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right) - \mu \frac{\partial z}{\partial t},$$

其中，非负的常量 μ 表达了流体的黏度。

考虑黏性阻力的二维波方程可以通过分离变量法解出。但是这个解的形式非常复杂，对于实时模拟而言，它的计算量太大了。因此，我们选择用数值分析的方法来模拟流体表面波的运动。

假设我们的流体表面是由均匀分布在 $n * m$ 规则网格上的三角形组成的，如下图所示。设邻接点在 x 和 y 方向上的距离均为 d ，令连续的流体状态计算的时间间隔为 t 。我们用 $z(i, j, k)$ 来表达网格上的一个顶点，其中 i, j 是满足 $0 \leq i < n$ 和 $0 \leq j < m$ 的整数，它们表达的是空间坐标点。 k 是一个表达临时坐标值得非负整数。也就是说， $z(i, j, k)$ 等价于时间 kt 时，在坐标点 (id, jd) 处的顶点位移。



则有以下近似的一阶导数：

$$\begin{aligned}\frac{\partial}{\partial x} z(i, j, k) &= \frac{\frac{z(i, j, k) - z(i-1, j, k)}{d} + \frac{z(i+1, j, k) - z(i, j, k)}{d}}{2} \\ &= \frac{z(i+1, j, k) - z(i-1, j, k)}{2d}\end{aligned}$$

$$\frac{\partial}{\partial y} z(i, j, k) = \frac{z(i, j+1, k) - z(i, j-1, k)}{2d}.$$

$$\frac{\partial}{\partial t} z(i, j, k) = \frac{z(i, j, k+1) - z(i, j, k-1)}{2t}.$$

而对于二阶导数，可以用一阶导数再次进行差分得到。

$$\Delta \left[\frac{\partial}{\partial x} z(i, j, k) \right] = \frac{\frac{\partial}{\partial x} z(i+1, j, k) - \frac{\partial}{\partial x} z(i-1, j, k)}{2}.$$

$$\begin{aligned}\Delta \left[\frac{\partial}{\partial x} z(i, j, k) \right] &= \frac{\frac{z(i+2, j, k) - z(i, j, k)}{2d} - \frac{z(i, j, k) - z(i-2, j, k)}{2d}}{2} \\ &= \frac{z(i+2, j, k) - 2z(i, j, k) + z(i-2, j, k)}{4d}\end{aligned}$$

则有二阶导数：

$$\frac{\partial^2}{\partial x^2} z(i, j, k) = \frac{z(i+2, j, k) - 2z(i, j, k) + z(i-2, j, k)}{4d^2}.$$

但是这个公式用到了与当前点相隔 $2d$ 的点的位移，我们对其进行放缩，得

$$\frac{\partial^2}{\partial x^2} z(i, j, k) = \frac{z(i+1, j, k) - 2z(i, j, k) + z(i-1, j, k)}{d^2}$$

类似地，有：

$$\frac{\partial^2}{\partial y^2} z(i, j, k) = \frac{z(i, j+1, k) - 2z(i, j, k) + z(i, j-1, k)}{d^2}$$

$$\frac{\partial^2}{\partial t^2} z(i, j, k) = \frac{z(i, j, k+1) - 2z(i, j, k) + z(i, j, k-1)}{t^2}$$

于是，我们得到了水面波动方程的离散近似形式：

$$\begin{aligned} \frac{z(i, j, k+1) - 2z(i, j, k) + z(i, j, k-1)}{t^2} = \\ c^2 \frac{z(i+1, j, k) - 2z(i, j, k) + z(i-1, j, k)}{d^2} \\ + c^2 \frac{z(i, j+1, k) - 2z(i, j, k) + z(i, j-1, k)}{d^2} \\ - \mu \frac{z(i, j, k+1) - z(i, j, k-1)}{2t} \end{aligned}$$

我们想要决定在时间 t 后的下一个位移 $z(i, j, k+1)$ ，如果我们已经知道当前位移 $z(i, j, k)$ 以及前一次位移 $z(i, j, k-1)$ 。 $z(i, j, k+1)$ 的方程为：

$$\begin{aligned} z(i, j, k+1) = & \frac{4 - 8c^2 t^2 / d^2}{\mu t + 2} z(i, j, k) + \frac{\mu t - 2}{\mu t + 2} z(i, j, k-1) \\ & + \frac{2c^2 t^2 / d^2}{\mu t + 2} [z(i+1, j, k) + z(i-1, j, k) + z(i, j+1, k) + z(i, j-1, k)], \end{aligned}$$

