安徽理工大學

ANHUI UNIVERSITY OF SCIENCE & TECHNOLOGY

本科毕业设计说明书

基于 OpenGL 的简单 3D 渲染引擎的实现

学院（部）：计算机科学与工程学院

专业班级：计算机科学与技术 14-2

学生姓名：程 亮

指导教师：程 建

2018 年 5 月 25 日

基于 OpenGL 的简单 3D 渲染引擎的实现

摘要

三维图形技术是近年来得到众多关注和发展较快的技术之一，在虚拟现实、 实时仿真、数字城市等领域有着广泛的应用。尤其数字娱乐产业在我国发展迅速， 然而其中的核心技术三维图形引擎大部分被国外所占领，因此对它的研究变得十分有意义。

图形渲染是三维引擎的重要组成部分。本文中我们利用 OpenGL，C++编程语言，图形学知识实现了一个简单的渲染引擎 MongoEnigne,该引擎采用的是可编程渲染管线,代码结构清晰简单，通过主要的一些部件构成整个引擎，主要实现了物体顶点坐标变换，Phong 氏光照模型，三种光源的光照效果（平行光、点光、聚光），obj 模型的加载，场景树的构建，引擎资源管理器。

本文的最后，我们对整个引擎的工作进行了总结，并探讨了 MongoEnigne 接下来的发展方向。

关键词：图形渲染，三维引擎，OpenGL，图形学，C++

THE DESIGN OF SIMPLE RENDER ENIGNE BASED ON OPENGL

ABSTRACT

The three-dimensional graphics technology is one of the technologies that has received much attention and rapid development in recent years and has a wide range of applications in fields such as virtual reality, real-time simulation, and digital city. In particular, the digital entertainment industry has developed rapidly in China. However, most of its core technology three-dimensional graphics engines have been occupied by foreign countries. Therefore, research on it has become very significant.

Graphics rendering is an important part of the 3D engine. In this paper, we use OpenGL, C++ programming language, graphics knowledge to achieve a simple rendering engine MongoEnigne, the engine uses a programmable rendering pipeline, the code structure is clear and simple, through the main components of the entire engine, the main realization of the object Vertex coordinate transformation, Phong's illumination model, lighting effects of three light sources (parallel light, spot light, spot light), loading of obj model, construction of scene tree, engine resource manager.

At the end of this article, we summarized the work of the entire engine and discussed the next development direction of MongoEnigne.

KEYWARDS：graphics rendering,three-dimensional engine,OpenGL, Graphics, c++

目录

摘要 I
ABSTRACT II
1. 绪论 1
1.1 计算机图形学介绍 1
1.2 OpenGL 介绍 1
1.3 常用模型和算法 2
1.3.1 渲染管线 2
1.3.2 颜色模型 3
1.3.3 Phong 氏光照模型 4
1.3.4 坐标系统 4
1.4 MongoEngine 技术架构 6
1.4.1 所用到的库 6
1.4.2 运行环境 6
2. 总体设计 7
2.1 需求分析 7
2.2 架构设计 7
3. 详细设计与实现 9
3.1 Camera 组件 9
3.2 Shader 组件 10
3.3 Texture 组件 11
3.4 Mesh 组件 12
3.5 Material 组件 13
3.6 Loader 组件 14
3.7 Light 组件 14
3.8 Scene 组件 17
3.9 其他组件 19
4. 实现效果 20
5. 总结 32
参考文献 34

1. 绪论

1.1 计算机图形学介绍

计算机图形学即CG(Computer Graphics)是随着计算机技术发展而产生和发展起来的，是计算机技术和和电视技术、图形处理技术相互融合的结果。

实际上计算机图形学的主要任务就是将数学公式和物理定律描述的世界场景展现在显示器的屏幕上。计算机图形学主要分成两个分支，一个是实时绘制技术，另一个是真实感绘制技术，本文实现的是实时绘制技术，真实感绘制技术的一个比较常用的技术是光线追踪。

现代计算机图形学主要讨论的是三维图形，二维图形的处理过程可以认为是三维图形处理过程的子集。图形学诞生的标志是1963年IvanSutherland的博士论文Sketchpad。Sketchpad仅涉及二维图形，Sutherland从事图形学研究直到1975年，这些年里他相继开拓了三维图形学、虚拟现实领域，成为计算机图形学之父。

1.2 OpenGL介绍

OpenGL是一套应用程序编程接口(API)，借助这个API我们开发人员就可以开发出对图形硬件具有访问的能力的程序。我们可以使用OpenGL开发出运行效率较高的图形程序或游戏，因为OpenGL非常接近底层硬件并且OpenGL使得我们不必去关注图形硬件的细节。OpenGL程序与平台是无关的，所以OpenGL API中不包含任何输入函数或窗口函数，原因是因为这两种函数都要依赖于特定的平台，例如Windows，Linux或是其他系统。

OpenGL API是过程性的，不是描述性的，即OpenGL不是面向对象的，所以OpenGL无法利用面向对象的特性，例如重载，继承等，但是我们可以使用面向对象的程序与OpenGL的实现进行链接就可以了。作为开发人员来说，我们不需要去描述场景的性质和外观，而是去确定一些操作步骤，为些操作步骤是为实现一定图形或图像所服务的。我们在实现这些步骤时可以调用OpenGL中的一些命令，可以利用这些命令绘制点、直线、多边形或是其它图形，还可以调用这些命令实现光照、着色，动画等各种效果。

OpenGL的实现可以是软件实现，也可以是硬件实现。软件实现是对OpengGL函数调用时作出的响应并创建二维或三维图像的函数库，那么硬件实现则是通过设置能够绘制图形或图像的图形卡驱动程序。一般来说，硬件实现要比软件实现快得多。我们都应该熟悉，在Windows上，是由图形设备接口将图形或图像显示在屏幕上或是其他显示设备上的。OpenGL的实现就软件实现来说，在Windows上会根据程序命令的要求，生成相应的图形或图像，然后会将这个图形或图像移交给图形设备接口，由图形设备接口将图形

或是图像显示在我们的屏幕上或是其他显示设备。

1.3 常用模型和算法

1.3.1 渲染管线

所谓渲染管线就是 OpenGL 的管道当中各个部分的功能以及如何在管道当中形成了我们想要的最终的一幅图。MongoEnginecai 用的是可编程渲染管线，其渲染管线主要包括以下流程：

图 1-1 可编程渲染管线

准备顶点数据：将顶点数据包括位置、法线、纹理坐标等等从 CPU 上传送到 GPU 上。

顶点着色器：顶点着色器中输入值是我们之前传入到 GPU 上的顶点数据，通过一系

列运算后，输出顶点在世界坐标系下的位置，同时可以向接下来的着色器传输必要数据。这个阶段是可编程的。

图元装配：将上一阶段顶点着色器的输出值根据图元类型进行装配。

几何着色器：几何着色器的输入是一个图元（如点或三角形）的一组顶点。几何着色器可以在顶点发送到下一着色器阶段之前对它们随意变换。这个阶段是可编程的。

测试和混合：测试包括深度测试，模板测试等，混合指的是当前像素值和原先的像素值进行组合，可以用不同的混合方式达到不同的效果。

片段着色器：在片段着色器中进行一系列计算，输出的片段的最终颜色。这个阶段是可编程的。

光栅化：光栅化的过程是将我们计算出来的每个片段的颜色同屏幕上的每个像素点对应，决定每个像素的颜色，即我们最终看到的样子。

概括来讲，渲染管线是从模型数据到图像生成过程的一种描述。Vertex Shader 能对顶点数据写处理算法，而 Fragment Shader 允许我们对像素数据写处理算法。

1.3.2 颜色模型

光是波长在可见光谱范围内的电磁波。可见光的波长大约在 400-700nm，正是这样的电磁波让人产生了颜色的感觉，颜色是外来的光线刺激人的视觉器官而产生的主观感觉。我们将红绿蓝作为三原色，其他所有的颜色都可以由这三种颜色以一定的比例混合得到。

