计算机图形学期末项目

烟花粒子系统说明文档

第 20 小组	
姓名	学号
陈德建	22336030
曹越	22336022
王俊亚	22307049
张晋	22336300

0 目录

I 项目说明

本项目的目的是对烟花粒子系统的构建。以下是本项目已完成或未完成的目标任务：

目标任务	完成情况
粒子系统的构建	✅
烟花系统的构建	✅
天空盒	✅
地面	✅
烟花的点光源创建与更新	✅
地面的 Blinn-Phong 光照	✅
烟花爆炸的音效	✅
中山大学校徽形状的烟花	✅
烟花爆炸的辉光特效	✅

II 实现过程

本项目中，重要的类(或着色器)及其功能如下所示：

类名/着色器	功能
`class Camera`	摄像机的相关操作
`class Shader`	着色器的集中管理
`class Laucher`	烟花的创建与更新
`class PointLigth`	点光源的相关操作
`terrain.frag`	地面的片段着色器

II.1 基础(待补充)

我们寻找得到了分别实现烟花系统和天空地面的示范性代码，通过结合 Assignment 0 中对 OpenGL 相关基本功能的构建，成功实现了基础功能：即粒子与烟花系统，天空盒与地面，全局点光源，以及地面对全局点光源的反射。

II.1.1 `terrain.frag` 的初版本

由于本项目并非软光栅化渲染器，所以与光线计算相关的逻辑都在 `GLSL` 着色器中(而不是在 `.cpp` 文件中)。地面对全局点光源的反射计算相关逻辑，在片段着色器 `terrain.frag` 中：

```glsl
#version 420 core

```

```
3  out vec4 FragColor;
4  in vec2 TexCoord;
5  in vec3 WorldPos; // 片段在世界空间的位置
6  uniform vec3 viewPos; // 视点位置
7  uniform vec3 lightPos; // 光源位置
8  uniform vec3 lightColor; // 光源颜色
9
10 layout (binding = 0) uniform sampler2D normalTexture;
11
12 float FogFactor(float d) {
13     const float FogMax = 750.0;
14     if (d >= FogMax) return 1.0;
15     return 1.0 - (FogMax - d) / (FogMax);
16 }
17
18 void main() {
19     // 法线处理
20     vec3 normalMap = texture(normalTexture, TexCoord).rgb;
21     normalMap = vec3(normalMap.x, normalMap.z, normalMap.y); // 转换法线
22     vec3 normal = normalize(normalMap * 2.0 - 1.0); // 从[0,1]转换到[-1,1]
23
24     // 环境光
25     vec3 terrainColor = vec3(0.31, 0.20, 0.08); // 地形颜色
26     float ambientStrength = 0.05;
27     vec3 ambient = ambientStrength * terrainColor;
28
29     // 光照计算
30     vec3 lightDir = normalize(lightPos - WorldPos); // 光源方向
31     vec3 viewDir = normalize(viewPos - WorldPos); // 视角方向
32     vec3 halfDir = normalize(lightDir + viewDir); // 半程向量
33
34     // Blinn-Phong 光照模型
35     float specularStrength = 0.5; // 高光强度
36     float shininess = 32.0; // 粗糙度
37     float ndotl = max(dot(normal, lightDir), 0.0); // 漫反射分量
38     float spec = pow(max(dot(normal, halfDir), 0.0), shininess); // 高光分量
39
40     // 计算最终光照
41      vec3 diffuse = ndotl * terrainColor * lightColor; // 漫反射 Add * lightColor: YuZhuZhi
```

```
42      vec3 specular = specularStrength * spec * lightColor * terrainColor; // 高光 Add * terrainColor: YuZhuZhi
43
44      vec3 lighting = ambient + diffuse + specular; // 总光照
45
46      // 雾效
47      float d = distance(viewPos, WorldPos);
48      float alpha = FogFactor(d);
49      vec3 FogColor = vec3(0.09, 0.11, 0.09); // 雾色
50
51      // 最终颜色
52      FragColor.rgb = mix(lighting, FogColor, alpha);
53      FragColor.a = 1.0;
54  }
```

II.2 烟花的点光源

为了实现烟花的光效，我们首先构造了点光源类 `PointLight`:

