

计算机图形学——单人FPS游戏

组员：

蒋仕彪——3170102587

刘明锐——3170105696

罗炜程——3170105902

王宇晗——3170106051

提交日期：2019-06-25

一、概述

本次我们的图形学大作业做的是一个基于现代OpenGL的单人FPS游戏 **pumpkinBattle**。以下是我们达到的要点：

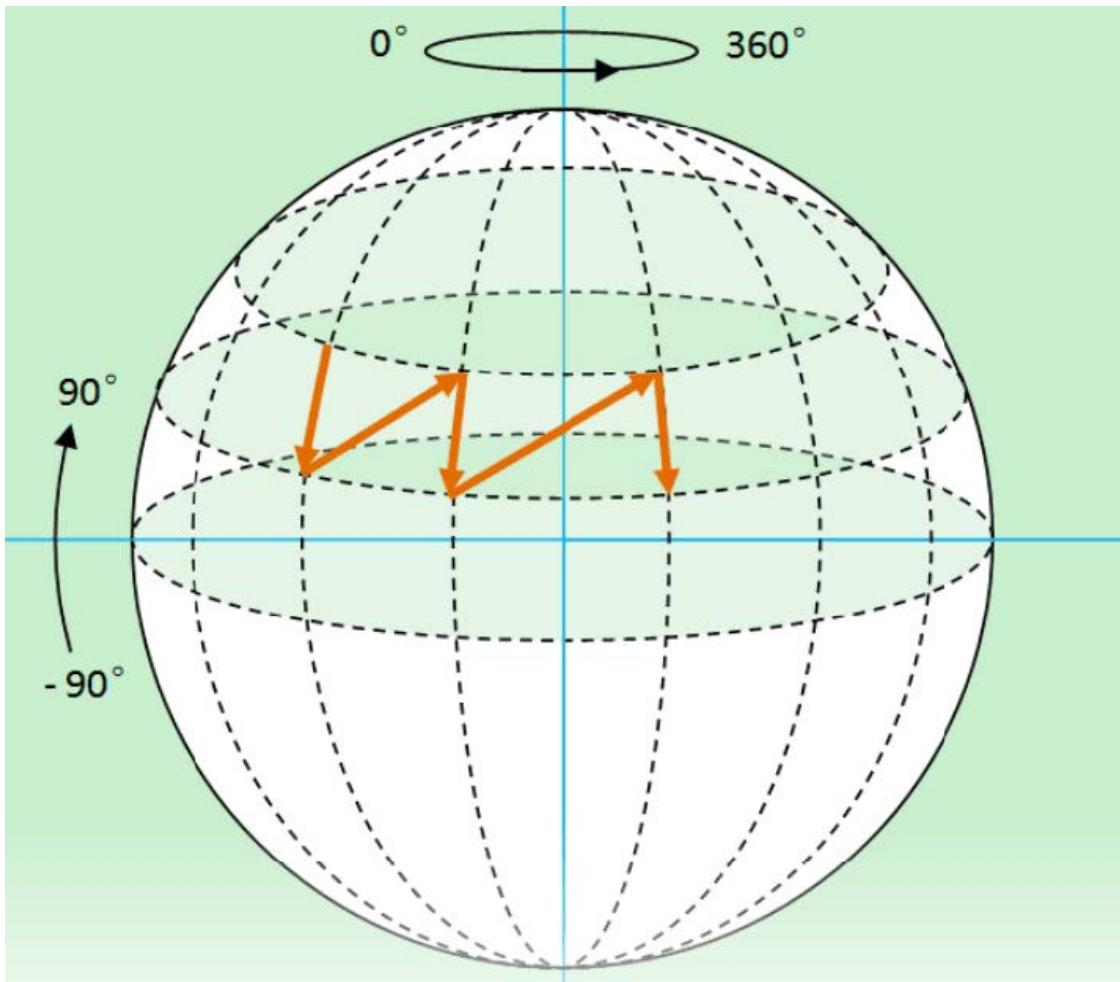
- 基本要求
 - 基本体素表达
 - 场景中间的球、圆柱、圆锥
 - obj的导入
 - 静态位置（星球）
 - 实时位置（手榴弹瞄准）
 - 动态位置（子弹的飞行）
 - 纹理
 - 制造（中间的体素）
 - 贴图（墙和天空盒）
 - 几何变换
 - 南瓜与人距离越近的时候，越会面向人走过来（旋转、平移）
 - 对场景里的物体有一定的平移、缩放
 - 光照
 - 实现了点光源
 - 实现了点光源随时间的动态变化
 - 漫游
 - 按鼠标和 WASD 进行移动
 - 空格进行跳跃
 - 按回车复活
 - 动画和截屏
 - 胜利后升天的动画
 - 按 P 截屏
- 高级要求
 - 实时检测的碰撞系统
 - 子弹碰到墙面反射
 - 子弹碰到图形反射
 - 人碰到墙/图形被阻挡
 - 人打破墙会越过墙。
 - 阴影
 - 实时根据场景物体产生深度图
 - 实时阴影体现

- 游戏性
 - 可以用手榴弹炸墙，来穿过去躲避南瓜
 - 可以对南瓜开枪来打死它
 - 显示剩余南瓜数量
 - 击败所有南瓜时胜利，打出 you win 并有升天效果
 - 可以按回车复活
- 对象表达能力（墙）
 - 墙的破坏（右键）
 - 人遇到墙会阻挡
 - 跳过墙或穿过墙
 - 墙的内外贴图
 - 子弹遇到墙的反弹
- 一些小特色
 - 跳跃和重力系统
 - 空格键跳跃，回车键复活
 - 可以跳着过墙
 - 打中南瓜后，南瓜会变红一会
 - 子弹多次打中南瓜，南瓜会消失，文字中显示的南瓜数量减一

二、技术要点

1. 基本体素的表达与绘制流程

- 除了墙本身使用立方体建模之外，游戏中还实现了圆、圆柱和圆锥三种基本体素，放置在场景中心。
- 体素表达
 - 圆
 - 表达的本质是将图形拆成三角形面来表示，分配好点坐标、法向量和贴图坐标后传输给绘制模块。
 - 如图所示，通过在经度和纬度方向上间隔一定角度采样，按照图中的方式将点相连就可以得到三角形。
 - 贴图时，将图中割出的方形小块贴上一个正方形贴图即可。



```

1  const int sphereprec = 15;
2  const double PI = acos(-1);
3  void generatesphere(vector<pair<pair<Point3D, Point3D>, Point2D>>
4    &vec) {
5      Point3D tmp[sphereprec + 1][sphereprec];
6      for (int i = 0; i <= sphereprec; ++i) {
7          double the = PI * i / sphereprec - PI / 2;
8          for (int j = 0; j < sphereprec; ++j) {
9              double alp = 2 * PI * j / sphereprec;
10             tmp[i][j] = Point3D(cos(the)*cos(alp), sin(the),
11               cos(the)*sin(alp));
12             if (i == 0 || i == sphereprec) tmp[i][j].x = tmp[i]
13               [j].z = 0;
14           }
15       }
16       for (int i = 1; i < sphereprec; ++i) {
17           for (int j = 0; j < sphereprec; ++j) {
18               int k = (j + 1) % sphereprec;
19
20               #define vebp(i,j,x,y) vec.emplace_back(make_pair(make_pair(tmp[i]
21                 [j],tmp[i][j]),Point2D(x,y)))
22               vebp(i, k, 1, 1); vebp(i, j, 0, 1); vebp(i - 1, j, 0, 0);
23               vebp(i, j, 0, 0); vebp(i, k, 1, 0); vebp(i + 1, k, 1, 1);
24           }
25       }
26   }

```

- 圆柱

- 类似于圆，圆柱只需要将 y 方向（经度方向）上的圆弧变化变为不变的定值即可。注意最上侧和最下侧要强制设定为点，以保证是闭合图形。

```

1   for (int i = 0; i <= sphereprec; ++i) {
2       double hei = 2. * i / sphereprec - 1.;
3       for (int j = 0; j < sphereprec; ++j) {
4           double alp = 2 * PI * j / sphereprec;
5           tmp[i][j] = Point3D(cos(alp), hei, sin(alp));
6           if (i == 0 || i == sphereprec) tmp[i][j].x = tmp[i][j].z =
7               0;
8       }

```

○ 圆锥

- 同样地，圆锥只要将 y 方向（经度方向）上的变化改为线性变化即可，最下侧需要保证为点。

```

1   for (int i = 0; i <= sphereprec; ++i) {
2       double hei = 2. * i / sphereprec - 1., rad = 1 - 1. * (i + 1)
3       / sphereprec;
4       for (int j = 0; j < sphereprec; ++j) {
5           double alp = 2 * PI * j / sphereprec;
6           tmp[i][j] = Point3D(rad * cos(alp), hei, rad * sin(alp));
7           if (i == 0 || i == sphereprec) tmp[i][j].x = tmp[i][j].z =
8               0;
9       }
10      }