现实世界中有无数种颜色，每一个物体都有它们自己的颜色。我们需要使用（有限的）数值来模拟真实世界中（无限）的颜色，所以并不是所有现实世界中的颜色都可以用数值来表示的。颜色可以数字化的由红色(Red)、绿色(Green)和蓝色(Blue)三个分量组成，它们通常被缩写为 RGB。仅仅用这三个值就可以组合出任意一种颜色。

MongoEngine 中用到的颜色模型就是 RGB 模型，同样 RGB 模型也是显示器的颜色物理模型。我们用三维向量来表示一种颜色，并将每个颜色分量规范化到 0-1.0 范围内。

图 1-2 RGB 三原色

1.3.3 Phong 氏光照模型

现实世界的光照是极其复杂的，而且会受到诸多因素的影响，这是我们有限的计算能力所无法模拟的。因此 OpenGL 的光照使用的是简化的模型，对现实的情况进行近似，这样处理起来会更容易一些，而且看起来也差不多一样。这些光照模型都是基于我们对光的物理特性的理解。其中一个模型被称为冯氏光照模型(Phong Lighting Model)。冯氏光照模型的主要结构由 3 个分量组成：环境(Ambient)、漫反射(Diffuse)和镜面(Specular)光照。

环境光照(Ambient Lighting)：即使在黑暗的情况下，世界上通常也仍然有一些光亮（月亮、远处的光），所以物体几乎永远不会是完全黑暗的。为了模拟这个，我们会使用一个环境光照常量，它永远会给物体一些颜色。

漫反射光照(Diffuse Lighting)：模拟光源对物体的方向性影响(Directional Impact)。它是冯氏光照模型中视觉上最显著的分量。物体的某一部分越是正对着光源，它就会越亮。

镜面光照(Specular Lighting)：模拟有光泽物体上面出现的亮点。镜面光照的颜色相比于物体的颜色会更倾向于光的颜色。

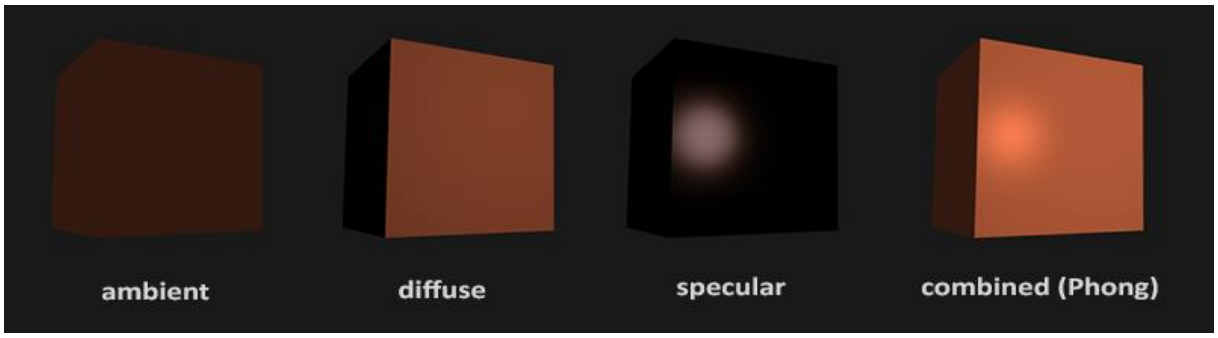

图 1-3 phong 氏光照模型

1.3.4 坐标系统

OpenGL 希望我们在顶点着色器运行后，所有可见的顶点坐标都变为标准化设备坐标，也就是说我们的 x,y,z 坐标都需要在 0-1 之间。超出这个坐标范围的顶点都将不可见。我们通常会自己设定一个坐标的范围，之后再在顶点着色器中将这些坐标变换为标准化设备坐标。然后将这些标准化设备坐标传入光栅器，将它们变换为屏幕上的二维坐标或像素。

将坐标变换为标准化设备坐标，接着再转化为屏幕坐标的过程通常是分步进行的，也就是类似于流水线那样子。在流水线中，物体的顶点在最终转化为屏幕坐标之前还会被变换到多个坐标系统。将物体的坐标变换到几个过渡坐标系的优点在于，在这些特定的坐标系统中，一些操作或运算更加方便和容易。对我们来说比较重要的总共有 5 个不

同的坐标系统：

图 1-4 坐标系变换

局部坐标是对象相对于局部原点的坐标，也是物体起始的坐标。

下一步是将局部坐标变换为世界空间坐标，世界空间坐标是处于一个更大的空间范围的。这些坐标相对于世界的全局原点，它们会和其它物体一起相对于世界的原点进行摆放。

接下来我们将世界坐标变换为观察空间坐标，使得每个坐标都是从摄像机或者说观察者的角度进行观察的。

坐标到达观察空间之后，我们需要将其投影到裁剪坐标。裁剪坐标会被处理至-1.0到 1.0 的范围内，并判断哪些顶点将会出现在屏幕上。

最后，我们将裁剪坐标变换为屏幕坐标，我们将使用一个叫做视口变换的过程。视口变换将位于-1.0 到 1.0 范围的坐标变换到由 glViewport 函数所定义的坐标范围内。最后变换出来的坐标将会送到光栅器，将其转化为片段。

我们将为每一个坐标系变换都创造一个矩阵：模型矩阵、观察矩阵和投影矩阵。一

个顶点坐标将会根据以下过程被变换到裁剪坐标：

$$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$$

最后的顶点应该被赋值到顶点着色器中的 gl_Position，OpenGL 将会自动进行透视除法和裁剪。

1.4 MongoEngine 技术架构

1.4.1 所用到的库

MongoEngine 中用到了一些现有的 c++库，模型解析用的是 assimp，数学库用的是 glm,文本渲染用的是 freetype，播放声音用的是 irrklang，版本控制用的是 git。