这个公式的每一个系数都可以提前计算，以减少计算量。

此外，如果波动速率 c 太快，或者时间间隔 t 太长，那么我们的迭代方程将会发散到无穷。为了保证位移是有限的，我们需要决定一个确切的坐标值，使得方程在这个值之下保持稳定。为了保证收敛，我们要求从水平表面离开的顶点应当沿着释放时的表面移动。

假设我们有一个 $n * m$ 的顶点数组，除了坐标为 (i_0, j_0) 的顶点，其余顶点都有 $z(i, j, 0) = 0$ 以及 $z(i, j, 1) = 0$ 。我们令 (i_0, j_0) 处的点处在某一位置，使得 $z(i_0, j_0, 0) = h$ 并且 $z(i_0, j_0, 1) = h$ ，其中 h 是一个非 0 的位移。现在假设在 (i_0, j_0) 的点在时间 $2t$ 时释放，那么计算 $z(i_0, j_0, 2)$ 的值时，方程的第三项为 0，因而我们有：

$$\begin{aligned} z(i_0, j_0, 2) &= \frac{4 - 8c^2 t^2 / d^2}{\mu t + 2} z(i_0, j_0, 1) + \frac{\mu t - 2}{\mu t + 2} z(i_0, j_0, 0) \\ &= \frac{2 - 8c^2 t^2 / d^2 + \mu t}{\mu t + 2} h. \end{aligned}$$

对于沿着水平表面移动的顶点，它在 $2t$ 时的位移一定比在 t 时的位移要小，因此，我们有

$$|z(i_0, j_0, 2)| < |z(i_0, j_0, 1)| = |h|.$$

把 $z(i_0, j_0, 2)$ 的值代入，我们得到

$$\left| \frac{2 - 8c^2 t^2 / d^2 + \mu t}{\mu t + 2} h \right| < |h|.$$

所以：

$$-1 < \frac{2 - 8c^2 t^2 / d^2 + \mu t}{\mu t + 2} < 1.$$

分离 c ，我们得到

$$0 < c < \frac{d}{2t} \sqrt{\mu t + 2}.$$

这告诉我们对于任意邻接点间给定距离 d 以及方程迭代的任意连续时间间隔 t ，波的速度 c 一定比这个方程现实的最大值要小。

或者说，我们可以在给定距离 d 和波速 c 的情况下算出最大时间间隔 t 。然后方程两边同乘 $-(\mu t + 2)$ 并化简：

$$0 < \frac{4c^2}{d^2} t^2 < \mu t + 2.$$

左边的不等式仅仅要求 $t > 0$ ，这是一个自然成立的条件。而右边的不等式可以转化为：

$$\frac{4c^2}{d^2} t^2 - \mu t - 2 < 0.$$

使用二次方程，这个多项式的根为：

$$t = \frac{\mu \pm \sqrt{\mu^2 + 32c^2/d^2}}{8c^2/d^2}.$$

由于方程中的二次项系数为正，所以对应的抛物线开口 **1** 向上，因此当 t 分布在根的两边时，多项式是负的。又因为根中根号里的式子大于 μ ，所以两个根中较小的那个是复数，所以可以被舍去。我们现在可以这样表达时间 t 的取值范围：

$$0 < t < \frac{\mu + \sqrt{\mu^2 + 32c^2/d^2}}{8c^2/d^2}.$$

$$0 < c < \frac{d}{2t} \sqrt{\mu t + 2}.$$

使用处在 范围外的波速 c ，或者使用处在落在

$$0 < t < \frac{\mu + \sqrt{\mu^2 + 32c^2/d^2}}{8c^2/d^2}.$$

范围外的时间，都会导致顶点位移的指数爆炸增长。

因此，在绘制时，需要对 t 和 c 进行控制。

在实现时，我们需要存储两个缓冲区，每个都包含 $n*m$ 个顶点位置信息。在每一帧，其中一个缓冲区包含了当前的顶点位置，另一个缓冲区包含了前一个顶点位置。当我们计算新的位移时，对于每个顶点，我们用包含新顶点位置的缓冲区替代旧顶点位置的缓冲区。包含当前顶点信息的缓冲区就转换成了包含之前顶点位置的缓冲区。所以我们事实上交换的是当前用于渲染帧的缓冲区。