```

• 体素绘制

- 计算出三角形的位置之后，只需要和上述其他绘制一样，整理成对应格式即可。
- 对于贴图，我们选择了自己绘制的黄红相间和黄绿相间的两种贴图。

```

1   void renderTriangle(unsigned int &trivAO, unsigned int &trivBO,
2   vector<pair<pair<Point3D, Point3D>, Point2D>> &vec)
3   {
4       float *vertices = new float [vec.size() * DATASIZE];
5       for (int i = 0; i < vec.size(); i++)
6           vec[i].first.first.PushPoint(vec[i].first.second, vertices +
7           DATASIZE * i, vec[i].second);
8
9       if (trivAO == 0)
10      {
11          glGenVertexArrays(1, &trivAO);
12          glGenBuffers(1, &trivBO);
13          // fill buffer
14          glBindBuffer(GL_ARRAY_BUFFER, trivBO);
15          glBufferData(GL_ARRAY_BUFFER, vec.size() * DATASIZE *
16          sizeof(float), vertices, GL_STATIC_DRAW);
17          // link vertex attributes
18          glBindVertexArray(trivAO);

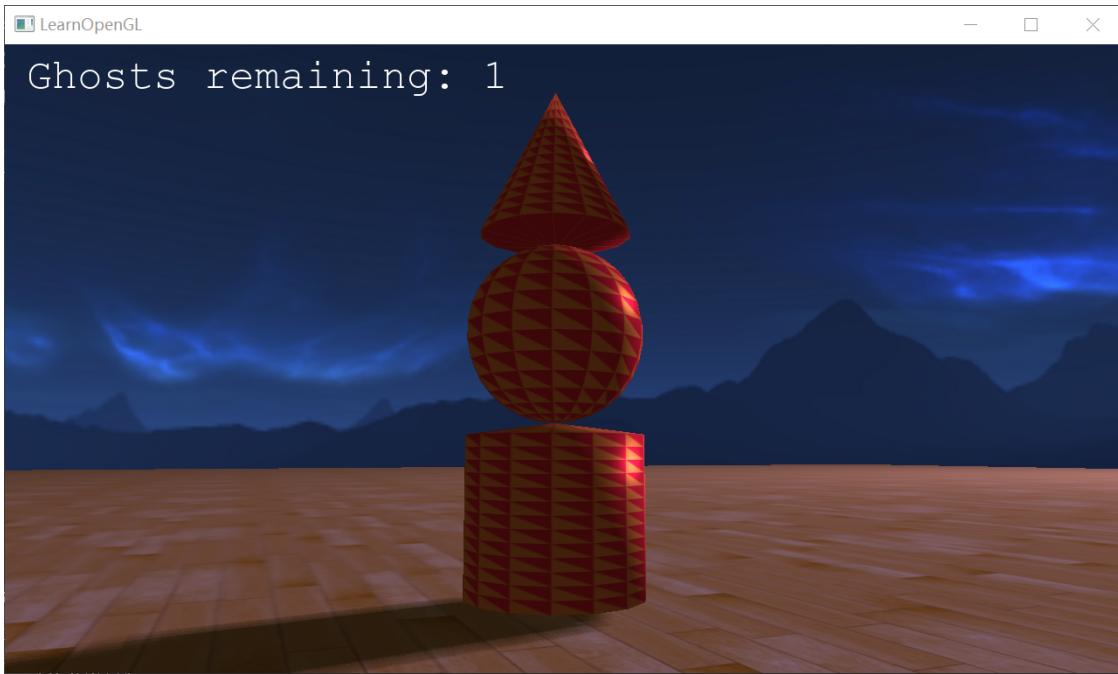
```

```

19         glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 * sizeof(float)));
20         glEnableVertexAttribArray(2);
21         glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
22         glBindBuffer(GL_ARRAY_BUFFER, 0);
23         glBindVertexArray(0);
24     }
25
26     glBindBuffer(GL_ARRAY_BUFFER, trivBO);
27     glBufferData(GL_ARRAY_BUFFER, vec.size() * DATASIZE * sizeof(float), vertices, GL_STATIC_DRAW);
28
29     glBindVertexArray(trivAO);
30     glDrawArrays(GL_TRIANGLES, 0, vec.size());
31     glBindVertexArray(0);
32
33     delete[] vertices;
34 }
```

- 最终效果

- 我们最终选择在地图中心放置这三个体素，按O键切换贴图。



2. OBJ的导入绘制与封装流程

- 概况和环境配置
 - 导入 `obj` 的时候，我们用到一个非常流行的模型导入库是 `Assimp` 库，它是 Open Asset Import Library（开放的资产导入库）的缩写。
 - 当使用 `Assimp` 导入一个模型的时候，它通常会将整个模型加载进一个场景对象 `Scene`。
 - 一个模型一般分为 `.obj` 和 `.mtl` 两个文件（当然还包括一些必要的图片文件）。`.obj` 用来描述顶点和面，而 `.mtl` 是纹理配置文件。
 - 我们需要做的第一件事是将一个物体加载到 `Scene` 对象中，遍历节点，获取对应的 `Mesh` 对象（我们需要递归搜索每个节点的子节点），并处理每个 `Mesh` 对象来获取顶点数据、索引以及它的材质属性。最终的结果是一系列的网格数据，我们会将它们包含在一个 `Model` 对象中。

- 一个很重要的类别是网格类 `Mesh`，它的定义如下：

```

1 class Mesh {
2     public:
3         vector<Vertex> vertices;
4         vector<unsigned int> indices;
5         vector<Texture> textures;
6         Mesh(vector<Vertex> vertices, vector<unsigned int>
7             indices, vector<Texture> textures);
8         void Draw(Shader shader);
9     private:
10        unsigned int VAO, VBO, EBO;
11        void setupMesh();
12    };

```

- 就像顶点缓冲数组和顶点索引数组一样，`Mesh` 类里也有顶点数组、索引数组，以及贴图的位置信息。

- 模型的封装

- 为了更好地封装模型，我们再引入一个 `model` 类。

```

1 class Model{
2     public:
3         Model(char *path){
4             loadModel(path);
5         }
6         void Draw(Shader shader);
7     private:
8         vector<Mesh> meshes;
9         string directory;
10        void loadModel(string path);
11        void processNode(aiNode *node, const aiScene *scene);
12        Mesh processMesh(aiMesh *mesh, const aiScene *scene);
13        vector<Texture> loadMaterialTextures(aiMaterial *mat,
14            aiTextureType type, string typeName);
15    };

```

- 我们预期的对外接口是：用 `loadModel()` 来初始化模型（从路径中导入），用 `draw()` 来绘制模型。外层会在给出的 `shader` 里定义模型的旋转量、摆放位置以及缩放程度。

- 模型的使用和继承

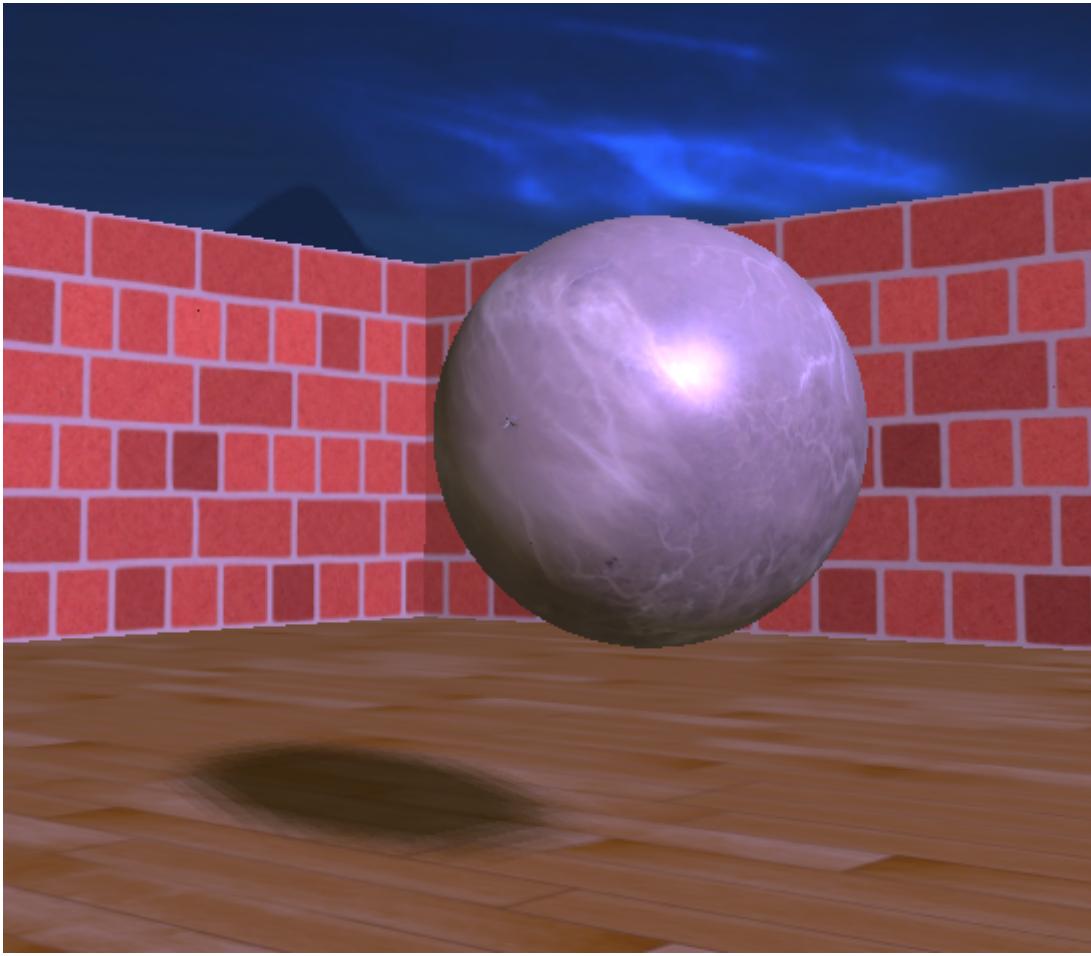
- 在官网下载 `assimp` 后，用 `cmake` 生成 `lib`，并加入以下代码。

```

1 #include <assimp/Importer.hpp>
2 #include <assimp/scene.h>
3 #include <assimp/postprocess.h>

```

- 为了更好地管理模型，我们在 `model.cpp` 里建立一个 `vector<Model *>`，用来管理所有模型。新建时，只需 `new` 一个新的 `model` 类并给出模型地址。
- 导入月球后效果如图：



- 现在考虑如何管理模型。可以在 `model` 类里增加一些 `vec3` 变量，表示模型目前的位置，模型的朝向等。绘制模型时如一下代码：

```
1 void drawModel(const shader &shader) {
2     for (auto it : ModelManager) {
3         it->setHit();
4         if (it->typ != 0 && it->checkHitPoint() == false) continue;
5         glm::mat4 model = glm::mat4(1.0f);
6         model = glm::translate(model, glm::vec3((it->pos).x, (it-
7             >pos).y, (it->pos).z));
8         Point3D origin = Point3D(0, 0, 1);
9         double theta = (it->Front % origin) / (it->Front).length();
10        Point3D axis = (it->Front) * origin;
11        if (sgn(axis.length()) != 0)
12            model = glm::rotate(model, (float)(-acos(theta)),
13                glm::vec3(axis.x, axis.y, axis.z));
14        model = glm::scale(model, glm::vec3(it->size, it->size, it-
>size));
15        shader.setMat4("model", model);
16    }
```

- 其中 `it->Front` 表示南瓜应该朝向的方向（在后面会有详细介绍），`it->size` 表示模型缩放大小，`it->pos` 表示模型当前位置。这里涉及到的点积和叉积，是为了将南瓜从默认朝向调整为我们想要的朝向。
- 但是，不同的模型可能还需要各种不同的功能，例如：
- 子弹模型需要对物体做反弹和射入的判定
- 南瓜需要进行血量判定
- 手榴弹需要实时判定（当用户视线中心在墙面上就显示）