1.4.2 运行环境

MongoEngine 的开发环境是 windows10 操作系统,VisualStudio2017 集成开发环境，用的编程语言是 C++。

2. 总体设计

2.1 需求分析

MongoEngine 作为一个简单的渲染引擎，完成的功能有：

1. 加载资源，包括 shader,texture,mesh 等；
2. 解析 obj 模型；
3. 冯氏光照系统的实现；
4. 场景节点树的构建；

2.2 架构设计

MongoEngine 主要的架构如图 1-3 所示：

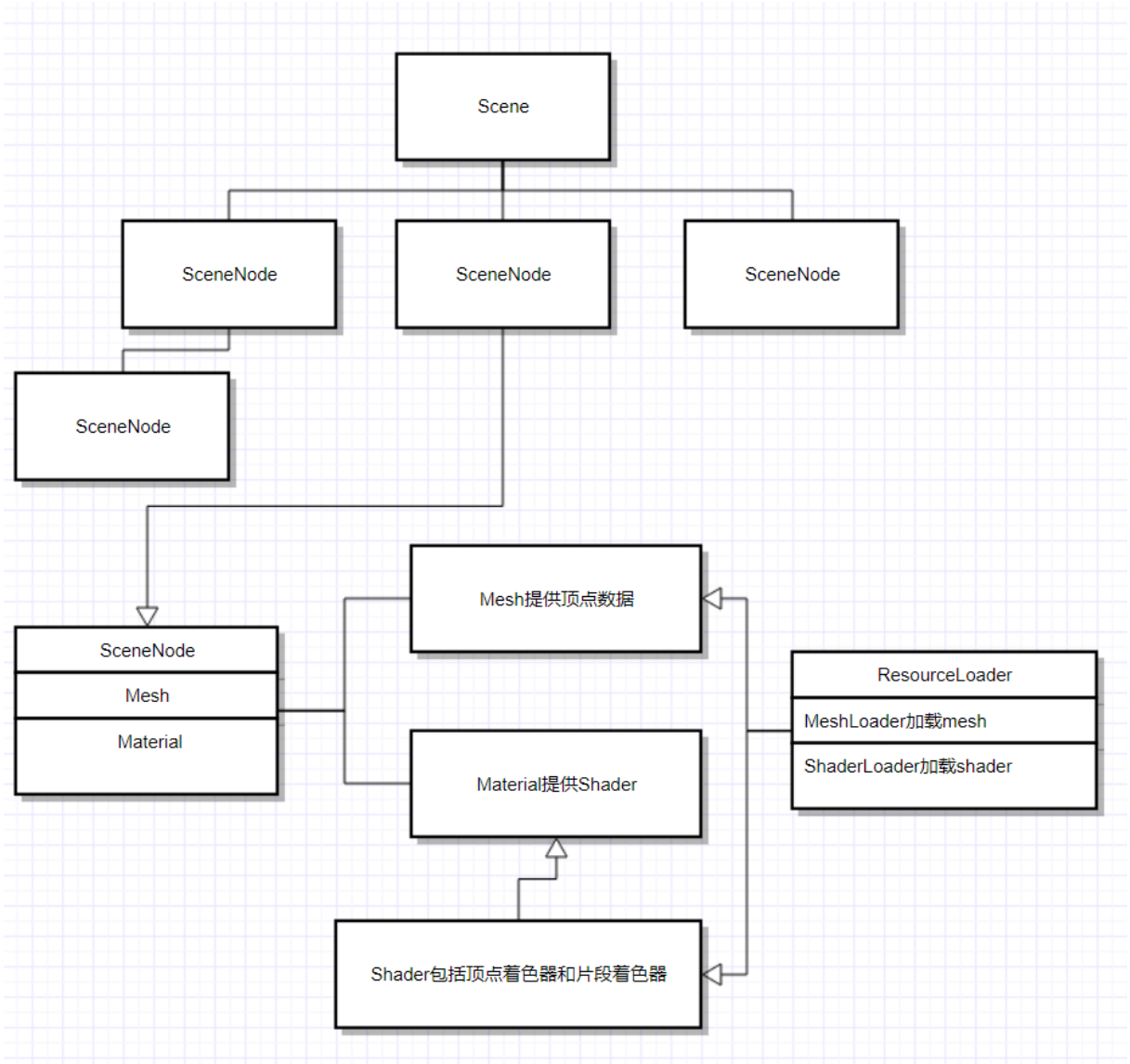

图 1-5 MongoEngine 架构

MongoEngine 主要由几个组件组成，包括:

1. Camera 组件：负责观察点的移动，在场景中漫游，以便更好的观察整个场景。
2. Shader 组件：负责读取 shader 文件，创建 shader，并编译，设置着色器中的 uniform 变量。
3. Texture 组件：对纹理的抽象形成的一种数据结构。
4. Mesh 组件：负责提供定点数据。
5. Material 组件：对 shader 的封装。
6. Loader 组件：MongoEngine 的资源加载系统，可以加载包括 shader、纹理、mesh 等资源。
7. Light 组件：对光源的抽象，不同光源持有不同的光源参数。
8. Scene 组件：负责整个场景的初始化，持有场景节点树，在渲染循环中通过深度优先遍历便利整个节点树，并渲染节点，并且持有整个场景中的光源集合，负责对光源的更新。

3. 详细设计与实现

3.1 Camera 组件

当我们渲染一个场景时，需要通过一个摄像机来观察整个场景，并且这个摄像机要能根据我们的鼠标键盘输入在场景中移动，OpenGL 中并没有摄像机的概念，所以我们需要自己通过抽象来实现一个摄像机系统。

当我们在讨论摄像机的时候，实际上就是在讨论场景中的顶点的世界坐标在摄像机定义的摄像机空间中的坐标，这个时候我们需要定义好这个坐标空间的三个轴和坐标原点即可，分别是摄像机在世界空间中的位置，观察方向，右方向，上方向。

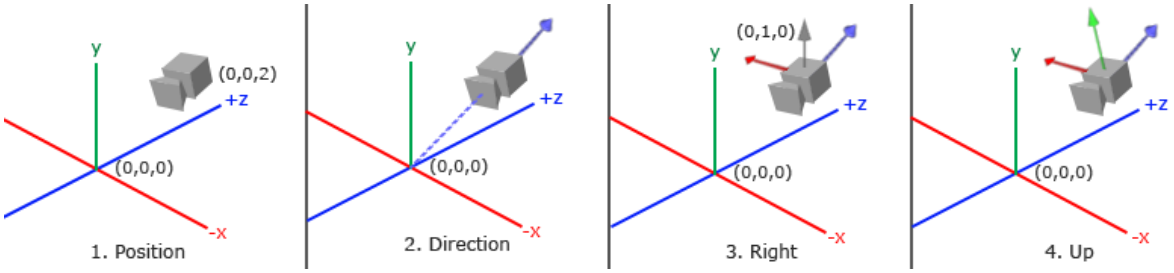

图 3-1 摄像机空间

摄像机类所持有的参数信息：

```
glm::vec3 Position;
glm::vec3 Front;
glm::vec3 Up;
glm::vec3 Right;
glm::vec3 WorldUp;
GLfloat Yaw;
GLfloat Pitch;
GLfloat MovementSpeed;
GLfloat MouseSensitivity;
GLfloat Zoom;
```

当我们创建好这个坐标空间后，就可以利用 glm 的 lookat 方法构建一个 view 矩阵，这个矩阵就是我们所需要的变换矩阵。

```
glm::lookAt(rightVec, posVec, upVec);
```

当我们需要用键盘鼠标控制摄像机移动时，需要向 opengl 注册鼠标键盘输入的回调函数，并在回调函数中处理我们的摄像机的相关参数。

```
void ProcessMouseMovement(GLfloat xoffset, GLfloat yoffset, GLboolean constrainPitch =
true)
{
    xoffset *= this->MouseSensitivity;
    yoffset *= this->MouseSensitivity;
```

```
        this->Yaw += xoffset;
        this->Pitch += yoffset;
        if (constrainPitch)
        {
            if (this->Pitch > 89.0f)
                this->Pitch = 89.0f;
            if (this->Pitch < -89.0f)
                this->Pitch = -89.0f;
        }
        this->updateCameraVectors();
    }

    void ProcessMouseScroll(GLfloat yoffset)
    {
        if (this->Zoom >= 1.0f && this->Zoom <= 45.0f)
            this->Zoom -= yoffset;
        if (this->Zoom <= 1.0f)
            this->Zoom = 1.0f;
        if (this->Zoom >= 45.0f)
            this->Zoom = 45.0f;
    }
```

当我们需要用到摄像机的 view 矩阵时只需要通过其暴漏在外部的接口调用即可。

```
    glm::mat4 GetViewMatrix()
    {
        return glm::lookAt(this->Position, this->Position + this->Front, this->Up);
    }
```

3.2 Shader 组件

着色器是运行在 GPU 上的小程序，这些程序是在图形渲染管线中的某些特定部分中运行的。着色器实际上是一种将输入转化为输出的程序。

Shader 组件实际上是对 opengl 中的着色器程序的封装，在 opengl 中我们可以提供着色器程序的源代码字符串，然后调用 opengl 的函数进行编译链接，通过我们的封装我可以直接给一个文本字符串给 shader 组件，然后它会自动帮我们读取字符串，编译链接着色器程序，而我们所需要的就是一个编译链接成功的着色器程序 id，当我们需要使用的时候，直接调用 glUseProgram(ID)即可。

同时，我们的 shader 组件还负责设置着色器程序中的 uniform 变量的值，使得我们的程序可以同着色器程序进行沟通。

我们的 shader 组件需要对顶点着色器和片段着色器负责，其中顶点着色器程序的输入是顶点属性，输出值是顶点位置，片段着色器程序的输入是上一阶段的输出，输出

值是片段的颜色。

我们的着色器组件的大致构成如下所示：

```
    unsigned int ID;
    Shader();
    Shader(std::string name, std::string vsCode, std::string fsCode,
std::vector<std::string> defines = std::vector<std::string>());
    void Load(std::string name, std::string vsCode, std::string fsCode,
std::vector<std::string> defines = std::vector<std::string>());
    void Use();
    bool HasUniform(std::string name);
    void SetInt(std::string location, int   value);
    void SetBool(std::string location, bool  value);
    void SetFloat(std::string location, float value);
    void SetVector(std::string location, glm::vec2  value);
    void SetVector(std::string location, glm::vec3  value);
    void SetVector(std::string location, glm::vec4  value);
    void SetVectorArray(std::string location, int size, const std::vector<glm::vec2>&
values);
    void SetVectorArray(std::string location, int size, const std::vector<glm::vec3>&
values);
    void SetVectorArray(std::string location, int size, const std::vector<glm::vec4>&
values);
    void SetMatrix(std::string location, glm::mat2 value);
    void SetMatrix(std::string location, glm::mat3 value);
    void SetMatrix(std::string location, glm::mat4 value);
```