神 C++

```
1   using Color = glm::vec3;
2   using Position = glm::vec3;
3   using Attenuation = glm::vec3;
4   using Direction = glm::vec3;
5
6   class PointLight
7   {
8   public:
9     PointLight();
10      PointLight(const Color& color, const Position& position, const Attenuation& attenuation, const float life);
11    ~PointLight() = default;
12
13    #pragma region Get&Set
14    Color getColor() { return _color; }
15    Position getPosition() { return _position; }
16    Attenuation getAttenuation() { return _attenuation; }
17    float getLife() { return _life; }
18    // Unable to Set.
19    #pragma endregion
20
21     float distance(const Position& fragment); // Calculate DISTANCE between light & fragment.
```

```
22    Color calcAddColor(const Position& fragment, const Direction& normal); // Calculate
  COLOR should be ADDED on the fragment.
23    bool updateLife(const float decrease);
24
25    bool addToShader(const std::shared_ptr<Shader> shader, const int index);
26    bool deleteFromShader(const std::shared_ptr<Shader> shader, const int index);
27
28  public:
29    Color _color;
30    Position _position;
31    Attenuation _attenuation;
32    float _life;
33
34  };
```

II.2.1 `terrain.frag` 的修改

这个类中最重要的两个函数是`addToShader(...)`和`deleteFromShader(...)`。如同之前所述：由于本项目并非软光栅化渲染器，所以与光线计算相关的逻辑都在着色器中，因此创建点光源后应当将其添加到着色器中。为此，首先需要修改`terrain.frag`着色器，使之能够存储点光源相关信息。在头部添加结构体与变量：

```GLSL
1  struct PointLight {
2      vec3 position;
3      vec3 color;
4      vec3 attenuation; // 衰减参数 (constant, linear, quadratic)
5  };
6
7  #define MAX_LIGHTS 10 // 支持的最大点光源数量
8  uniform PointLight pointLights[MAX_LIGHTS];
```

在这里，为了节省计算时间，我们固定支持的最大点光源数量为10个。然后，仿照原先代码中的Blinn-Phong光照实现，也计算每个片段对烟花点光源的反应：

```GLSL
1  vec3 calcPointLightLighting(PointLight light, vec3 fragColor, vec3 fragPos,
   vec3 normal, vec3 viewPos) {
2      // 计算光源到片段的方向距离
3      vec3 lightDir = normalize(light.position - fragPos);
4      float distance = length(light.position - fragPos);
5
6      vec3 viewDir = normalize(viewPos - fragPos); // 计算视角方向
7      vec3 halfDir = normalize(lightDir + viewDir); // 半程向量
8
```

```
9       // 计算光源的衰减(这个参数计算结果有问题，可能根本没有设置到 pointLights 数组)
10      float attenuation = 1.0 / (light.attenuation.x + light.attenuation.y * distance
    + light.attenuation.z * distance * distance);
11
12      vec3 ambient = light.color * fragColor;
13
14      // 漫反射计算
15      float diff = max(dot(normal, lightDir), 0.0);
16      vec3 diffuse = diff * light.color * fragColor; // 漫反射
17
18      // 高光计算
19      float specularStrength = 0.5; // 高光强度
20      float shininess = 32.0; // 粗糙度
21      float spec = pow(max(dot(normal, halfDir), 0.0), shininess); // 高光部分
22      vec3 specular = specularStrength * spec * light.color * fragColor;
23
24      // 返回计算的漫反射和高光部分的总和
25      return (ambient + diffuse + specular) * attenuation;
26  }
```

这样，在着色器的主函数中，就可以对 pointLigths 中每一个点光源遍历计算并累加，所得结果再与之前实现的全局点光源相加，就能得到地面最终的颜色：

```
1       vec3 pointLighting = vec3(0.0);                                    懲 GLSL
2       for (int i = 0; i < MAX_LIGHTS; i++) {
3         pointLighting += calcPointLightLighting(pointLights[i], terrainColor, WorldPos,
    normal, viewPos);
4       }
5
6       vec3 lighting = ambient + diffuse + specular + pointLighting; // 总光照
```

现在我们可以结合在C++中实现的点光源类了。在创建点光源之后，应当手动将其添加到着色器中，否则着色器不能知道这个点光源的存在：

```
1   bool PointLight::addToShader(const std::shared_ptr<Shader> shader, const int
    index)                                                                 神 C++
2   {
3       if (index >= MAX_LIGHTS || index < 0) return false;
4       shader->use();
5
6           shader->SetVec3("pointLights["  +  std::to_string(index)  +  "].position",
    _position); // 设置位置
7       shader->SetVec3("pointLights[" + std::to_string(index) + "].color", _color); //
    设置颜色
```

```
8           shader->SetVec3("pointLights[" + std::to_string(index) + "].attenuation",
    _attenuation); // 设置衰减
9
10      return true;
11  }
```