- 为了更好地管理 `model` 类，我们还从 `model` 中派生出一些子类，并用动态绑定技术，用统一的基类指针管理。

3. 纹理贴图的实现方法

- 墙的贴图
 - 在最开始讨论的时候我们认为，一块一块砖去渲染贴图效率会比较低。我们相用 `KDtree`（四分树）来加速这个过程。具体的，我们每次判断当前的长方形是否是完整的：
 - 如果是，直接绘制这个长方形（其实是两个三角形）并退出
 - 将图形分裂成两部分或者四部分，并递归下去。
 - 但这里遗留一个问题：在破坏部分砖块后，我们必须要能看到别的砖块里面的颜色。**所以我们要针对每块砖去进行渲染，`KDtree`也就失去了优化的意义。
 - 所以我们最终采用的渲染方式是：先将墙拆成一个一个立方体吗，再进行贴图。我们先一个形式化地涂立方体的代码，每次用矩阵操作在 `shader` 里去限定它的位置和大小，再调用渲染函数。
 - 即使采取暴力渲染的方式，墙的贴图也会比普通的立方体要难很多，因为每个单位的砖块只是调用了一小块的贴图。我们需要具体地计算每个砖块所在的坐标。而且对于垂直的墙和竖直的墙，绘制的面显然是不一样的。
 - 综合以上的考量，绘制墙代码的函数变得比较笨重：

```

1     void renderCube(unsigned int &cubeVAO, unsigned int &cubeVBO,
2 Point3D texture, bool type = 0)
3     {
4         static float vertices[] = {};
5         //注意这里省略了细节，本来是一个正方体的标准贴图
6         if (cubeVAO == 0)
7         {
8             glGenVertexArrays(1, &cubeVAO);
9             glGenBuffers(1, &cubeVBO);
10            // fill buffer
11            glBindBuffer(GL_ARRAY_BUFFER, cubeVBO);
12            glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
13 GL_STATIC_DRAW);
14            // link vertex attributes
15            glBindVertexArray(cubeVAO);
16            glEnableVertexAttribArray(0);
17            glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 *
18 sizeof(float), (void*)0);
19            glEnableVertexAttribArray(1);
20            glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 *
21 sizeof(float), (void*)(3 * sizeof(float)));
22            glEnableVertexAttribArray(2);
23            glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 *
24 sizeof(float), (void*)(6 * sizeof(float)));
25            glBindBuffer(GL_ARRAY_BUFFER, 0);
26            glBindVertexArray(0);
27        }
28        for (int st = 0; st < 6; st++) {
29            vertices[st * 48 + 8 * 0 + 6] = vertices[st * 48 + 8 * 4
+ 6] = vertices[st * 48 + 8 * 5 + 6] = 0.05;
30            vertices[st * 48 + 8 * 0 + 7] = vertices[st * 48 + 8 * 1
+ 7] = vertices[st * 48 + 8 * 5 + 7] = 0.05;
31            vertices[st * 48 + 8 * 1 + 6] = vertices[st * 48 + 8 * 2
+ 6] = vertices[st * 48 + 8 * 3 + 6] = 0.06;
32        }
33    }
34 }
```

```

27         vertices[st * 48 + 8 * 2 + 7] = vertices[st * 48 + 8 * 3
+ 7] = vertices[st * 48 + 8 * 4 + 7] = 0.06;
28     }
29
30     if (type == 0) {
31         vertices[0 * 48 + 8 * 0 + 6] = vertices[0 * 48 + 8 * 4 +
6] = vertices[0 * 48 + 8 * 5 + 6] = texture.x;
32         vertices[0 * 48 + 8 * 0 + 7] = vertices[0 * 48 + 8 * 2 +
7] = vertices[0 * 48 + 8 * 4 + 7] = texture.y;
33         vertices[0 * 48 + 8 * 1 + 6] = vertices[0 * 48 + 8 * 2 +
6] = vertices[0 * 48 + 8 * 3 + 6] = texture.x + texture.z;
34         vertices[0 * 48 + 8 * 1 + 7] = vertices[0 * 48 + 8 * 3 +
7] = vertices[0 * 48 + 8 * 5 + 7] = texture.y + texture.z;
35
36         vertices[1 * 48 + 8 * 0 + 6] = vertices[1 * 48 + 8 * 4 +
6] = vertices[1 * 48 + 8 * 5 + 6] = texture.x;
37         vertices[1 * 48 + 8 * 0 + 7] = vertices[1 * 48 + 8 * 1 +
7] = vertices[1 * 48 + 8 * 5 + 7] = texture.y;
38         vertices[1 * 48 + 8 * 1 + 6] = vertices[1 * 48 + 8 * 2 +
6] = vertices[1 * 48 + 8 * 3 + 6] = texture.x + texture.z;
39         vertices[1 * 48 + 8 * 2 + 7] = vertices[1 * 48 + 8 * 3 +
7] = vertices[1 * 48 + 8 * 4 + 7] = texture.y + texture.z;
40     }
41     else {
42         vertices[2 * 48 + 8 * 1 + 6] = vertices[2 * 48 + 8 * 2 +
6] = vertices[2 * 48 + 8 * 3 + 6] = texture.x;
43         vertices[2 * 48 + 8 * 2 + 7] = vertices[2 * 48 + 8 * 3 +
7] = vertices[2 * 48 + 8 * 4 + 7] = texture.y;
44         vertices[2 * 48 + 8 * 0 + 6] = vertices[2 * 48 + 8 * 4 +
6] = vertices[2 * 48 + 8 * 5 + 6] = texture.x + texture.z;
45         vertices[2 * 48 + 8 * 0 + 7] = vertices[2 * 48 + 8 * 1 +
7] = vertices[2 * 48 + 8 * 5 + 7] = texture.y + texture.z;
46
47         vertices[3 * 48 + 8 * 0 + 6] = vertices[3 * 48 + 8 * 4 +
6] = vertices[3 * 48 + 8 * 5 + 6] = texture.x;
48         vertices[3 * 48 + 8 * 1 + 7] = vertices[3 * 48 + 8 * 3 +
7] = vertices[3 * 48 + 8 * 5 + 7] = texture.y;
49         vertices[3 * 48 + 8 * 1 + 6] = vertices[3 * 48 + 8 * 2 +
6] = vertices[3 * 48 + 8 * 3 + 6] = texture.x + texture.z;
50         vertices[3 * 48 + 8 * 0 + 7] = vertices[3 * 48 + 8 * 2 +
7] = vertices[3 * 48 + 8 * 4 + 7] = texture.y + texture.z;
51     }
52     glBindBuffer(GL_ARRAY_BUFFER, cubeVBO);
53     glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);
54
55     glBindVertexArray(cubeVAO);
56     glDrawArrays(GL_TRIANGLES, 0, 36);
57     glBindVertexArray(0);
58 }
```

- 天空盒：

- 绘制一个较大的立方体对象，使其将整个场景包含在内。然后对其使用立方体贴图：

```

1     unsigned int loadCubemap(std::vector<std::string> faces){
2         unsigned int textureID;
```

```

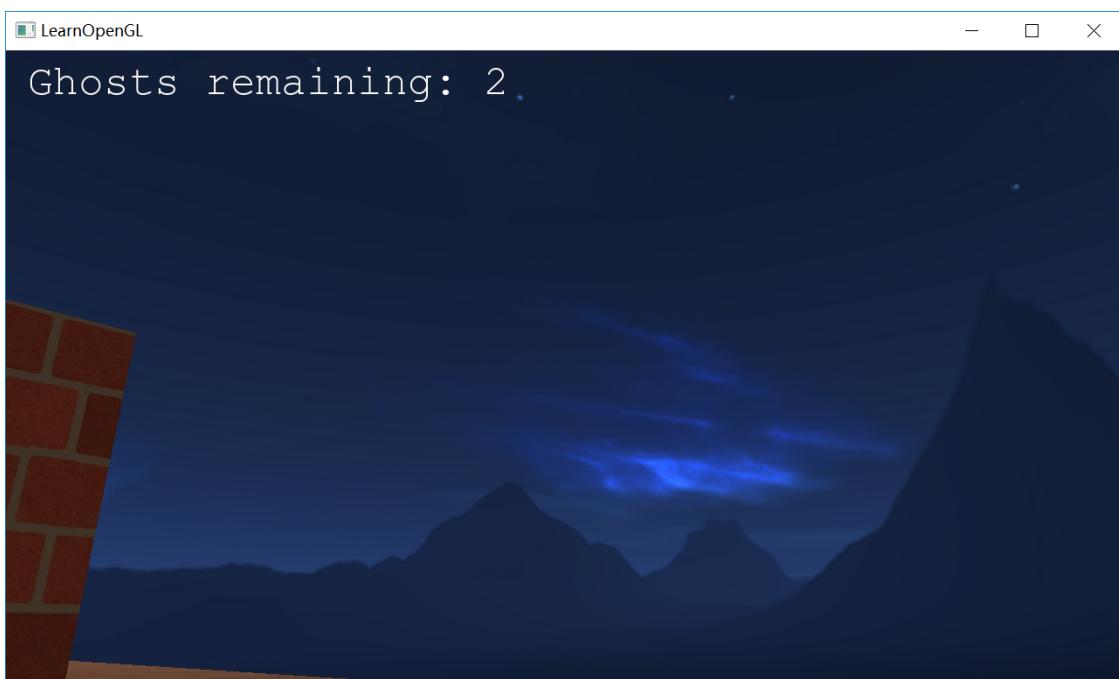
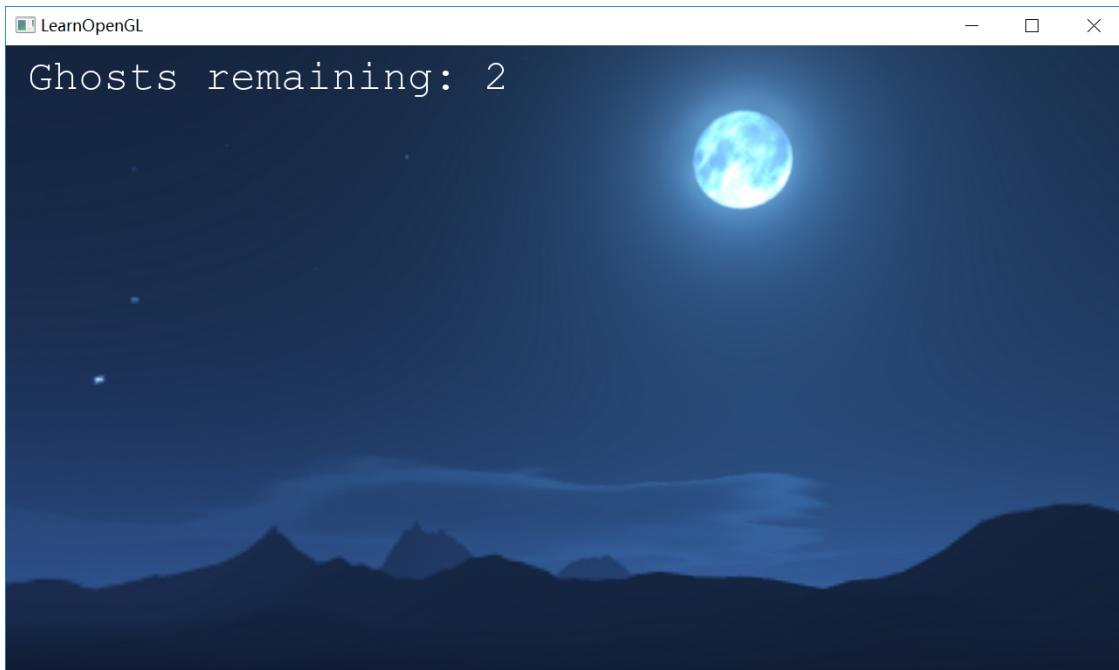
3     glGenTextures(1, &textureID);
4     glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
5     int width, height, nrChannels;
6     for (unsigned int i = 0; i < faces.size(); i++)
7     {
8         unsigned char *data = stbi_load(faces[i].c_str(), &width,
9             &height, &nrChannels, 0);
10        if (data)
11        {
12            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
13                         0, GL_RGB, width, height, 0, GL_RGB,
14                         GL_UNSIGNED_BYTE, data
15                     );
16            stbi_image_free(data);
17        }
18        else
19        {
20            std::cout << "Cubemap texture failed to load at path: "
21             << faces[i] << std::endl;
22            stbi_image_free(data);
23        }
24        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
25                         GL_LINEAR);
26        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
27                         GL_LINEAR);
28        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
29                         GL_CLAMP_TO_EDGE);
30        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
31                         GL_CLAMP_TO_EDGE);
32        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
33                         GL_CLAMP_TO_EDGE);
34    }
35    return textureID;
36}