3.3 Texture 组件

当我们在指定顶点属性时，可以指定顶点的 UV 坐标，这样我们就可以在纹理中根据纹理坐标进行采样，从而决定顶点的颜色。纹理是一张 2D 图片，用来添加物体的细节。

纹理坐标在 x-y 轴上，两个轴上的坐标范围被限定在 0-1 之间。

我们的 Texture 组件主要负责生成纹理，向纹理中传输纹理的像素数据，并可以设置纹理的环绕方式和纹理过滤的方式。

Texture 组件的大致构成如下所示：

```
    Texture();
    ~Texture();
    void Generate(unsigned int width, GLenum internalFormat, GLenum format, GLenum type,
void* data);
    void Generate(unsigned int width, unsigned int height, GLenum internalFormat, GLenum
format, GLenum type, void* data);
```

```
    void Generate(unsigned int width, unsigned int height, unsigned int depth, GLenum
internalFormat, GLenum format, GLenum type, void* data);
    void Resize(unsigned int width, unsigned int height = 0, unsigned int depth = 0);
    void Bind(int unit = -1);
    void Unbind();
    void SetWrapMode(GLenum wrapMode, bool bind = false);
    void SetFilterMin(GLenum filter, bool bind = false);
    void SetFilterMax(GLenum filter, bool bind = false);
```

3.4 Mesh 组件

Mesh 组件是 MongoEnigne 比较重要的一个组件，我们通过 mesh 组件向 GPU 上传输顶点数据，其中 mesh 类有几个子类，是 MongoEnigne 中自带的一些常用网格。

1. CircleMesh：圆形平面的网格类；
2. CubeMesh：长方体网格类；
3. PlaneMesh:平面网格类；
4. QuardMesh：面片网格类；
5. SphereMesh：球体网格类；

这里我们以比较复杂的球体网格类的实现为例：

SphereMesh 中我们需要生成球体上面的每个点的坐标，用到的是球的参数方程。

$$\begin{cases} x = R\sin\varphi\cos\theta \\ y = R\sin\varphi\sin\theta \\ z = R\cos\varphi \end{cases}$$

$(0\le\theta\le 2\pi,\ 0\le\varphi\le\pi)$

图 3-2 球体的参数方程

核心代码实现为：

```
Sphere::Sphere(unsigned int xSegments, unsigned int ySegments)
{
    Name = "sphereMesh";
    for (unsigned int y = 0; y <= ySegments; ++y)
    {
        for (unsigned int x = 0; x <= xSegments; ++x)
        {
            float xSegment = (float)x / (float)ySegments;
            float ySegment = (float)y / (float)ySegments;
            float xPos = std::cos(xSegment * TAU) * std::sin(ySegment * PI); // TAU is 2PI
            float yPos = std::cos(ySegment * PI);
```

```
            float zPos = std::sin(xSegment * TAU) * std::sin(ySegment * PI);

            Positions.push_back(glm::vec3(xPos, yPos, zPos));
            UV.push_back(glm::vec2(xSegment, ySegment));
            Normals.push_back(glm::vec3(xPos, yPos, zPos));
        }
    }
    bool oddRow = false;
    for (int y = 0; y < ySegments; ++y)
    {
        for (int x = 0; x < xSegments; ++x)
        {
            Indices.push_back((y + 1) * (xSegments + 1) + x);
            Indices.push_back(y       * (xSegments + 1) + x);
            Indices.push_back(y       * (xSegments + 1) + x + 1);
            Indices.push_back((y + 1) * (xSegments + 1) + x);
            Indices.push_back(y       * (xSegments + 1) + x + 1);
            Indices.push_back((y + 1) * (xSegments + 1) + x + 1);
        }
    }
    Topology = TRIANGLES;
    Finalize();
}
```

3.5 Material 组件

在现实世界中物体对光会产生不同的反应。我们在 opengl 中来模拟不同的物体时就需要为每个物体指定一个材质属性。

Material 组件，实际上是对 Shader 组件的封装，在这里面我们会有对 shader 中的 uniform 变量设置的接口，同时为了效率，当我们设置非纹理类型的变量时，我们不直接设置 shader 中的变量，而是先将变量名和变量值通过 map 储存起来，当我们使用的时候再同统一向 shder 中设置对应的值。

并且我们在 Material 中还提供了一个 Copy 接口，通过这个接口可以返回一个深度拷贝的材质副本。

Material 的大致构成如下所示：

```
    Shader * m_Shader;
    std::map<std::string, UniformValue>        m_Uniforms;
    std::map<std::string, UniformValueSampler> m_SamplerUniforms;
    void Use();
    Material();
    Material(Shader* shader);
```

```
    Shader* GetShader();
    void    SetShader(Shader* shader);
    Material Copy();
    void SetBool(std::string name, bool value);
    void SetInt(std::string name, int value);
    void SetFloat(std::string name, float value);
    void SetTexture(std::string name, Texture* value, unsigned int unit = 0);
    void SetTextureCube(std::string name, TextureCube* value, unsigned int unit = 0);
    void SetVector(std::string name, glm::vec2 value);
    void SetVector(std::string name, glm::vec3 value);
    void SetVector(std::string name, glm::vec4 value);
    void SetMatrix(std::string name, glm::mat2 value);
    void SetMatrix(std::string name, glm::mat3 value);
    void SetMatrix(std::string name, glm::mat4 value);
```

3.6 Loader 组件

Loader 组件是 MongoEnigne 的资源加载器，负责加载我们需要用的资源，主要由 MeshLoader、ShaderLoader、TextureLoader 三大部分构成。

MeshLoader

MongoEngine 中自带的 mesh 我们可以直接实例化得到，在它的构造函数中我们就填充了它的顶点数据，而 MongoEnigne 可以加载其他建模软件做好的模型文件 obj 文件，所以我们需要用 MeshLoader 来加载解析这样的 obj 文件。其中用到的库是 assimp。

在模型的加载过程中，我们直接给出模型的文件名，然后通过 assimp 库加载出来，并且由于 obj 文件中已经包含了材质信息，所以我们通过 meshLoader 中加载出来的就是一个完整的 SceneNode 了，并且默认的父亲节点就是场景根节点。

MeshLoader 组件的大致构成如下所示：

```
    static std::map<std::string, Mesh*> meshMap;
    static std::map<std::string, Material*> matMap;
    static SceneNode* LoadMesh(std::string path, bool setDefaultMaterial = true);
    static SceneNode* processNode(aiNode* aNode, const aiScene* aScene, std::string
directory, bool setDefaultMaterial);
    static Mesh*      parseMesh(aiMesh* aMesh, const aiScene* aScene, glm::vec3& out_Min,
glm::vec3& out_Max);
    static Material*  parseMaterial(aiMaterial* aMaterial, const aiScene* aScene,
std::string directory);
    static std::string processPath(aiString* path, std::string directory);
```

3.7 Light 组件

Light 组件实际上是对光源的抽象，目前 MogonEngine 中支持三种光源，分别是平行光，点光源，聚光。

平行光：

当一个光源位于很远的地方的时候，来自光源的每条光线我们就可以近似地把它们看作是平行的，不论观察者站在任何地方，看起来光线都来自同一个方向。

平行光的一个典型例子就是太阳光，太阳离我们并不是无限远，但我们在计算光照的时候可以把它视作为无限远。

对平行光的抽象如下所示：

```
class DirectionalLight
{
public:
    glm::vec3 Direction = glm::vec3(1, -1.0, 1);
    glm::vec3 Color = glm::vec3(1.0f);
    float Intensity = 1.0f;
};
```

