游戏项目设计分报告

作者	学号
唐小虎	3160103866

GLSL着色器类

接口设计

```
class Shader {
public:
    Shader() = delete;
    Shader(const std::string &vs_path, const std::string &fs_path);
    void Use() const;
    template <typename T> void SetUniform(const std::string &identifier,
const T&) const;

private:
    const FileManager file_manager = FileManager();

    static uint32_t Compile(GLenum type, const std::string &source,
const std::string &path);
    static uint32_t Link(uint32_t vs_id, uint32_t fs_id);

    uint32_t id;
};
```

其中默认构造函数被显式删除了，只提供`Shader::Shader(const string &, const string &)`的构造函数，两个字符串分别是定点着色器和像素着色器的地址；`Shader::Use()`只是`glUseProgram(uint32_t)`的封装。

而`template <typename T> Shader::SetUniform<T>(const string &, const T &)`则是其中最重要的接口，它是C++程序和运行在GPU上的GLSL连接点。它提供给C++环境直接设置GLSL全局变量的能力。其中我们支持的模板参数有`btTransform`、`glm::vec3`、`glm::mat4`、`int32_t`、`float`、`Attenuation`、`Material`、`Light`和

LightCollection。囊括了我们渲染所需要的全部变量（各种光源、光线衰减、矩阵、坐标空间转换等等）。

光线衰减

接口设计

```
class Attenuation {
private:
    float range_;
    float constant_;
    float linear_;
    float quadratic_;

public:
    Attenuation(float range);
    float range() const;
    float constant() const;
    float linear() const;
    float quadratic() const;
};
```

Attenuation类主要是个数据类，提供便捷的设置光线衰减参数的接口，并可以在Shader类中快速设置GLSL中对应的光线衰减结构体。

光线衰减一直到渲染时才会真正起作用：

```
float distance = length(light.position - vPosition);
float attenuation = 0.0f;
if (distance < light.attenuation.range) {
    attenuation = 1.0f / (light.attenuation.constant +
light.attenuation.linear * distance + light.attenuation.quadratic *
pow(distance, 2));
}

vec3 diffuseColor = diffuseFactor * material.diffuse * light.color;
vec3 specularColor = specularFactor * material.specular * light.color ;
return attenuation * (diffuseColor + specularColor);
```

上述着色器代码简单的展示了attenuation会在渲染中发生的作用，它会根据距离自适应地调整亮度（显然是越远越暗），让渲染效果看起来自然。

各类光源

接口设计

光的纯虚基类

```
enum class LightType {
    kPoint, kParallel, kSpot
};

class Light {
private:
    glm::vec3 color_;
    float intensity_;

public:
    virtual LightType type() const = 0;
    Light(glm::vec3 color, float intensity);
    virtual void Attach(std::weak_ptr<Object>) = 0;
    glm::vec3 color() const;
    float intensity() const;
};
```

点光源

```
class PointLight: virtual public Light {
 protected:
    glm::vec3 position_;
    Attenuation attenuation_;
    std::weak_ptr<Object> anchor_;
 public:
    PointLight(glm::vec3 position, glm::vec3 color, float intensity,
const Attenuation& attenuation);
    void Attach(std::weak_ptr<Object>);
    LightType type() const;
    glm::vec3 position() const;
    Attenuation attenuation() const;
};
```

点光源是包含位置和衰减信息的光。

平行光

```
class ParallelLight: virtual public Light {
 protected:
    glm::vec3 direction_;
 public:
    ParallelLight(glm::vec3 direction, glm::vec3 color, float
intensity);
    void Attach(std::weak_ptr<Object>);
    LightType type() const;
    virtual glm::vec3 direction() const;
};
```

平行光是包含方向信息的光。

聚光灯

```
class SpotLight: public PointLight, public ParallelLight {
 protected:
    float angle_;
 public:
    SpotLight(glm::vec3 position, glm::vec3 direction, glm::vec3 color,
float intensity, const Attenuation& attenuation, float angle);
    void Attach(std::weak_ptr<Object>);
    float angle() const;
    glm::vec3 direction(void) const;
    LightType type() const;
};
```

聚光灯则既是平行光（包含方向信息），也是点光源（包含位置和光线衰减信息）。

光线集合

```
class LightCollection {
private:
    glm::vec3 ambient_;
    std::vector<const Light*> light_ptrs_;

public:
    LightCollection(glm::vec3 ambient);
    void PushBack(const Light* light);
    const Light* operator[](int i) const;
    int Size() const;
    glm::vec3 ambient() const;
};
```