```

- 由于随着位置的移动，天空盒的立方体与你的相对距离并不会改变（即天空与你的距离并不会随着你的移动而缩短），天空盒的观察矩阵应不存在深度变化。设置矩阵时，齐次坐标向量的最后一维固定为0：

```
1 |     view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
```

- 最后天空盒的效果如图：



4. 导入物体的移动与旋转

- 南瓜怪的基本设定
 - 南瓜的基本信息包括:
 - 世界系坐标 `Position`
 - 血量 `hitpoint`
 - 被击中时间 `hittime`
 - 我们的场景中南瓜不会翻转，因此只需要描述出南瓜的朝向，就可以完成南瓜的绘制。因此我们对每个南瓜维护三个方向向量 `Front`, `Left` 和 `Up`，他们满足右手系，只要确定了 `Front` 就可以确定剩下两个向量:
 - $\overrightarrow{Left} = (0, 1, 0) \times \overrightarrow{Front}$
 - $\overrightarrow{Up} = \overrightarrow{Front} \times \overrightarrow{Left}$



- 南瓜怪的行为

- 我们给南瓜幽灵设计了移动函数，来实现符合以下性质的场景内追逐
 - 移动连续，移动方向较少发生突变
 - 移动具有随机性
 - 在远离玩家时，做类似于无规则的漫游
 - 在接近玩家时，会直接冲向玩家
- 为此我们设定在每一帧，南瓜有 1% 的概率改变朝向，在改变朝向时有 $\frac{3}{dis}$ 的概率转向玩家的方向，其中`dis`为该南瓜与玩家的距离。这样在场景中南瓜本身改变方向的频率很小，但在距离较近时可以认为一定会转向玩家，使得南瓜更有幽灵的体验感。

```

1  bool Trymove(Point3D goal, double movelen) {
2      if (Checkhitpoint() == false) return false;
3      if((goal - pos).length() <= 0.5) {
4          return true;
5      }
6      else {
7          goal.y = pos.y;
8          // 1% probability
9          double nowdis = (goal - pos).length();
10         int trymodified = rand() % 100 <= 1;
11         bool check = abs(pos.x) > bordersize || abs(pos.y) >
bordersize;
12         if (trymodified) {
13             // 3/dis probability
14             int go = rand() % max(1, (int)(goal - pos).length());
15             if (go <= 3 || check)
16                 Front = goal - pos;
17             else
18                 Front = Point3D(rand() % 101 - 50, 0, rand() % 101
- 50);
19         }
20         // pumpkin move
21         pos = pos + Front.resize(movelen);
22         // maintain coordinates
23         Front = Front.resize(1);
24         Point3D temp = Point3D(0, 1, 0) * Front;
25         if (sgn(temp.length()) != 0) Left = temp;
26         Left = Left.resize(1);

```

```

27
28         Up = Front * Left;
29         Up = Up.resize(1);
30     }
31     return false;
32 }
```

- 子弹在场景中以直线运动；且碰墙会反弹，若一直没有碰到南瓜寿命为500帧；若碰到南瓜会使南瓜掉血并变红。与墙的碰撞在碰撞检测部分介绍。

5. 光源实现方法

- Phong光照模型

- 理论基础

$$\overrightarrow{lightdir} = \overrightarrow{fragpos} - \overrightarrow{lightpos}$$

$$\overrightarrow{viewdir} = \overrightarrow{fragpos} - \overrightarrow{viewpos}$$

$$\overrightarrow{normal} = \overrightarrow{fragedge_a} \times \overrightarrow{fragedge_b}$$

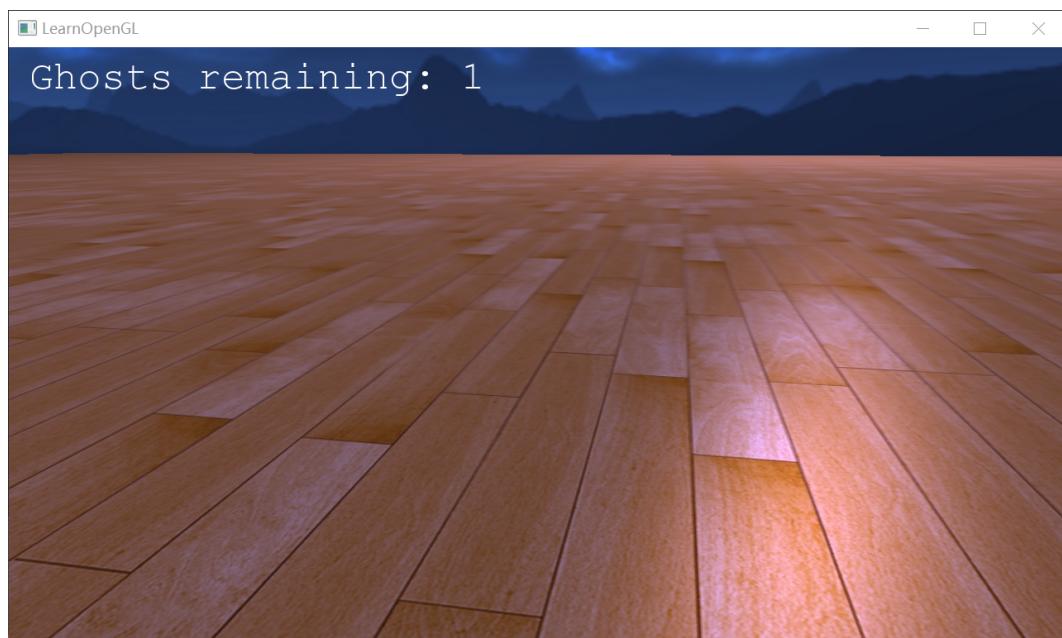
$$diff_intensity = \max(\overrightarrow{lightdir} \cdot \overrightarrow{normal}, 0)$$

$$spec_intensity = \max((\overrightarrow{lightdir} + \overrightarrow{viewdir}) \cdot \overrightarrow{normal}, 0)$$

$$color = ambient + (diffuse \cdot diff_intensity + specular \cdot spec_intensity) \cdot lightcolor$$

- 计算出物体表面各片元的颜色值。所得到的物体表面显示效果不仅有随角度而定的明暗效果，还有一定的镜面高光。

- 随着观察者位置和光源位置的移动，物体的明暗效果也会随之动态变化。



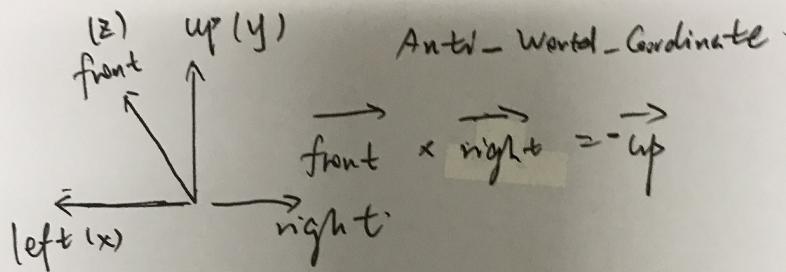
- 具体计算方式与实现：

- 由于使用了Phong而非Gouraud光照模型，在顶点着色器中只需要传送顶点与其法向数据。在片段着色器中，对于每一个面，使用三个顶点的光照向量进行插值，再代入上述公式中依次计算。
- 需要注意的是，在计算漫反射与镜面反射强度时，如果两个向量成钝角将会得到负数点积。此时如果直接将其代入计算会得到与事实不符的结果。实际上这个强度最小应该是0。
- 对于面数较少的简单几何体，可以把法向量直接输入到顶点信息中。这样能够减少片段着色器的计算。