最后，找到一张睡眠的贴图，贴在这个网格上就能形成水面的波动效果。

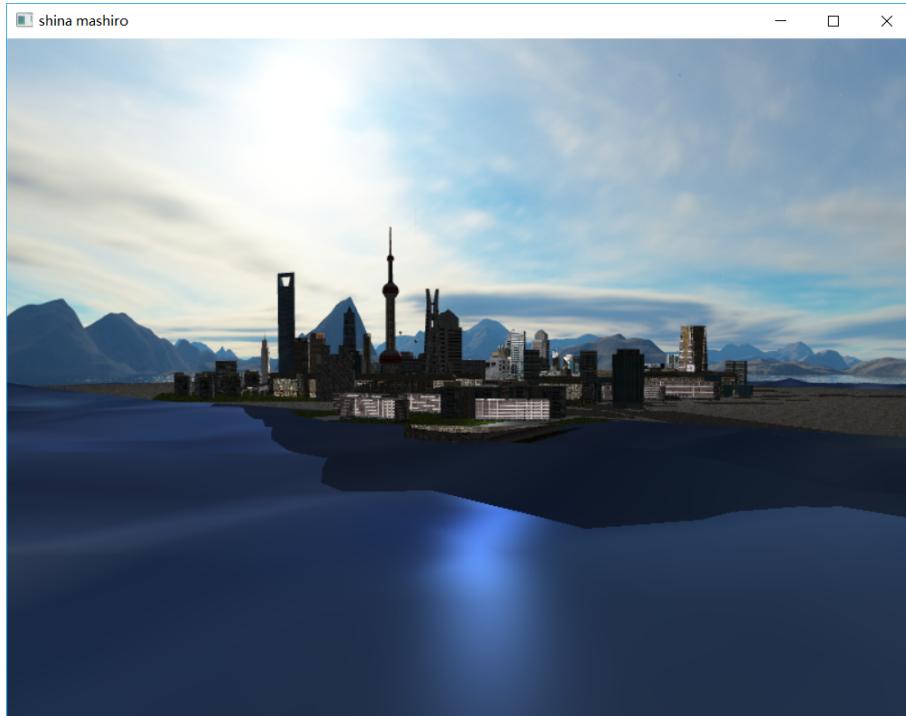


Figure 6 波浪效果

2.13. 粒子系统

粒子系统是一个物理模拟的过程，我们定义了两个类 `cloud` 和 `particle` 用来表示粒子系统。

一个粒子系统为一个类 `cloud`,一个 `cloud` 类中定义了粒子的 `vector`:
`vector<particle>`, 最大个数, 最大显示帧数和粒子初始化函数 `initparticle` (这是一个函数指针, 通过指向写的函数, 可以达到自定义的效果)。

`particle` 是一个粒子, 其中定义了这个粒子的初始位置, 速度, 加速度, 旋转角度, 角速度, 角加速度, 生命周期和衰减速度等变量。还记录了每个粒子的初始化次数, 通过限制该变量, 可以使粒子的初始化停止, 从而使我们要的效果逐渐消失。

我们的粒子类中可以选择粒子的形态, 可以是一个 `obj` 的或者是一个矩形, 分别用来显示爆炸效果 (粒子为一个 `obj`) 和火焰效果 (粒子为一个矩形)。因此 `particle` 类里面有 `obj` 的指针和矩形的指针, 我们用 `is_which` 变量判断绘制的模式。

在每次画面更新时，`cloud` 类对其中的所有 `particle` 调用 `show` 函数，而 `particle` 里面会根据 `is_which` 来调用 `obj` 的 `show` 函数或者矩形的 `show` 函数进行显示。

在调用 `show` 函数进行绘制之前，我们会先对物体的运动状态进行更新，这个更新由 `update` 函数完成，在更新完成后我们会检查函数的生命周期，当函数的生命周期衰减为 0 时，我们会再次调用 `initParticle` 函数对其进行初始化。这样就可以产生一个循环的效果。

此外，`cloud` 类中还有一个 `time` 变量，用来表示效果的最大显示帧数。每次 `cloud` 的 `show` 函数被调用时，`time` 会自减，当 `time` 为 0 时，不再显示所有粒子。

1) 爆炸系统

对于爆炸效果的实现，我使用了上面写的粒子系统。为了对一个物体进行爆炸，首先需要定义一个 `tex2cloud` 函数，这个函数根据材质信息将 `texture` 类进行拆分，每一个 `group` 作为一个 `texture`，放到 `cloud` 类的 `vector<particle>` 中，并且对碎片的 `hide` 信息设置为 `true`，这个碎片的 `died` 属性设置为 `false`。`tex2cloud` 函数在场景初始化的时候被调用，预先把所有信息存储好，以防止在爆炸的时候对信息的复制造成卡顿。

当碰撞检测系统检测到物体进行了碰撞，则调用一个 `hit` 函数，这个函数会将这个 `cloud` 中的所有碎片的 `hide` 改为 `false`，`died` 属性设置为 `false`，使碎片可见并且在下一次计算的时候被初始化，同时将原物体的 `hide` 属性设置为 `true`，使爆炸的物体消失。这个 `hit` 函数只能被调用一次，否则会出现多次初始化的情况。