主要是光的方向和光的颜色，光的强度是为了让我们调节光的颜色。

当我们在对平行光进行光照计算的时候，漫反射的时候我们需要会用视线的方向和平面法线的夹角的余弦值乘以光的颜色来作为漫反射的颜色。

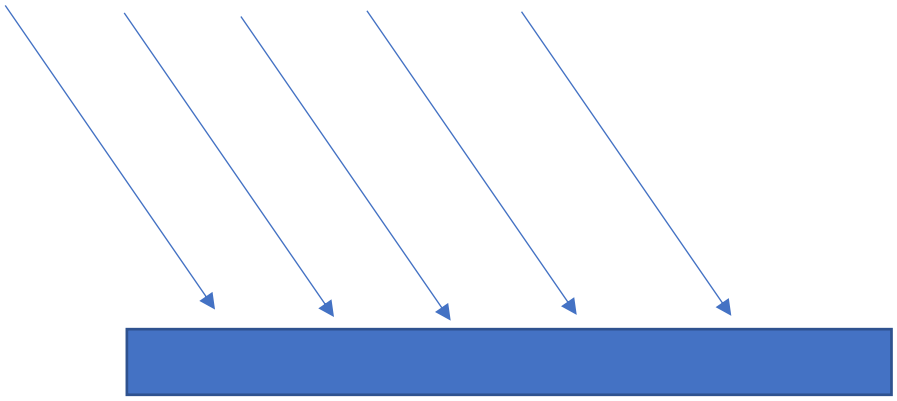

图 3-3 平行光

1. 点光源

点光源是处于世界中某一个位置的光源，它会朝着所有方向发光，但光线会随着距离逐渐衰减。随距离减少光强度的一种方式是使用一个线性方程，这样的一个方程可以让光的强度线性的随着距离的增长而减少，使得远处的物体要更暗，但这样的方程看起来会比较假。

在现实世界中灯照射时，近处一般会非常亮，但随着距离的增加光源的亮度一开始会下降的特别快，但在远处下降的就会非常缓慢，MongoEngine 中使用的是下面这样的一个公式：

$$F_{att} = \frac{1.0}{K_c + K_l * d + K_q * d^2}$$

其中 d 表示片段距离光源的距离；

常数项一般保持为 1，它的作用是为了让分母永远不会比 1 小；

一次项与距离值相乘，以线性的方式减少光的强度；

二次项会和距离的平方相乘，让光的强度以二次递减的方式减少。

当我们抽象一个点光源的时候，我们主要关注它的位置和颜色，同样强度参数是为了让我们更方便地控制光的强度。

抽象如下所示：

```
class PointLight
{
public:
    glm::vec3 Position = glm::vec3(0.0f);
    glm::vec3 Color = glm::vec3(1.0f);
    float      Intensity = 1.0f;
};
```

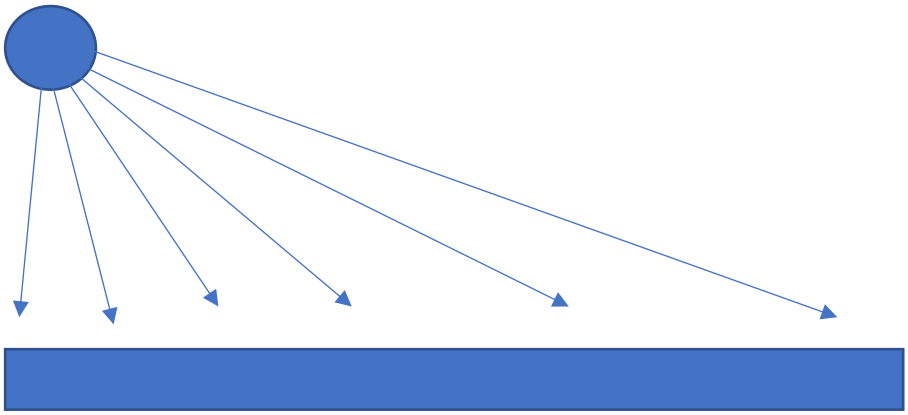

图 3-4 点光源

2. 聚光

聚光是处于世界某个位置的一个光源，它只朝一个特定的方向而不是所有的放心投射光线，所以只有在聚光方向的特定半径内的物体才会被照亮，现实中的聚光例子就是手电筒。

当我们在抽象聚光的时候，我们需要一个空间位置、一个方向、一个切光角。对于一个片段，我们计算它是否在聚光的切光方向上，如果是的话，我们就会照亮相应的片

段。

聚光的抽象如下所示：

```
class SpotLight
{
public:
    glm::vec3 Position = glm::vec3(0, 1, 0);
    glm::vec3 Direction = glm::vec3(0, -1, 0);
    glm::vec3 Color = glm::vec3(1.0f);
    float     Intensity = 1.0f;
    float     CutOff = glm::cos(glm::radians(12.5f));
    float     OuterCutOff = glm::cos(glm::radians(17.5f));
    bool      RenderMesh = false;
};
```

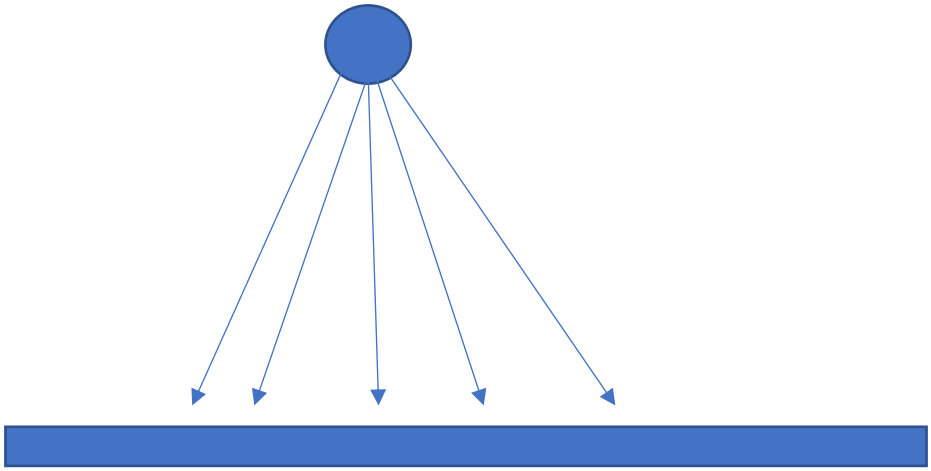

图 3-5 聚光

3.8 Scene 组件

Scene 组件是 MongoEnigne 中的核心组件，其持有了整个场景的场景树，在场景中每个物体都被抽象为一个 SceneNode，由 Scene 持有整个场景树的根节点，并且 Scene 组件持有所有的光源的引用，场景的节点树结构如下图所示：