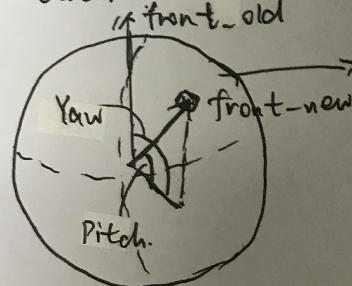
6. 场景漫游与人物移动的实现

- 基本场景漫游的实现
 - 为了简化建模，我们将游戏设计为第一人称游戏。因此人物移动和场景漫游所需要维护的方向向量就是在改变 camera 的视角等信息。因此以下的内容都被设计在 camera 类中。
 - 场景漫游通过维护一个观察者坐标系来实现，camera 的**左手系坐标系**不能仅由一个 front 来确定，简单来说同样是向前看，歪头后看到的场景也是不同的，这和viewport非常类似。为了简化模型，我们约定观察者坐标系的 Right 必须与世界坐标系中的地平面平行，这可以通过将 front 与世界坐标系的 up 叉乘以实现：
 - $\overrightarrow{Right} = \overrightarrow{Front} \times \overrightarrow{WorldUp}$
 - $\overrightarrow{Up} = \overrightarrow{Right} \times \overrightarrow{Front}$
 - 漫游的功能通过鼠标的动加以实现，视角的变化被描述为方向向量在球坐标系上的旋转。我们将该逻辑描述在函数 ProcessMouseEvent 中
 - 上下移动是 front 向量绕 x(-right) 轴旋转；
 - 左右移动是 front 向量绕 y(up) 轴旋转。
 - 这些可以描述为球坐标系中角度的变化：

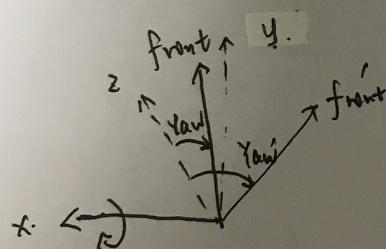
SO GED & MATH



Zuler

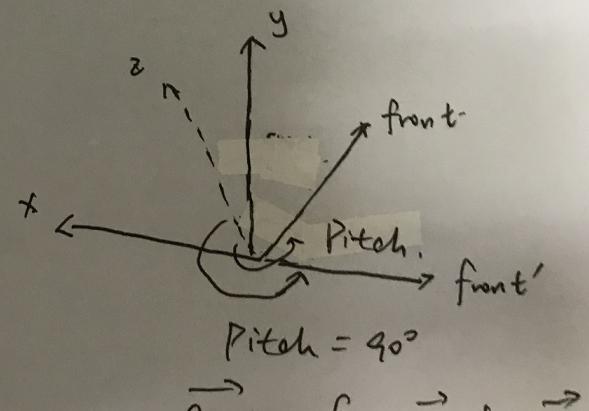


$$\begin{cases} x = R \cos(\text{Pitch}) \cos(\text{Yaw}) \\ y = R \sin(\text{Pitch}) \\ z = R \cos(\text{Pitch}) \cdot \sin(\text{Yaw}) \end{cases}$$



$$\text{Yaw} = 90^\circ$$

$\overrightarrow{\text{front}}$ from \vec{z} to \vec{y}



$$\text{Pitch} = 90^\circ$$

$\overrightarrow{\text{front}}$ from \vec{z} to \vec{x}

```

1 void ProcessMouseMovement(float xoffset, float yoffset, GLboolean
2 constrainPitch = true)
3 {
4     xoffset *= MouseSensitivity;
5     yoffset *= MouseSensitivity;
6
7     Yaw += xoffset;
8     Pitch += yoffset;
9
10    // Make sure that when pitch is out of bounds, screen doesn't get
11    // flipped
12    if (constrainPitch)
13    {
14        if (Pitch > 89.0f)
15            Pitch = 89.0f;
16        if (Pitch < -89.0f)
17            Pitch = -89.0f;

```

```

16     }
17
18     // Update Front, Right and Up vectors using the updated Euler
19     angles
20 }
```

- 将漫游附加在观察者坐标系中就构成了正确的玩家坐标系：

```

1 void updateCameraVectors()
2 {
3     // Calculate the new Front vector
4     glm::vec3 front;
5     front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
6     front.y = sin(glm::radians(Pitch));
7     front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
8     // to bring up accuracy
9     Front = glm::normalize(front);
10    // Also re-calculate the Right and Up vector
11    Right = glm::normalize(glm::cross(Front, WorldUp)); // Normalize
12    // the vectors, because their length gets closer to 0 the more you look up
13    // or down which results in slower movement.
14    Up = glm::normalize(glm::cross(Right, Front));
15 }
```

- 人物行为

- 人物移动服从物理规律。在地面上时，玩家可以：
 - 使用 WASD 在地面前后左右移动
 - 使用 SPACE 跳跃
 - 在跳跃时不可以改变移动方向
 - 即跳跃过程中的行为完全由跳起时的初速度决定
- 为了实现这一逻辑，我们给人物再维护一个速度向量 `velocity` 和一个加速度，人物移动通过两个函数来完成：
 - `ProcessKeyboard` 实现人物在地面上时的前后左右移动，`WASD` 只能调整 `x` 和 `z` 坐标，而 `SPACE` 会给速度向量添加 `y` 分量的初值；

```

1 if (Position.y <= 0)
2 {
3     //Point3D(Position).debug();
4     Position[1] = 0;
5     velocity[1] = 0;
6     glm::vec3 vFront, vRight;
7     vFront = glm::vec3(Front[0], 0, Front[2]);
8     vRight = glm::vec3(Right[0], 0, Right[2]);
9     normalize(vFront); normalize(vRight);
10    if (direction == FORWARD)
11        velocity = vFront;
12    else if (direction == BACKWARD)
13        velocity = -vFront;
14    else if (direction == RIGHT)
15        velocity = vRight;
16    else if (direction == LEFT)
17        velocity = -vRight;
```

```

18     else if (direction == JUMP)
19         velocity[1] = 1.4;
20     Position += velocity * k;
21
22     pair<bool, Point3D> ret = CheckvaliPosi(last);
23     if (ret.first == false)
24         Position = last;
25 }

```

- fall 在每一帧都会尝试进行，如果在地面上则无效，如果在空中则会先根据加速度和时间调整 velocity，再根据速度调整人物的位置。

```

1     if (Position[1] <= 0)
2     {
3         Position[1] = 0;
4         velocity = glm::vec3(0, 0, 0);
5     }
6     else
7     {
8         //Point3D(Position).debug();
9         velocity[1] -= g * k;
10        Position += k * velocity;
11        pair<bool, Point3D> ret = CheckvaliPosi(last);
12        if (ret.first == false)
13        {
14            if (sgn((ret.second - Point3D(0, 0, 0)).length())>0)
15            {
16                Point3D newv = subsao(Point3D(velocity), ret.second);
17                bool fg = Position.y <= -0.2;
18                velocity.x = newv.x;
19                velocity.y = newv.y;
20                velocity.z = newv.z;
21                Position += k * velocity;
22                if (fg)velocity = glm::vec3(0);
23            }
24        else
25        {
26            Position = last;
27        }
28    }
29 }

```

- checkValiPosi 是实现碰撞检测的函数，将在[场景实时碰撞检测实现方法](#)着重介绍。

7. 屏幕截图与加载文字

- 截图的主要功能函数
 - openGL 提供强大的函数 `glReadPixels(0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE, screenData);`
 - 分别表示截取的四个顶点位置，图片的色彩格式，保存数据的类型，保存到的数组首地址。
- 截图函数的实现
 - 在 `screenshort.h` 里要提供一个对外接口 `screeshot(width, height)`，接受屏幕大小，并将结果储存到一个指定的文件里。
 - 如果要保存到一个 `bmp` 文件里，还要有一点简单的图像信息处理的知识。在输出 `bmp` 前我们要指定它的头。例如：

```

1     BITMAPFILEHEADER bitmapFileHeader;
2     memset(&bitmapFileHeader, 0, sizeof(BITMAPFILEHEADER));
3     bitmapFileHeader.bfSize = sizeof(BITMAPFILEHEADER);
4     bitmapFileHeader.bfType = 0x4d42; //BM
5     bitmapFileHeader.bfOffBits = sizeof(BITMAPHEADER) +
6     sizeof(BITMAPINFOHEADER);
7
8     //填充BITMAPINFOHEADER
9     BITMAPINFOHEADER bitmapInfoHeader;
10    memset(&bitmapInfoHeader, 0, sizeof(BITMAPINFOHEADER));
11    bitmapInfoHeader.biSize = sizeof(BITMAPINFOHEADER);
12    bitmapInfoHeader.biWidth = width;
13    bitmapInfoHeader.biHeight = height;
14    bitmapInfoHeader.biPlanes = 1;
15    bitmapInfoHeader.biBitCount = 24;
16    bitmapInfoHeader.biCompression = 0;
17    bitmapInfoHeader.biSizeImage = width * abs(height) * 3;

```

- 注意，`bmp` 文件里的储存格式是 `BGR`，输出的时候要换一些位置。
- 会输出 `bmp` 后，我们就可以很轻松地实现这个接口了。

```

1     void screenshot(int width, int height){
2         unsigned char *screenData = new unsigned char[width * height *
3];
3         glReadPixels(0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE,
4         screenData);
5         WriteBitmapFile("screenshot.bmp", width, height, screenData);
6         delete []screenData;
7     }

```

- 最后，我们把这个函数关联到主程序的一个按键即可。
- 文字绘制的环境配置
 - 在 `openGL` 里绘制文字不是一件简单的事情。
 - 首先我们要在库文件里加入 `freetype` 文件夹，在链接文件里加入 `freetyped.lib`，然后主程序里加入如下代码；

```

1     #include <ft2build.h>
2     #include FT_FREETYPE_H

```

- 在写文字之前，还要专门给文字一个 `VAO` 和 `VBO`，并做如下预处理：

```

1     void textInit(unsigned int &VAO, unsigned int &VBO) {
2         FT_Library ft;
3         if (FT_Init_FreeType(&ft))
4             std::cout << "ERROR::FREETYPE: Could not init FreeType
5             Library" << std::endl;
6             FT_Face face;
7             if (FT_New_Face(ft, "cour.ttf", 0, &face))
8                 std::cout << "ERROR::FREETYPE: Failed to load font" <<
9                 std::endl;
10            FT_Set_Pixel_Sizes(face, 0, 48);
11            glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
12            for (GLubyte c = 0; c < 128; c++)
13            {