在下一帧中，物体被初始化，物体的所有零件的起始坐标都被设置在爆炸中心点。而速度，加速度，旋转角度，角速度，角加速度和碎片的生命周期，衰减周期被随机赋值（在一个固定的区间）。这样当零件数量足够多的时候，从远处看上去就像是从这个爆炸中心四散开来，达到爆炸效果。并且每个粒子的最大初始化次数设置为 1，这样所有碎片都只会被初始化一次。

当所有的碎片的生命周期结束后，将该类清理掉，以后不再更新

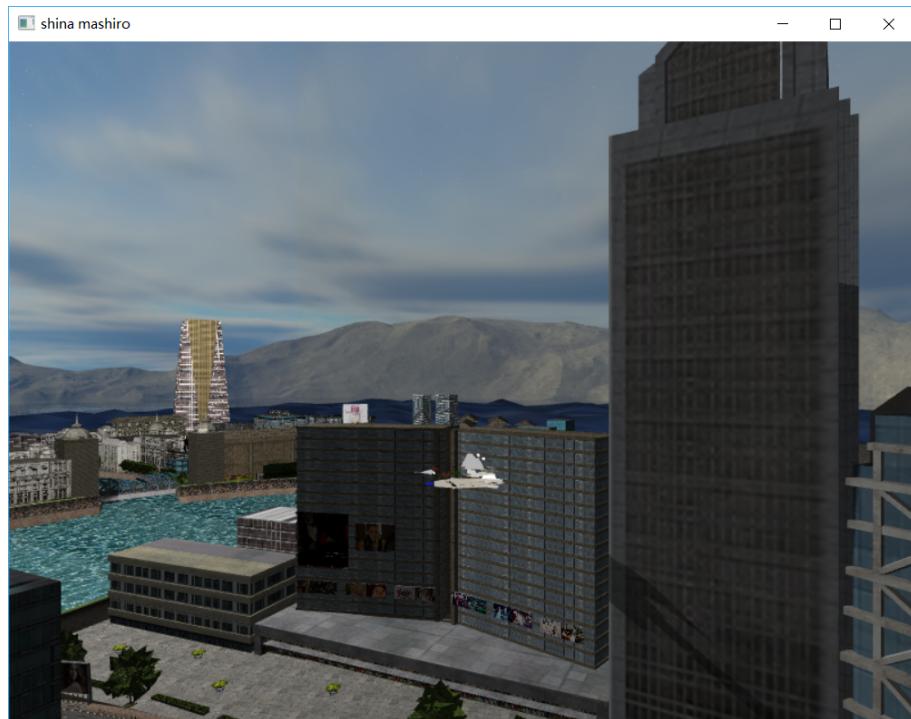


Figure 7 飞机爆炸

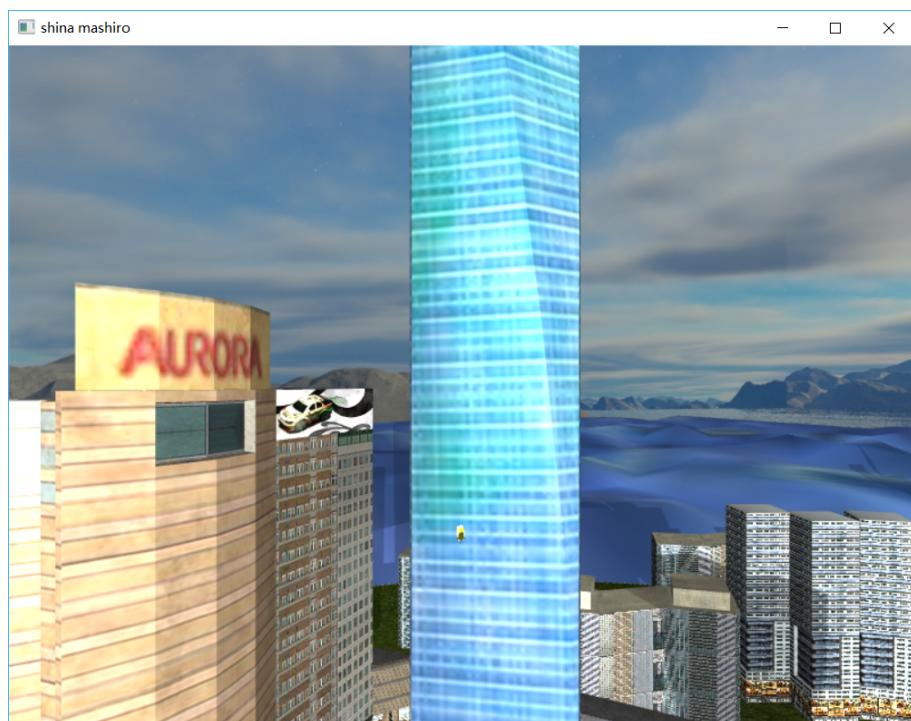


Figure 8 导弹爆炸 1

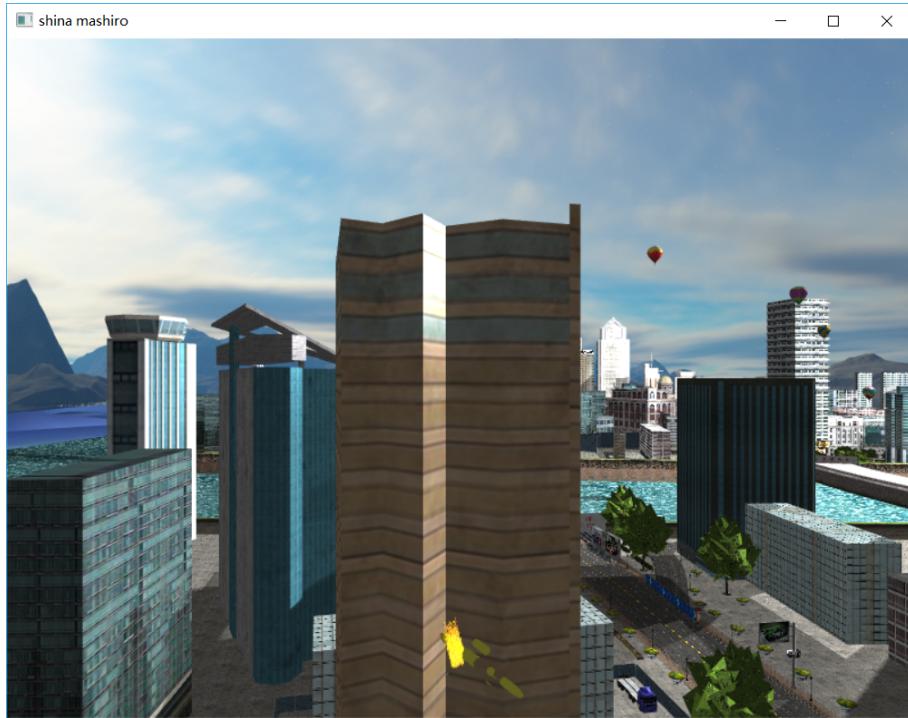


Figure 9 导弹爆炸 2

2) 火焰系统

火焰是我们特别优化过的粒子系统。优化的地方有以下：

1. 所有的粒子都是平面片，CPU 中只存储计算平面片中心点位置，其四个点是在几何着色器中完成。
2. 所有的粒子都只做平移（为了减少计算量，但效果其实是看不出来的，因为离子数非常多）
3. 所有的粒子一次性绑定，一次性送到 GPU，一帧渲染一次。
4. 优化随机数，使用随机数池预处理，然后随机下标就可以获得任意多的随机数。

a) 初始化火焰粒子

初始化火焰粒子需要知道初始位置，火焰发射的法向。因此我需要将一个 `model` 矩阵保存。

将初始位置加上一个随机的偏移量，就可以得到在原始坐标轴上的 $xyzw$ ，然后左乘变换矩阵就可以转移到世界坐标系。

```
glm::vec4 firePosition = (*model)*glm::vec4(*fireX + randNumber[random] * radius
- radius/2, *fireY + randNumber[(random * 2) % 10000] * radius - radius/2, *fireZ,
1.f);
```

火焰发射的方向也是做同样的事。将向量的初始点和结束点做一个 `model` 变换，然后相减单位化即可。

火焰的速度发射方向随机乘一个数生成的。

火焰还有颜色，越接近中心就越白，远离就会变黄。因此将其颜色预先定为白色。

```