应当注意到，C++的点光源类中具有属性_life，而着色器中的是没有的。因此点光源的生命控制应在 C++中完成。当点光源的生命耗尽，需要手动调用 deleteFromShader(...):

```
1   bool PointLight::deleteFromShader(const std::shared_ptr<Shader> shader, const
    int index)                                                                    神 C++
2   {
3       if (index < 0) return false;
4       shader->use();
5
6           shader->SetVec3("pointLights[" + std::to_string(index) + "].color",
    glm::vec3(0.0f)); // 设置颜色
7
8       return true;
9   }
```

II.2.2 嵌入到 class Laucher 中

烟花爆炸时生成点光源、消散时移除点光源，因此控制点光源的生命的任务自然要交给烟花类 class Laucher。首先在烟花类的成员变量中添加一个与着色器中对应的数组，以及一个指向相关着色器的指针:

```
1   std::array<std::shared_ptr<PointLight>, MAX_LIGHTS> pointLights;              神 C++
2   std::shared_ptr<Shader> shader;
```

烟花类中有一个 explode(...)函数，自然就在这个函数中创建点光源:

```
1   void Launcher::explode2(Particle& p)                                          神 C++
2   {
3     int randomSound = getRandomNumber(1, 6); // 随机选择爆炸声音
4     soundEngine->play2D(explosionSounds[randomSound - 1]); // 播放爆炸声音
5
6     // 添加点光源 Add: YuZhuZhi
7     auto pointLight = std::make_shared<PointLight>(
8       Color(p.r, p.g, p.b), // 光的颜色
9       p.pos,                    // 光的位置
10      Attenuation(1.0f, 0.014f, 0.007f), // 光的衰减参数
11      sparkleLife + 0.75
12    );
```

```
13    auto it = std::find(pointLights.begin(), pointLights.end(), nullptr);
14    if (it != pointLights.end()) { // 添加到集中管理
15      *it = pointLight;
16      pointLight->addToShader(shader, it - pointLights.begin());
17    }
18
19    ......
20  }
```

同时，烟花类中还有一个逐帧更新的函数 `update(...)`，它原先控制了粒子的生命的衰减，现在也要控制点光源的生命了：

```
1   void Launcher::update(Camera& camera, GLfloat* particle_position, GLubyte* particle_color)  神 C++
2   {
3     float deltaTime = Camera::getDeltaTime(); // 获取时间增量
4     ......
5     simulate(camera, particle_position, particle_color); // 模拟粒子
6
7     for (auto light = pointLights.begin(); light != pointLights.end(); light++) { // Add: YuZhuZhi
8       if (light->get() && !(light->get()->updateLife(deltaTime))) { // 删除点光源
9         int index = light - pointLights.begin();
10        light->get()->deleteFromShader(shader, index);
11        pointLights[index] = nullptr;
12      }
13    }
14
15    sortParticles(); // 排序粒子
16  }
```

以上就基本完成了点光源相关的逻辑。

II.2.3 遇到的困难

原先我们希望每一个爆炸的粒子都创建一个点光源，这就要求烟花类 `class Launcher` 中对点光源的管理的数据结构是动态数组 `std::vector`。但是，GLSL 中并没有动态数组，只有静态裸数组。因此，为了与着色器对应，我们只能也在烟花类中用相似的数据结构，即 `std::array`，并且限定相同的大小。

在原先的设想中，当一个点光源生命耗尽，只需要将它从动态数组中移除即可。但这显然带来一个问题：移除之后，C++中的点光源的下标就不与着色器中的对应了。因此这也是我们不得不选用 `std::array` 的理由。

同样的道理，即便使用静态数组，被删去的点光源可能在一串连续存在数组中的点光源的中间。而 GLSL 并没有指针类型，因此下次渲染时无法通过判断空指针来跳过这一被删去的点光源。

我们最终的解决方法是：考虑颜色为 `glm::vec3(0.0f)` 的光。这种光(向量)无论和谁相乘，都得到零向量或者零标量，即不影响最终结果。这就是为什么在 `deleteFromShader(...)` 中只是简单地将 `color` 设为 `glm::vec3(0.0f)` 的原因。

当然，这一问题也有更加聪明的解决方案，那就是在着色器的结构体 `PointLight` 中附设一个是否有效的属性。如果无效，就跳过不渲染。但由于前一种方法需要附带修改的地方不多，所以实现的是前一种方法。

II.3 高斯模糊(待补充)

III 小组分工

姓名	完成内容
陈德建	相关源代码契合 校徽变大过程 高斯模糊
曹越	相关源代码资源搜集 相关源代码契合 校徽形状烟花的创建
王俊亚	点光源类 烟花的点光源相关逻辑 README 撰写
张晋	PPT 制作