```

```

12         if (FT_Load_Char(face, c, FT_LOAD_RENDER))
13     {
14         std::cout << "ERROR::FREETYTPE: Failed to load Glyph"
15         << std::endl;
16         continue;
17     }
18     GLuint texture;
19     glGenTextures(1, &texture);
20     glBindTexture(GL_TEXTURE_2D, texture);
21     glTexImage2D(
22             GL_TEXTURE_2D,
23             0,
24             GL_RED,
25             face->glyph->bitmap.width,
26             face->glyph->bitmap.rows,
27             0,
28             GL_RED,
29             GL_UNSIGNED_BYTE,
30             face->glyph->bitmap.buffer
31         );
32     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
33 GL_CLAMP_TO_EDGE);
34     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
35 GL_CLAMP_TO_EDGE);
36     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
37 GL_LINEAR);
38     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
39 GL_LINEAR);
40     Character character = {
41         texture,
42         glm::ivec2(face->glyph->bitmap.width, face->glyph-
43 >bitmap.rows),
44         glm::ivec2(face->glyph->bitmap_left, face->glyph-
45 >bitmap_top),
46         face->glyph->advance.x
47     };
48     Characters.insert(std::pair<GLchar, Character>(c,
49 character));
50     }
51     glBindTexture(GL_TEXTURE_2D, 0);
52     FT_Done_Face(face);
53     FT_Done_FreeType(ft);
54     glGenVertexArrays(1, &VAO);
55     glGenBuffers(1, &VBO);
56     glBindVertexArray(VAO);
57     glBindBuffer(GL_ARRAY_BUFFER, VBO);
58     glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 6 * 4, NULL,
59 GL_DYNAMIC_DRAW);
60     glEnableVertexAttribArray(0);
61     glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 *
62 sizeof(GLfloat), 0);
63     glBindBuffer(GL_ARRAY_BUFFER, 0);
64     glBindVertexArray(0);
65 }
```

- 文字的渲染
 - 为了增强可拓展性，我们要对外提供这么一个接口：

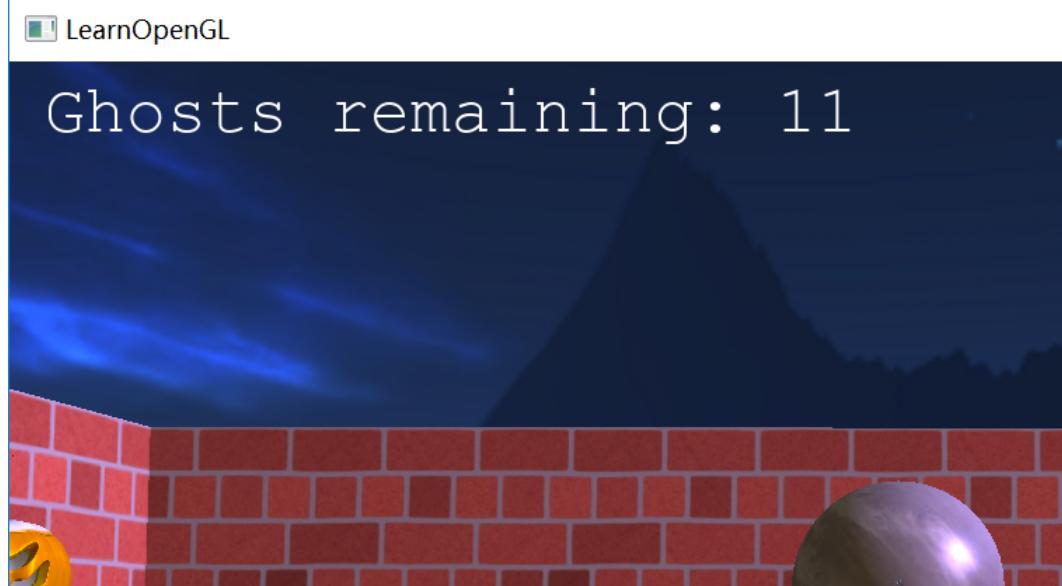
- `RenderText(shader &shader, unsigned int &VAO, unsigned int &VBO, std::string text, GLfloat x, GLfloat y, GLfloat scale, glm::vec3 color)`。
- `text` 表示渲染的文字内容, `x,y,scale` 表示字体的位置和大小, `color` 表示字体的颜色。
 - 渲染的时候, 注意还要打开开关:

```

1 |     glEnable(GL_CULL_FACE);
2 |     glEnable(GL_BLEND);
3 |     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

```

- 大概是类似于线框模式的渲染, 否则会渲染出一个一个方形。
- 效果如下:



8. 光照明实时阴影的实现

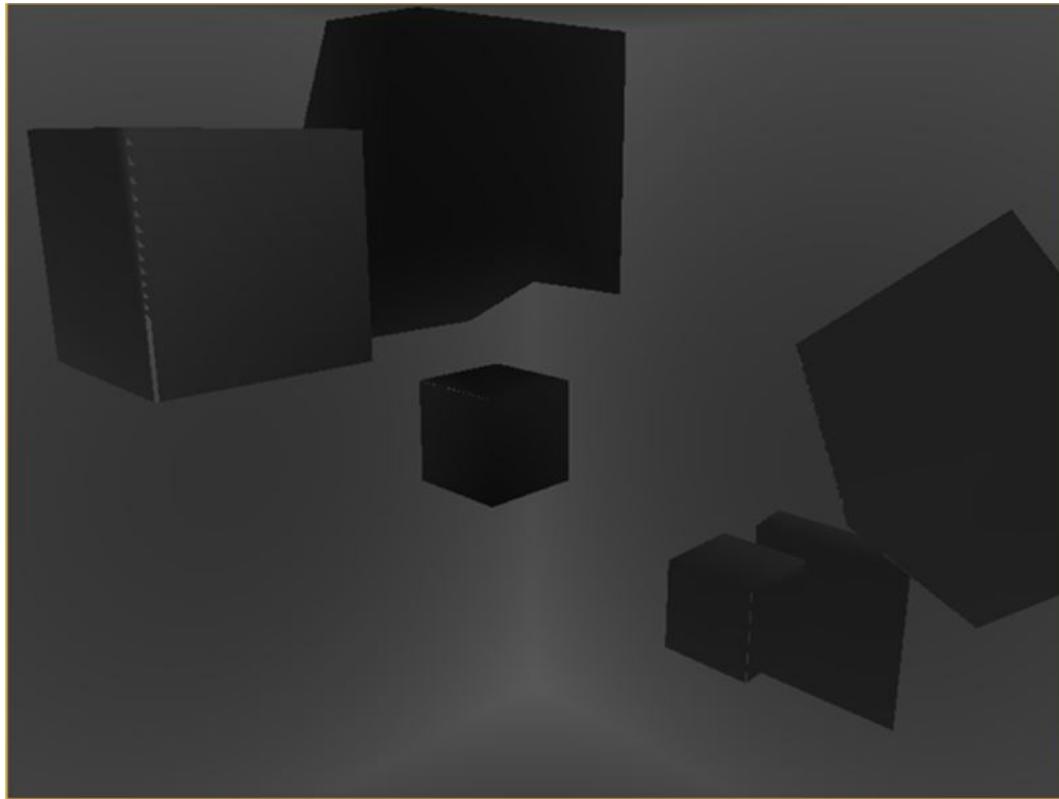
- **实时阴影是本作品的一大亮点。** 实现实时阴影的过程也是程序编写中遇到曲折比较多的一个部分。在制作阴影效果时, 我们使用的是阴影贴图技术。
 - 生成阴影贴图需要生成新的帧缓冲。区别于显示在屏幕上的缓冲, 我们需要在光源的位置观察各对象, 并以此计算各片元的深度, 以生成阴影贴图。

```

1 |     float lightDistance = length(FragPos.xyz - lightPos);
2 |     // map to [0;1] range by dividing by far_plane
3 |     lightDistance = lightDistance / far_plane;
4 |     // write this as modified depth
5 |     gl_FragDepth = lightDistance;

```

- 生成的阴影贴图大致是这个效果:



- 通过比较当前片元的深度是否在可观察平面的“后面”，来应用阴影贴图。

```
1 |     float shadow = shadows ? ShadowCalculation(fs_in.FragPos) : 0.0;
```

- 与平行光阴影不同的是，点阴影需要向六个方向分别计算深度（一个视界内不可能看到后面的物体）。这部分可以在几何着色器中完成。结束后，六个面的深度贴图都会加载到帧缓冲上。

```
1 |     for(int face = 0; face < 6; ++face)
2 |     {
3 |         gl_Layer = face; // built-in variable that specifies to which
4 |         // face we render.
5 |         for(int i = 0; i < 3; ++i) // for each triangle's vertices
6 |         {
7 |             FragPos = gl_in[i].gl_Position;
8 |             gl_Position = shadowMatrices[face] * FragPos;
9 |             EmitVertex();
10 |         }
11 |         EndPrimitive();
12 |     }
```

9. 场景实时碰撞检测实现方法

- 场景实时碰撞检测是我们的又一大亮点。我们实现了子弹和玩家与所有可破坏场景内物体的碰撞。
- 在进行碰撞检测时我们使用了如下逻辑：
 - 先沿当前方向尝试做一帧的运动
 - 得到从当前位置到下一位置的向量（尽管这个运动可能是一个斜抛运动，但我们可以用一个直线运动来近似）
 - 用该有向线段与模型面先判断是否有交
 - 若没有交肯定不会与该面碰撞

```

1 // ball
2 // here we consider all object has its shape
3 pair<double, Point3D> Ball::collide(const Point3D &A, const
Point3D &C, const Point3D &B) const {//this is reversed for
some reason...
4     double projdis = pointtoplanedistance(posi, A, B, C);
5     if (projdis < 0.01) return make_pair(1, velo);
6     if (pointtoplanedistance(posi + velo, A, B, C) >
0.25) return make_pair(1, velo);
7     Point3D temp = ((C - A) * (B - A)).resize(1);
8     if (sgn(pointtoplanedistance(posi, A, B, posi + velo))
< 0) return make_pair(1, velo);
9     if (sgn(pointtoplanedistance(posi, B, C, posi + velo))
< 0) return make_pair(1, velo);
10    if (sgn(pointtoplanedistance(posi, C, A, posi + velo))
< 0) return make_pair(1, velo);
11    return make_pair((projdis - 0.25) / -(velo%temp), velo
- temp.resize(velo%temp * 2));
12 }
13
14 // camera
15 pair<bool, double> getDistToFace(glm::vec3 position,
glm::vec3 velo, Point3D a, Point3D b, Point3D c)
16 {
17     Ball tmp = Ball(Point3D(position), Point3D(velo));
18     pair<double, Point3D> r = tmp.collide(a, b, c);
19     bool flag = sgn(r.first - 1) < 0;
20     return make_pair(flag, r.first);
21 }
```