glm::vec4 fireVec = (*model)*flyVec;
glm::vec4 yuan = (*model)*glm::vec4(0, 0, 0, 1);
fireVec = yuan - fireVec;
fireVec /= sqrt(fireVec.x*fireVec.x + fireVec.y*fireVec.y + fireVec.z*fireVec.z);
fireVec *= (randNumber[(random * 4) % 10000] * 1);
```

b) 更新

火焰速度随着时间叠加即可，和普通的粒子系统一样。

火焰的颜色需要另外处理。我的方法是根据时间，做一个初始颜色和消去颜色的线性插值，和纹理颜色做一个混合（在 `shader` 中完成）。

如果火焰灭亡，就重新开始。

```
pos[i * 7] += particles[i].vx;
pos[i * 7 + 1] += particles[i].vy;
pos[i * 7 + 2] += particles[i].vz;
pos[i * 7 + 3] = interpolate<float>(particles[i].lifeTime /
particles[i].fullLife, initialColor.r, fadeColor.r);
pos[i * 7 + 4] = interpolate<float>(particles[i].lifeTime /
```

```

particles[i].fullLife, initialColor.g, fadeColor.g);

    pos[i * 7 + 5] = interpolate<float>(particles[i].lifeTime /
particles[i].fullLife, initialColor.b, fadeColor.b);

    pos[i * 7 + 6] -= particles[i].decrease;

    particles[i].lifeTime -= particles[i].decrease;

    if (particles[i].lifeTime < 0 || pos[i*7+6]<=0)

    {

        initFireParticle(i);

    }