LightCollection则是简单地封装了多光源的场景。通过Shader类可以直接赋值给GLSL中对用的uniform变量。

Controller

纯虚基类

```
class Controller {
protected:
    Character &controlee_;

public:
    Controller() = delete;
    Controller(Character &controlee);
    virtual void Elapse(double time) = 0;
};
```

其中Character是一个拥有Move，Rotate或是Jump等操作方式的角色（这样的角色可以是人类角色、可以是怪物等等）。而Controller的作用就是抽象出它们的运动逻辑并操控这一类Character。

另外Character::Elapse(double)的作用是刷新角色模拟。因为真正连续的操作是不可能实现的，可以实现的只有在一帧一帧画面之间离散的模拟运动逻辑。它被设计成一个纯虚函数，留给子类单独实现。

使用键盘控制的角色控制器

```
class KeyboardController: public Controller {
private:
    Keyboard &keyboard_;

public:
    KeyboardController() = delete;
    KeyboardController(Character &controlee, Keyboard &keyboard =
Keyboard::shared);
    void Elapse(double time);
};
```

KeyboardController绑定一个Character和一个键盘实例，每当键盘触发一些事件的时候，都会在Keyboard::Elapse(double)中以回调的形式发送给KeyboardController，之后用这个刷新事件，并控制主角的移动。

自动机的NPC控制器

```
class AutomationController: public Controller {
private:
    enum class PatrolDirection {
        kXPosition, kXNegative
    };
    PatrolDirection patrol_direction_;
    Character &target_;
    std::shared_ptr<glm::vec3> patrol_ptr_;
    float patrol_radius_;

public:
    AutomationController() = delete;
    AutomationController(Character &controlee, Character &target, float
patrol_radius = 10);
    void Elapse(double time);
};
```

自动机控制器不依赖于键盘或是鼠标等外设，它会绑定一个被控制者controlee，它的目标角色（即攻击对象）target以及一个巡逻半径。当角色在视角之外的时候，怪物就会按照特定的巡逻半径周期性巡逻，否则就会追着攻击对象跑。

玩家池（包括NPC和主角）

```
class PlayerCollection {
private:
    std::vector<std::shared_ptr<Player>> hostile_collection_,
friendly_collection_;

public:
    PlayerCollection();
    void PushBackHostile(std::shared_ptr<Player> player_ptr);
    void PushBackFriendly(std::shared_ptr<Player> player_ptr);
    void Traverse(
        std::function<void(std::weak_ptr<Player>)> yield,
        std::function<bool(const Player &, const Player &)>
compare_function = [] (const Player &a, const Player &b) -> bool {
            if(!a.object_ptr().lock())return false;
            if(!b.object_ptr().lock())return true;
            return a.object_ptr().lock()->GetOrigin()[2] >
b.object_ptr().lock()->GetOrigin()[2];
        }

    );
```

```
    void Query(glm::vec3 location, float width, float depth,
std::function<void(std::weak_ptr<Player>)> yield);
    void InitPlayerCollection(World *world_ptr);
    std::shared_ptr<Player> leader();
};
```

- `PlayerCollection::PlayerCollection()`默认会构建一个空的玩家池。
- `void PlayerCollection::PushBackHostile(shared_ptr<Player>)`会往敌人中添加一个玩家。
- `void PlayerCollection::PushBackFriendly(shared_ptr<Player>)`会往我方添加一个玩家（通常只有主角一个人）。
- 由于渲染上的一些限制（比如说由于我们的角色贴图是`png`格式带透明度的，渲染的时候一定要先渲染远处的角色再渲染近处的角色），我们在遍历角色池的时候可能必须要按照一定的顺序。所以我们设计了这样的遍历接口`void Traverse(function<void(std::weak_ptr<Player>)>, function<bool(const Player &, const Player &)>`。这个接口会按照`compare_function`提供的排序顺序依次给出角色池中的角色。
- 另外，在判定受攻击的角色对象时，我们需要查询角色池中属于特定区域的角色。而`void Query(glm::vec3, float, float, function<void(std::weak_ptr<Player>)>)`会帮我们做好这些工作。
- `shared_ptr<Player> leader()`的作用是返回主角。