- 若有交，则计算出从当前位置到该面沿移动方向的距离，存入vector
- 在所有有交的面中，选择距离最短的一个面，这个面必然就是碰撞发生的面。
- 这里以与墙的碰撞为例

```

1 for (wall *pwall : wallManager) {
2     int siz = pwall->allface.size();
3
4     vector< pair<double, Point3D> > collisions;
5     for (int i = 0; i < siz; i += 4) {
6         Point3D n;
7         pair<bool, double> res = getDistToFace(last, velo_vec,
pwall->allface[i], pwall->allface[i + 1], pwall->allface[i + 2]);
8         n = (pwall->allface[i + 1] - pwall->allface[i]) *
(pwall->allface[i + 2] - pwall->allface[i]);
9         if (res.first)
10            collisions.push_back(make_pair(res.second, n));
11
12         res = getDistToFace(last, velo_vec, pwall->allface[i +
2], pwall->allface[i + 3], pwall->allface[i]);
13         n = (pwall->allface[i + 3] - pwall->allface[i + 2]) *
(pwall->allface[i] - pwall->allface[i + 2]);
14         if (res.first)
15            collisions.push_back(make_pair(res.second, n));
16     }
17     if (collisions.size() > 0)
18     {
19         sort(collisions.begin(), collisions.end(), tcmp);
20     }
21 }
```

```

18         return make_pair(false, collisions[0].second);
19     }
20 }
```

- 反弹效果的实现

- 如上述，我们就完成了碰撞是否发生的检测，检测过后需要实现反弹的效果，这里我们在输出碰撞面的同时，就可以通过面上的点，得到他的法向量 \vec{n} 。假定当前速度向量是 \vec{v} ，则碰撞后满足：

$$\vec{v}' = \vec{v} - 2 \frac{\vec{v} \cdot \vec{n}}{|\vec{n}|} \hat{n}$$

- 在完成碰撞后，应该沿新的速度向量防线再做一次运动。

```

1     Point3D subsao(Point3D vx, Point3D vo)
2 {
3     double len = (vx % vo) / vo.length();
4     Point3D diao = vo * (len / vo.length());
5     return vx - diao * 2;
6 }
7
8     pair<bool, Point3D> ret = CheckValiPosi(last);
9     if (ret.first == false)
10    {
11        if (sgn((ret.second - Point3D(0, 0, 0)).length())>0)
12        {
13            Point3D newv = subsao(Point3D(velocity), ret.second);
14            bool fg = Position.y <= -0.2;
15            velocity.x = newv.x;
16            velocity.y = newv.y;
17            velocity.z = newv.z;
18            Position += k * velocity;
19            if (fg)velocity = glm::vec3(0);
20        }
21        else
22        {
23            Position = last;
24        }
25    }
```

10. 视口中心点对应方块与墙破坏的确定方法

- 确定视口中心点对应方块

- 首先确定每个墙与视线射线相交的方块，然后取出其中最近的方块。
- 最近的方块也不够近时会判定没有破坏。

```

1     double FlipFace(bool modified) {
2         double mi = inf;
3         if (endgame) return mi;
4         pair<wall*, pair<int, int>> ret =
5             make_pair(nullptr, make_pair(-1, -1));
6         for (wall *pwall : WallManager) {
7             pair<double, pair<wall*, pair<int, int>>>temp = pwall-
8             >clicked(
9                 Point3D(camera.Position),
10                Point3D(camera.Front))
```

```

9      );
10     if (temp.first < mi) {
11         mi = temp.first;
12         ret = temp.second;
13     }
14 }
15 if (ret.first != nullptr && mi < clickdist && modified) {
16     ret.first->bkdown(ret.second.first, ret.second.second),
17     puts("Modified");
18 }
19 }
```

- 墙与视线交点是射线与平面相交问题：

- 取点在平面上投影 \vec{p} ，方向向量投影 \vec{v} ，点到平面距离 d ，方向向量在平面法向量方向上长度 l ，有交点为 $\vec{p} + \vec{v} * d/l$ ，注意判定 $l \leq 0$ 的情况。

```

1     pair<double, pair<wall*, pair<int, int>>> wall::clicked(const
2     Point3D &circenter, const Point3D &direction) { //returns inf when not
3     clicked
4         //assert(sgn(direction.length())!=0);
5         //proj to 2D
6         Point2D projcenter(
7             (circenter - center) % (px - center) / xlength,
8             (circenter - center) % (py - center) / ylength
9         );
10        double projdis = pointtoplanedistance(circenter, center, px,
11        py);
12        Point3D temp = (px - center) * (py - center) / xlength /
13        ylength;
14        double clickdis = (direction % temp) /
15        direction.length(); //direction.length() == 1? what?
16        if (projdis < 0) {
17            projdis = -projdis - zlength;
18            if (projdis < 0) {
19                projdis = 0;
20                return make_pair(inf, make_pair(nullptr,
21                make_pair(-1, -1))); //walls can not be clicked when you are inside of
22                it; Should be handled separately?
23            }
24            if (sgn(clickdis) >= 0) return make_pair(inf,
25            make_pair(nullptr, make_pair(-1, -1)));
26            clickdis = -clickdis;
27        }
28        else {
29            if (sgn(clickdis) <= 0) return make_pair(inf,
30            make_pair(nullptr, make_pair(-1, -1)));
31        }
32        //intersection
33        Point2D intersection(
34            direction % (px - center) / xlength / direction.length()
35            * projdis / clickdis, //direction.length() == 1?
36            direction % (py - center) / ylength / direction.length()
37            * projdis / clickdis
38        );
39        intersection = intersection + projcenter;
```

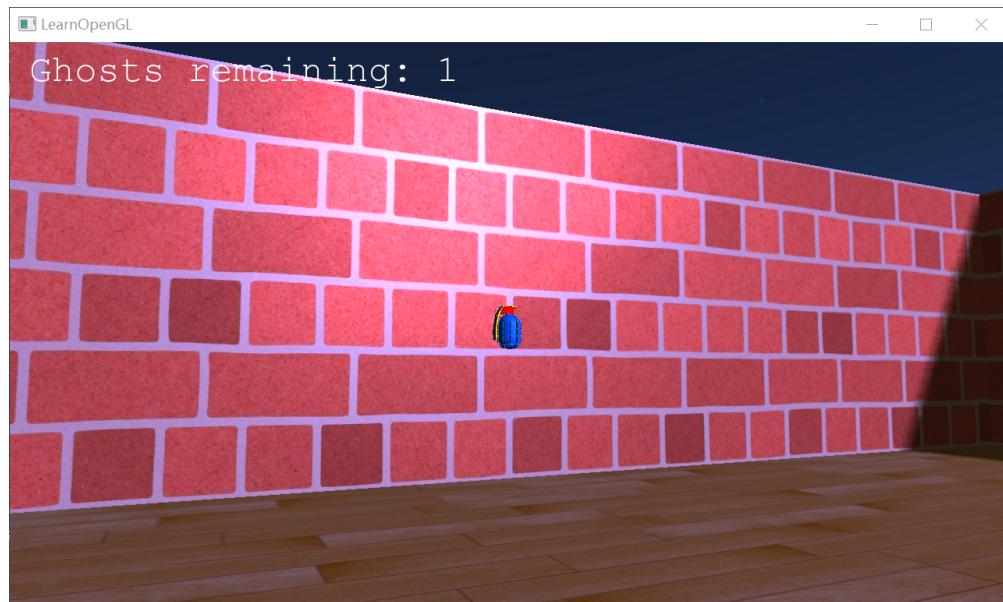
```

29         if (intersection.x < 0 || intersection.x >= xlength ||
30             intersection.y < 0 || intersection.y >= ylength)
31             return make_pair(inf, make_pair(nullptr, make_pair(-1,
32                                         -1)));
32
33         if (face[trunc(intersection.x / (xlength / accx))]
34             [trunc(intersection.y / (ylength / accy))].live == 0)
35             return make_pair(inf, make_pair(nullptr, make_pair(-1,
36                                         -1)));
37
38         return make_pair(
39             projdis / clickdis,
40             make_pair(
41                 this,
42                 make_pair(
43                     (int)trunc(intersection.x / (xlength / accx)),
44                     (int)trunc(intersection.y / (ylength / accy))
45                 )
46             )
47         );
48     }

```

- 最终效果

- 为了便于理解，我们在交点位置显示了一个手雷，嵌在墙的内部。



- 墙破坏

- 墙在被右键点击时，会被随机破坏。
- 破坏的形式是先保证中心以及上下左右四个位置被破坏，然后对角的四个位置每个都有 $\frac{1}{3}$ 的概率被破坏。

```

1 void wall::filp(int x, int y) {
2     if (x < 0 || x >= accx || y < 0 || y >= accy) return;
3     face[x][y].live = 0;
4 }
5 void wall::bkdown(int x, int y) {
6     filp(x, y);
7     filp(x - 1, y); filp(x + 1, y); filp(x, y - 1); filp(x, y + 1);
8     if (rand() % 3 == 0) filp(x - 1, y - 1);
9     if (rand() % 3 == 0) filp(x + 1, y - 1);
10    if (rand() % 3 == 0) filp(x - 1, y + 1);
11    if (rand() % 3 == 0) filp(x + 1, y + 1);
12 }
```