```

Shader 中：

几何着色器：

几何着色器收到顶点着色器的输出，会重新生成四个点的坐标和相应的火焰贴图纹理坐标，传递给片段着色器。

```

#version 330 core

layout (points) in;

layout (triangle_strip, max_vertices = 4) out;

in VS_OUT {
    vec4 color;
} gs_in[];

out vec4 fColor;
out vec2 texcoord;

uniform mat4 u_projection;
uniform mat4 u_view;

void build_house(vec4 position)
{

```

```

fColor = gs_in[0].color; // gs_in[0] since there's only one input vertex

gl_Position = u_projection * u_view *(position + vec4(-0.11f, -0.11f, 0.0f, 0.0f));    //

1:bottom-left

texcoord=vec2(0,0);

EmitVertex();

gl_Position = u_projection * u_view *(position + vec4( 0.11f, -0.11f, 0.0f, 0.0f));    //

2:bottom-right

texcoord=vec2(1,0);

EmitVertex();

gl_Position = u_projection * u_view *(position + vec4(-0.11f,  0.11f, 0.0f, 0.0f));    // 3:top-left

texcoord=vec2(0,1);

EmitVertex();

gl_Position = u_projection * u_view *(position + vec4( 0.11f,  0.11f, 0.0f, 0.0f));    //

4:top-right

texcoord=vec2(1,1);

EmitVertex();

EndPrimitive();

}

void main() {

    build_house(gl_in[0].gl_Position);

}

```

片段着色器：

片段着色器会收到顶点和纹理坐标和颜色，做一个简单的乘法作为混合。

```
color =texture2D(u_textureMap,texcoord)*fColor;
```

这样火焰的性能就会非常好，效果也非常棒。经测试，这个粒子系统在 10 左右也不会卡顿，优化之前内存非常浪费而且 3k 个就已经非常卡了。这个事情也同样说明了灵活的可编程管线的强大之处。