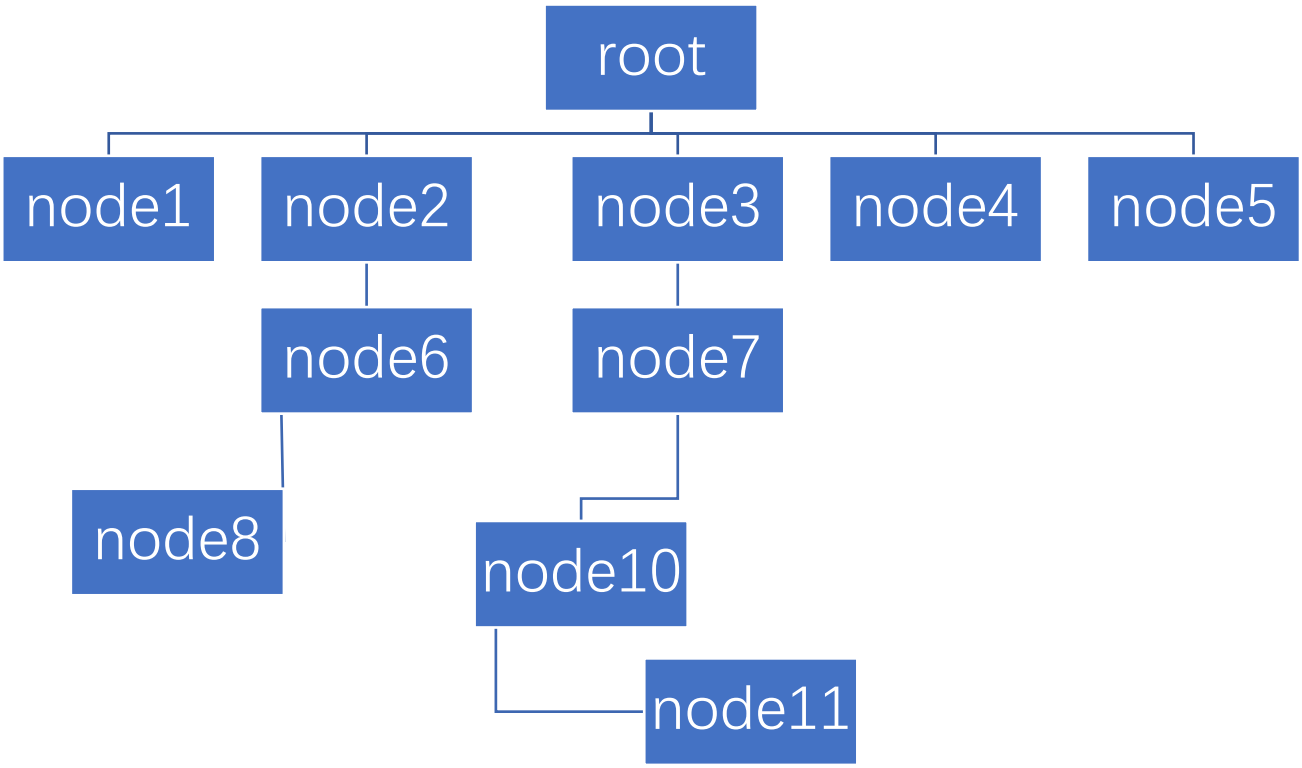

图 3-6 场景树

当我们实例化 SceneNode 后，就可以将它加入到场景树中，将它置为场景中某个节点的子物体即可。

深度遍历的核心代码如下：

```
void Scene::render(SceneNode* node)
{
    if (node != nullptr)
    {
        for (int i = 0; i < node->GetChildCount(); i++)
        {
            if (node->GetChildByIndex(i) != nullptr)
            {
                node->GetChildByIndex(i)->Render();
            }
            render(node->GetChildByIndex(i));
        }
    }
}
```

当我们渲染场景时，会以深度优先遍历的方式遍历整个场景树，当访问到每个节点的时候，我们会调用该节点的 render 接口，渲染该节点。

同时在每个渲染循环中我们都会先更新 Uniform 缓冲区张的数据，然后渲染每个节点，渲染灯光，最后渲染天空盒。

Scene 组件作为一个全局单例类，负责管理场景中所有的实体，包括物体节点，灯光，相机，声音播放工具。

3.9 其他组件

为了 MongoEnigne 更加有趣，我们在里面添加了文本渲染组件和声音播放组件。

文本渲染我们用的是 freetype 库，FreeType 是一个完全开源的、可扩展、可定制且可移植的字体引擎，它提供 TrueType 字体驱动的实现统一的接口来访问多种字体格式文件，包括点阵字、TrueType、OpenType、Type1、CID、CFF、Windows FON/FNT、X11 PCF 等。

声音播放我们用的是 IrrKlang，IrrKlang 是一个可以播放 WAV，MP3，OGG 和 FLAC 文件的高级二维和三维（Windows，Mac OS X，Linux）声音引擎和音频库。它还有一些可以自由调整的音频效果，如混响、延迟和失真。

4. 实现效果

1.纹理加载

加载纹理的效果如图 4-1 所示：

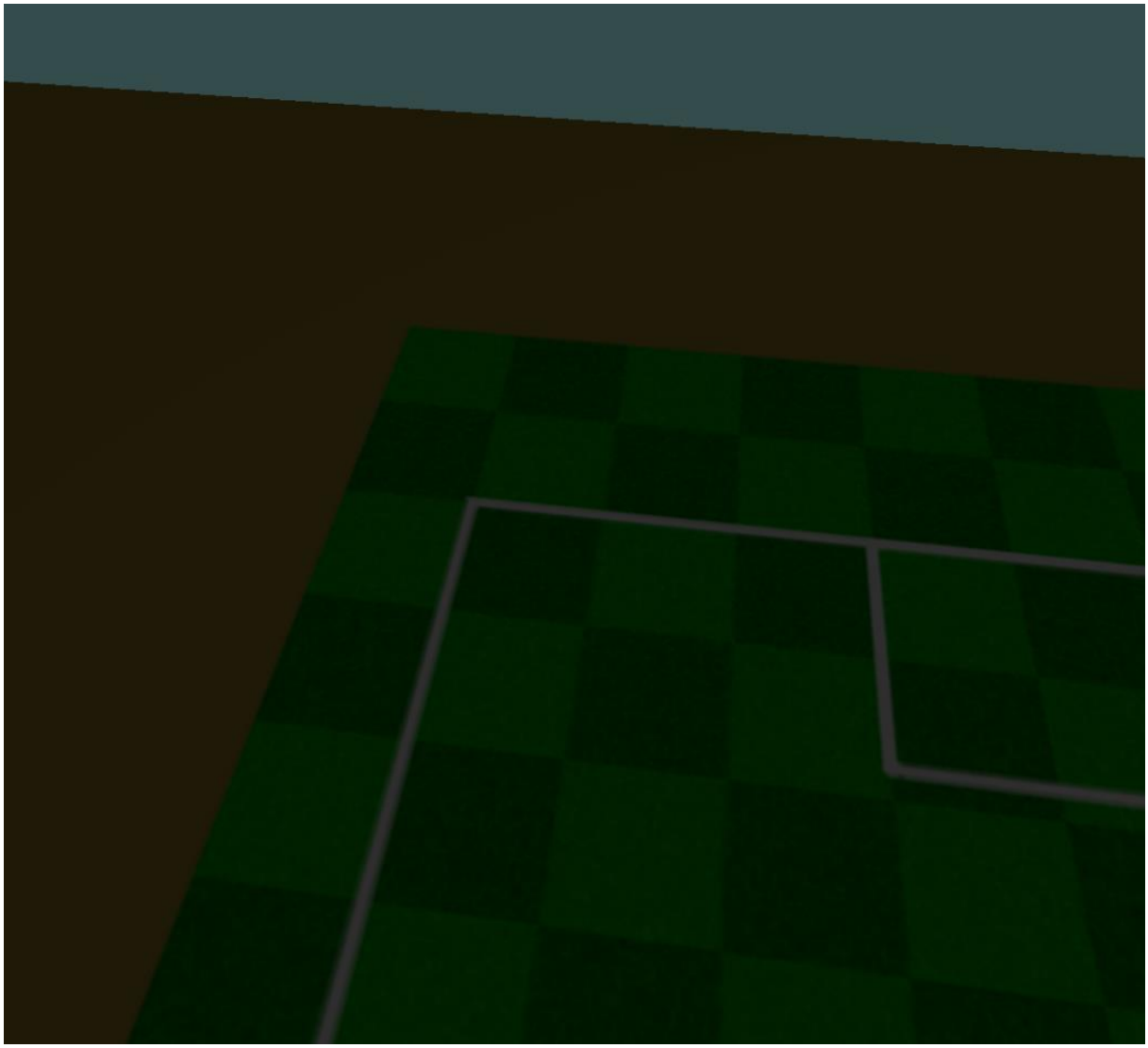

图 4-1 纹理加载

2.三种光源

平行光的效果如图 4-2 所示:

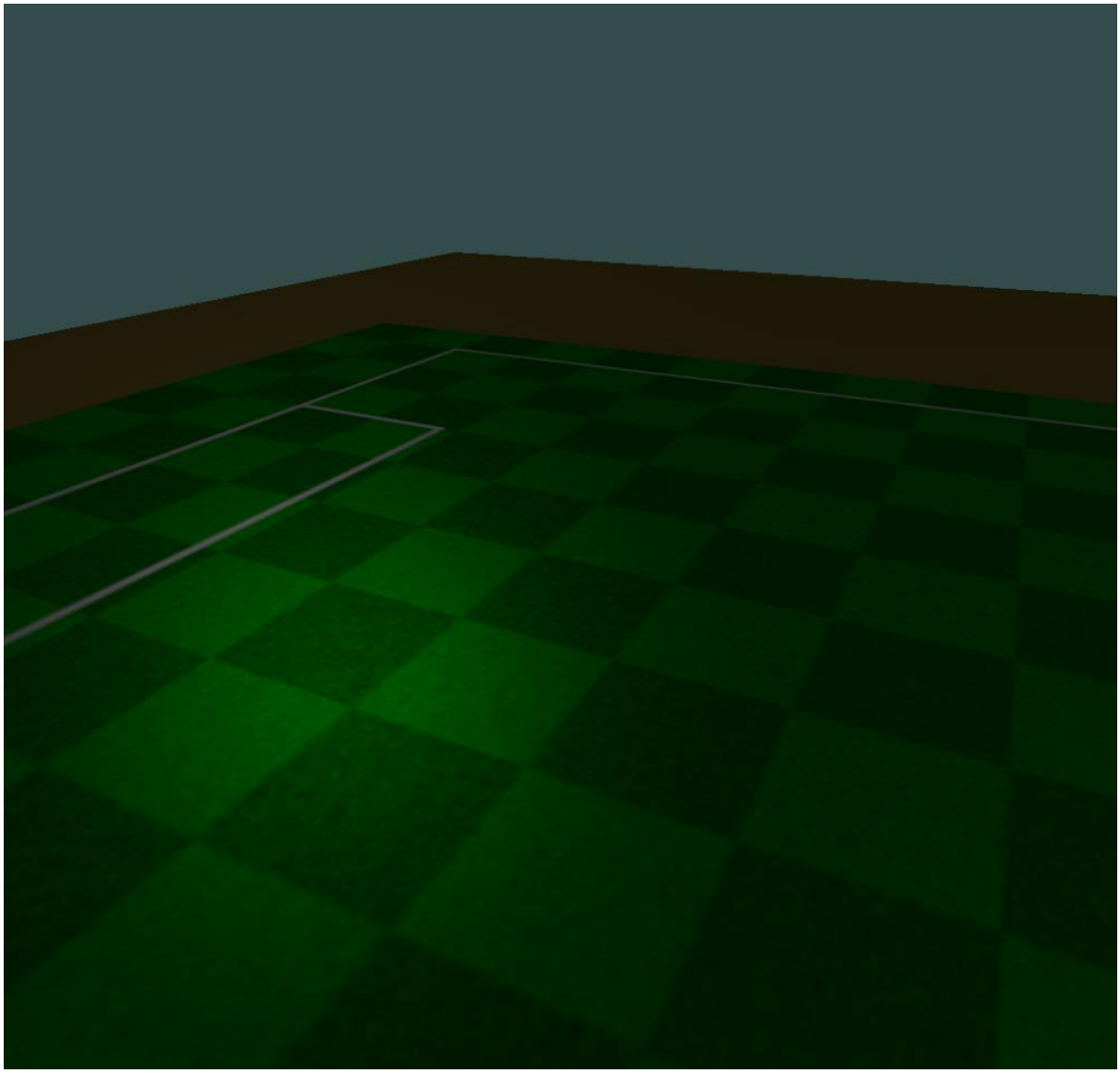

图 4-2 平行光

点光源的效果如图 4-3 所示：

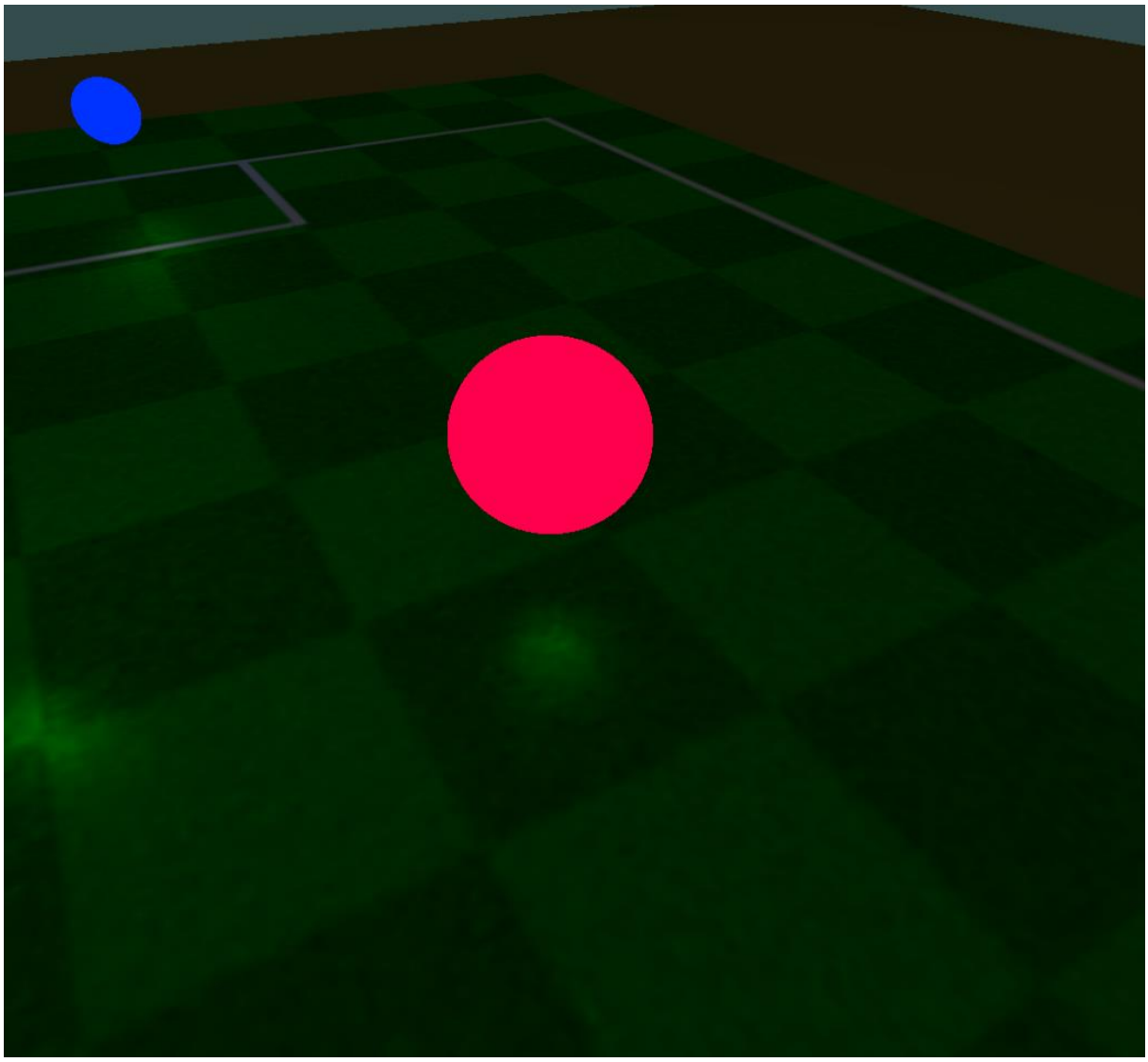

图 4-3 点光源

聚光的效果如图 4-4 所示：

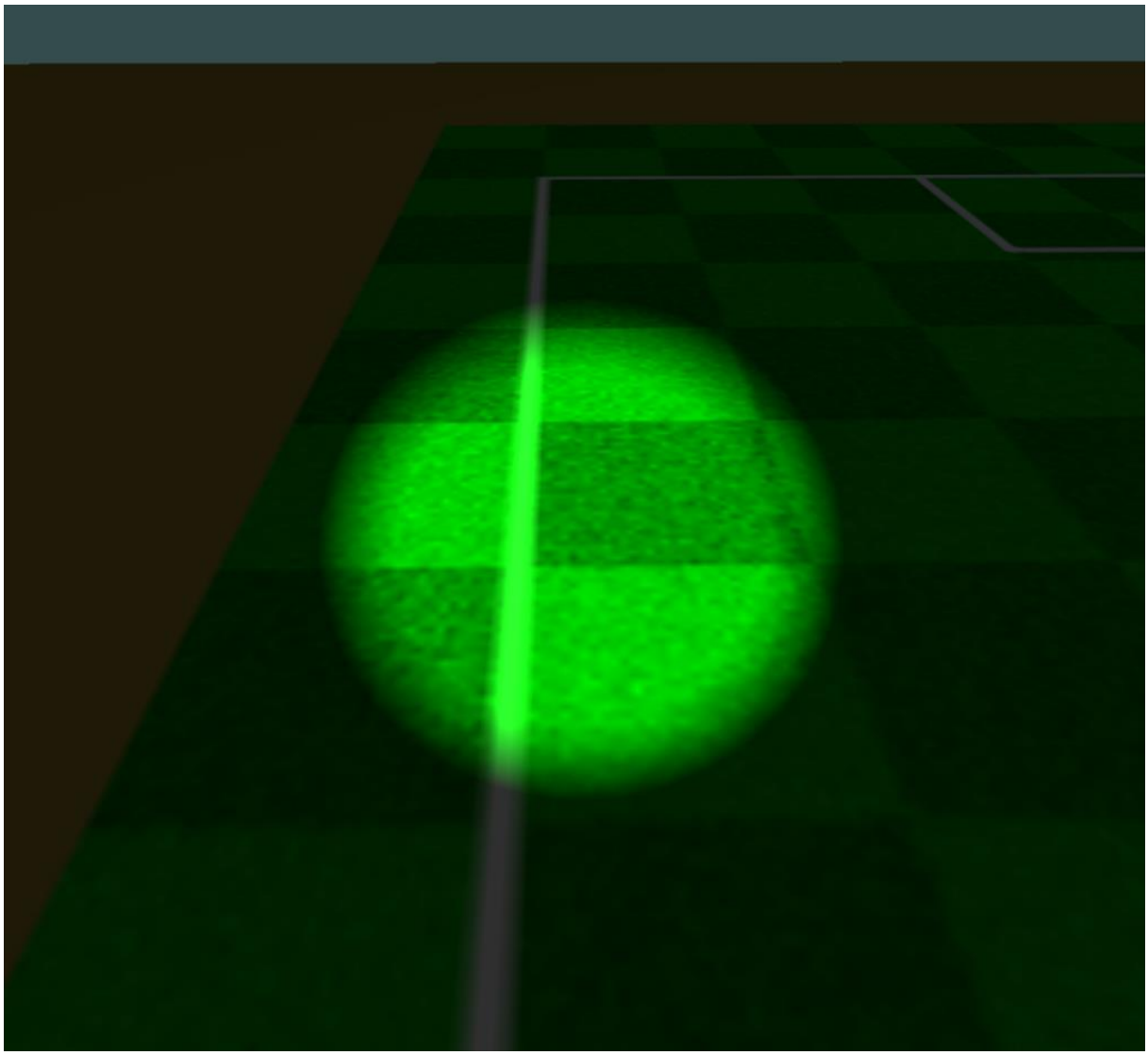

图 4-4 聚光的效果