- 最终效果

- 已经被破坏的墙不能被点击，并且墙不会掉落，造成一种类似 Minecraft 的效果。



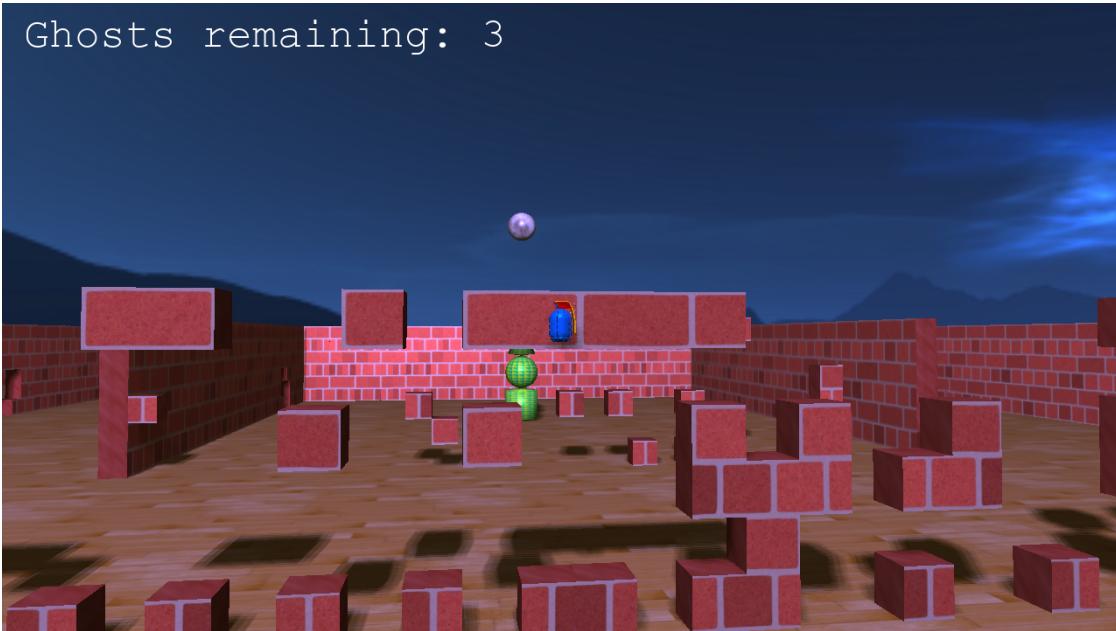
三、成果展示

- 游戏界面



- 墙破坏和跳跃

Ghosts remaining: 3



- 子弹和阴影

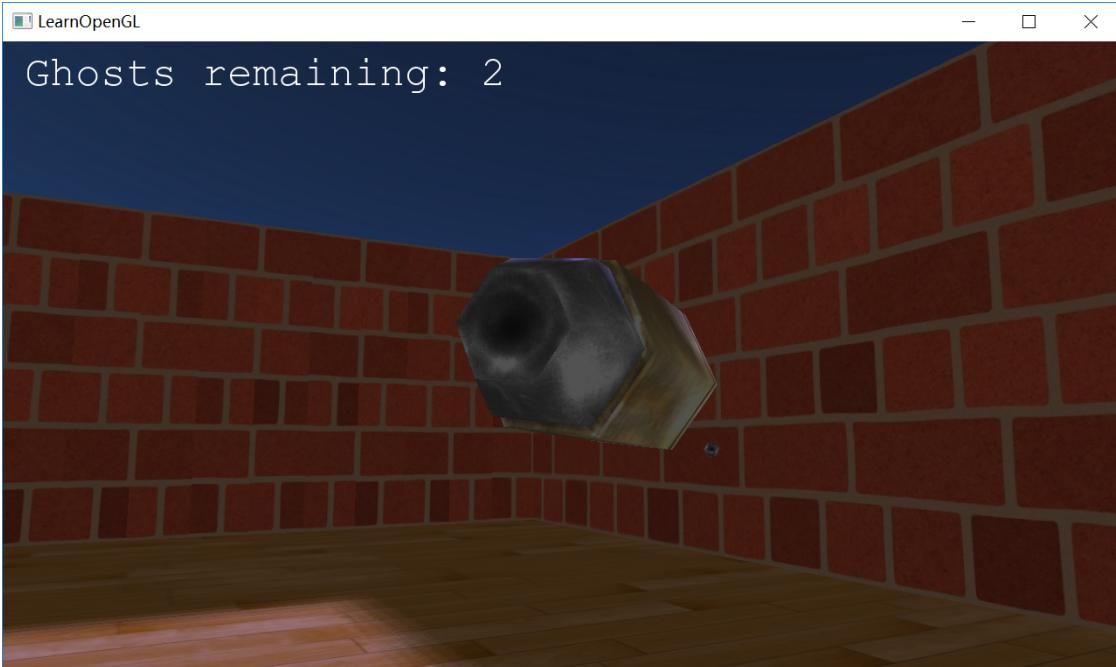
Ghosts remaining: 7



- 子弹放大图

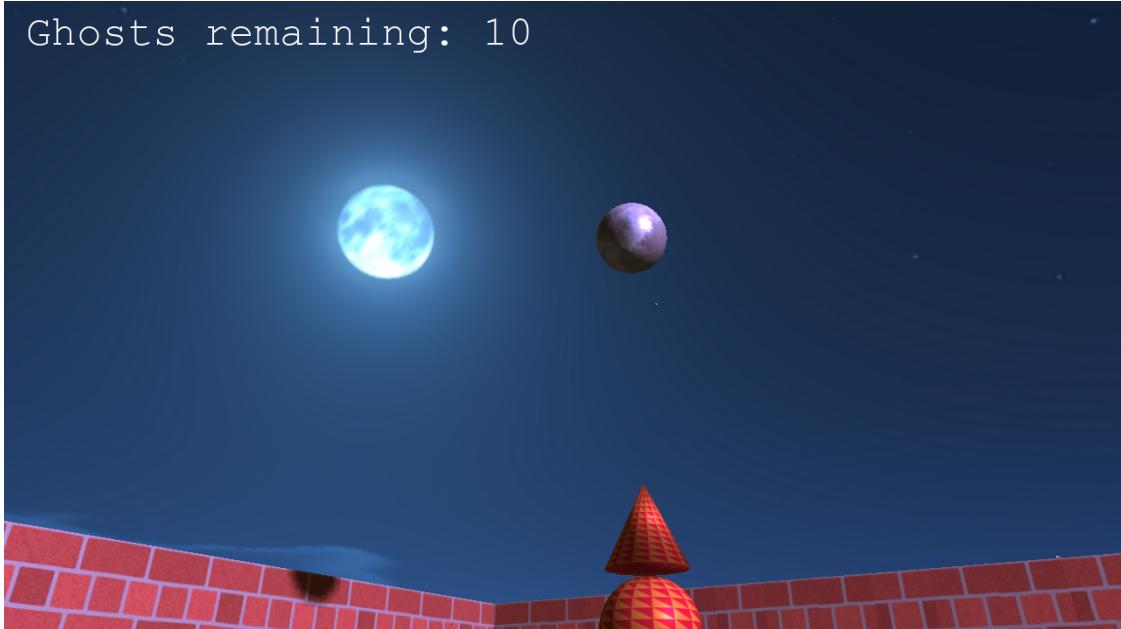
LearnOpenGL

Ghosts remaining: 2



- 月亮和天空盒

Ghosts remaining: 10

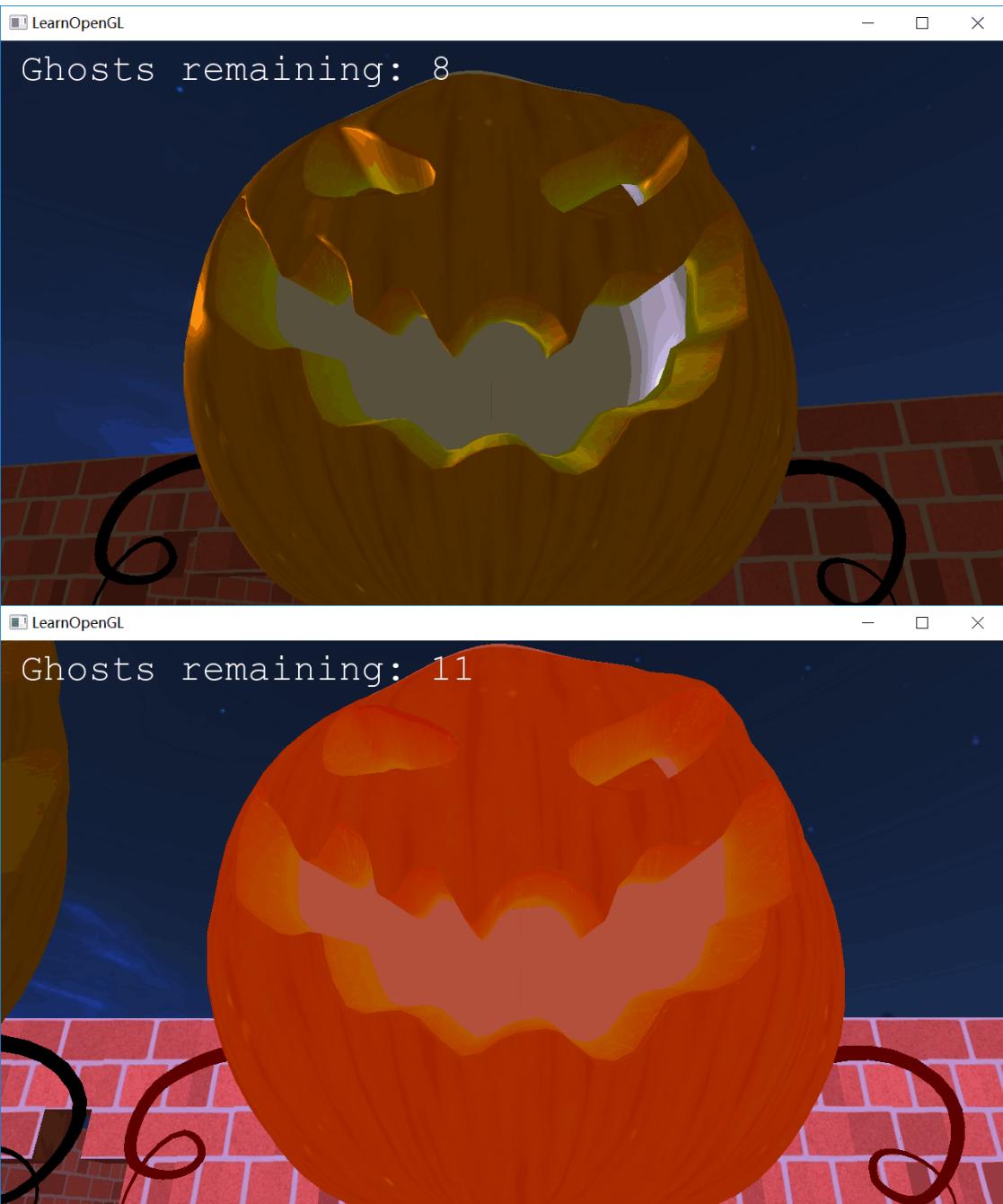


- 墙破坏的阴影

Ghosts remaining: 8



- 南瓜被击中的效果



四、小组分工

- 我们四个人配合默契，基本是按模块来分工的。
- 最终报告书写也基本是按照各自写的代码来分配的，详细如下：
 - 蒋仕彪 (Charpter 2, 3, 7)
 - 刘明锐 (Charpter 1, 9, 10)
 - 罗炜程 (Charpter 3, 5, 8)
 - 王宇晗 (Charpter 4, 6)