3. 流程展示图片

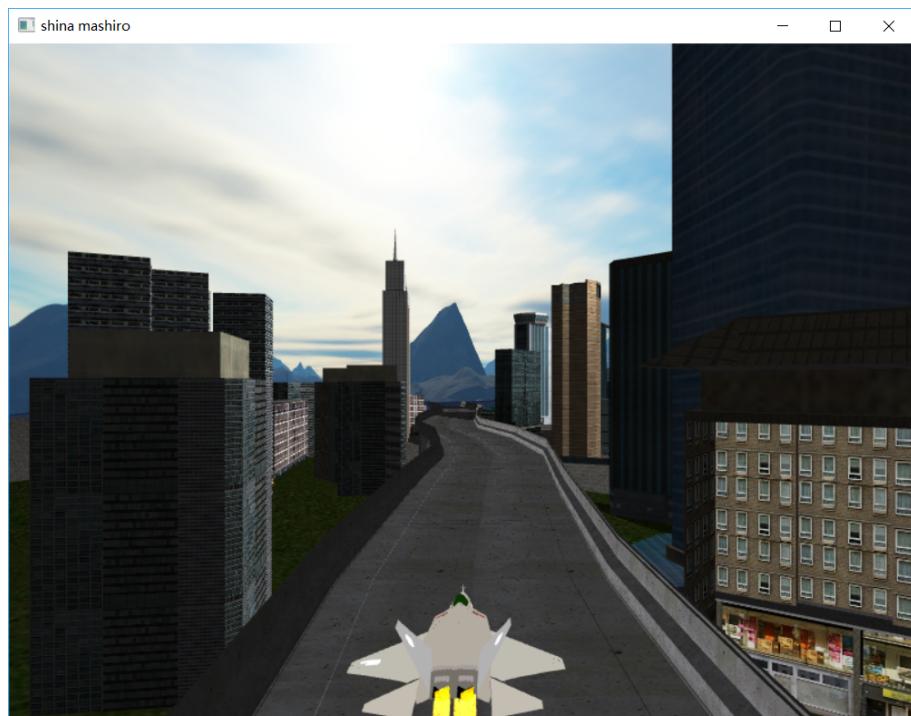


Figure 10 飞机起飞

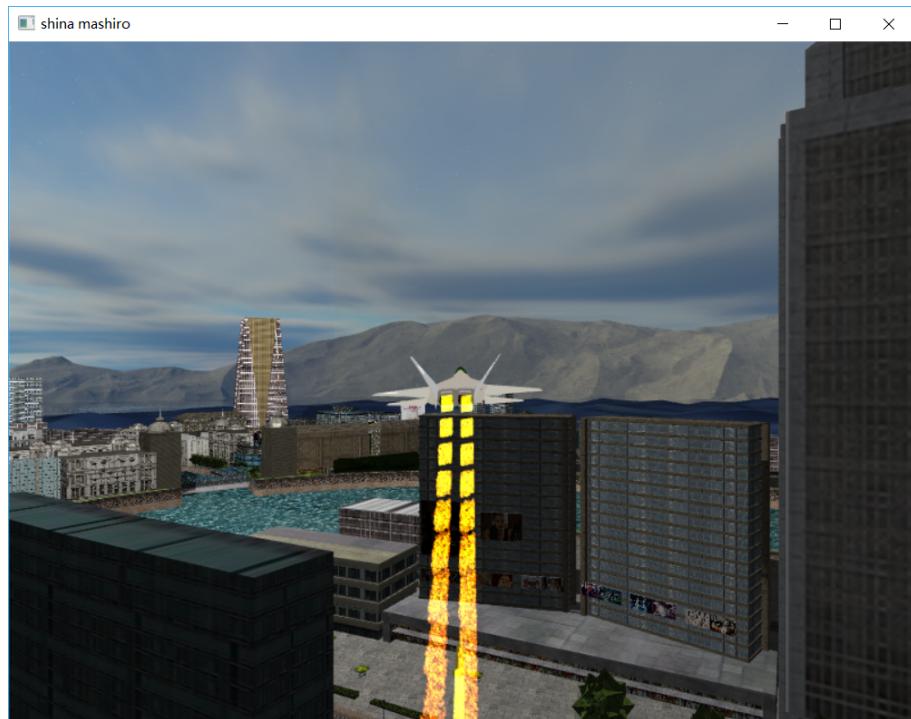


Figure 11 飞机发射导弹

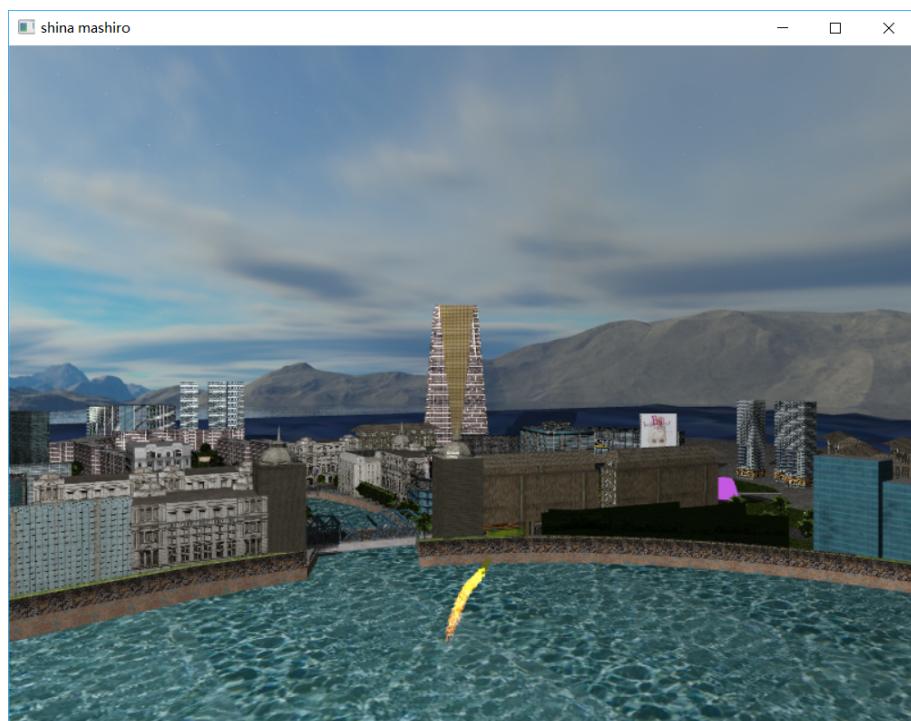


Figure 12 导弹飞行

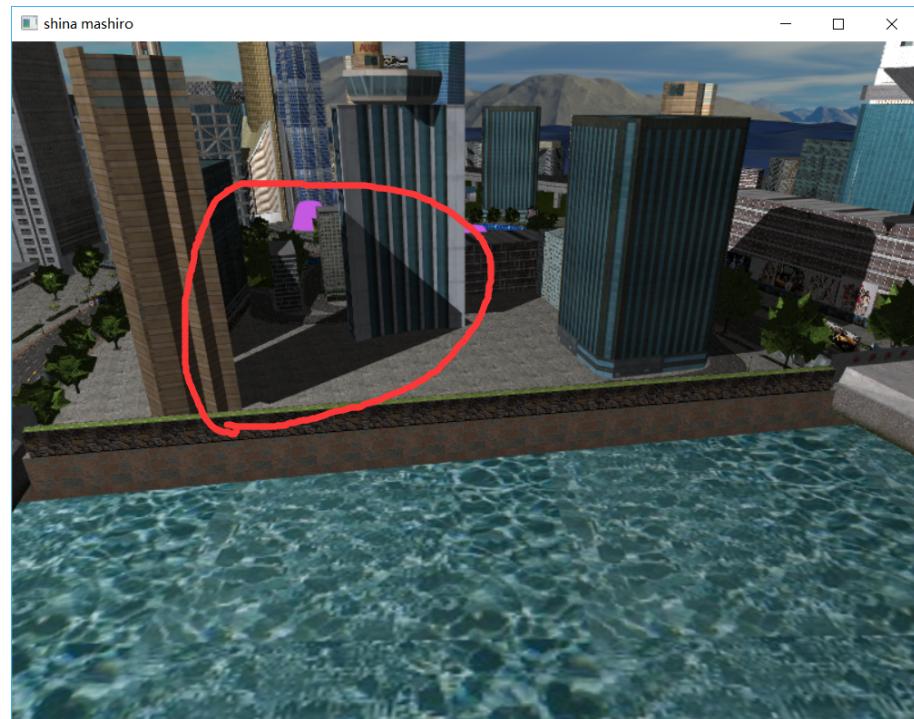


Figure 13 实时阴影

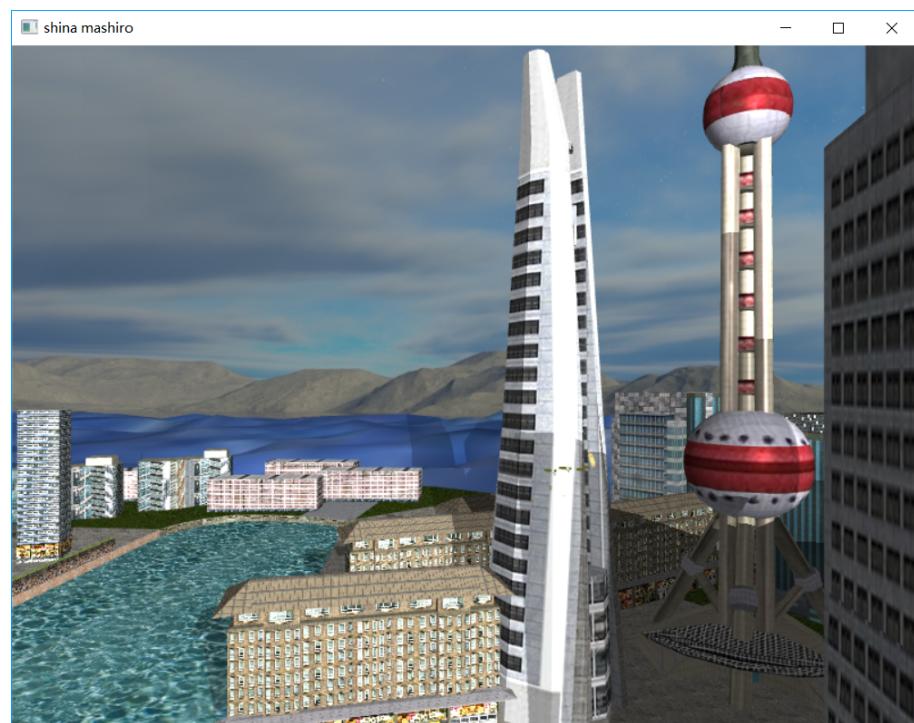


Figure 14 爆炸时的多光源

4. 感想

在此次的图形学大程编写过程中，我们遇到了不计其数的困难，因为使用大规模的模型，同时功能复杂，所以无论是在难度还是工作量方面都非常大。每个小组成员都付出了很多心血，最终构成了这个上万行代码，数百 M 大小的工程。

感谢老师和助教在一个学期中的辛勤付出！我们在图形学的课堂和实践中掌握了很多知识，也将带着这份兴趣和热爱投入接下来的学习中。