3.模型加载

加载简单的立方体块的效果如图 4-5 所示;

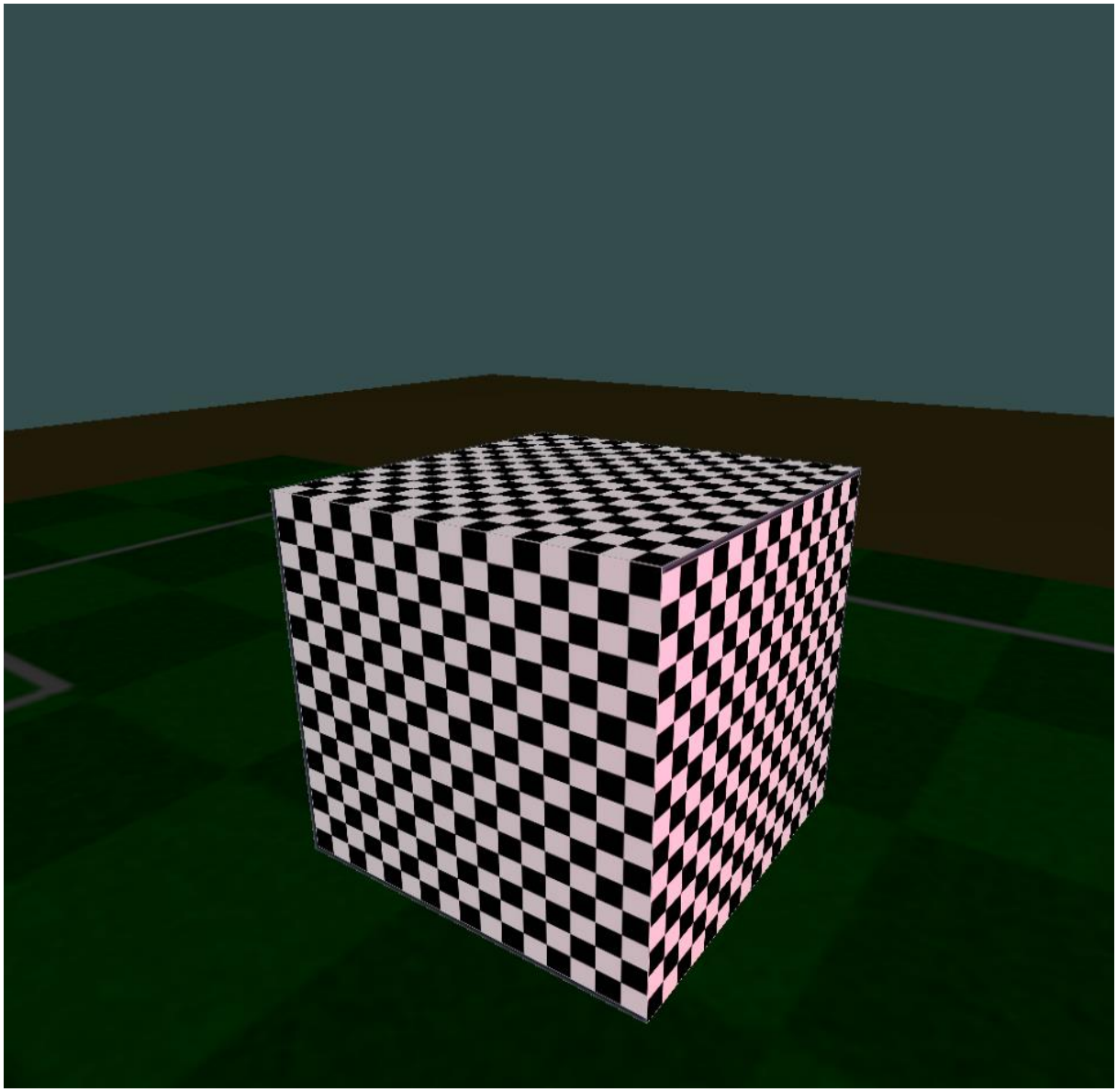

图 4-5 简单的立方体块

加载箱子模型的效果如图 4-6 所示：

图 4-6 箱子模型

加载复杂的 obj 模型效果如图 4-7 所示：

图 4-7 复杂的 obj 模型

4. 天空盒

加载天空盒子的效果如图 4-8 所示：

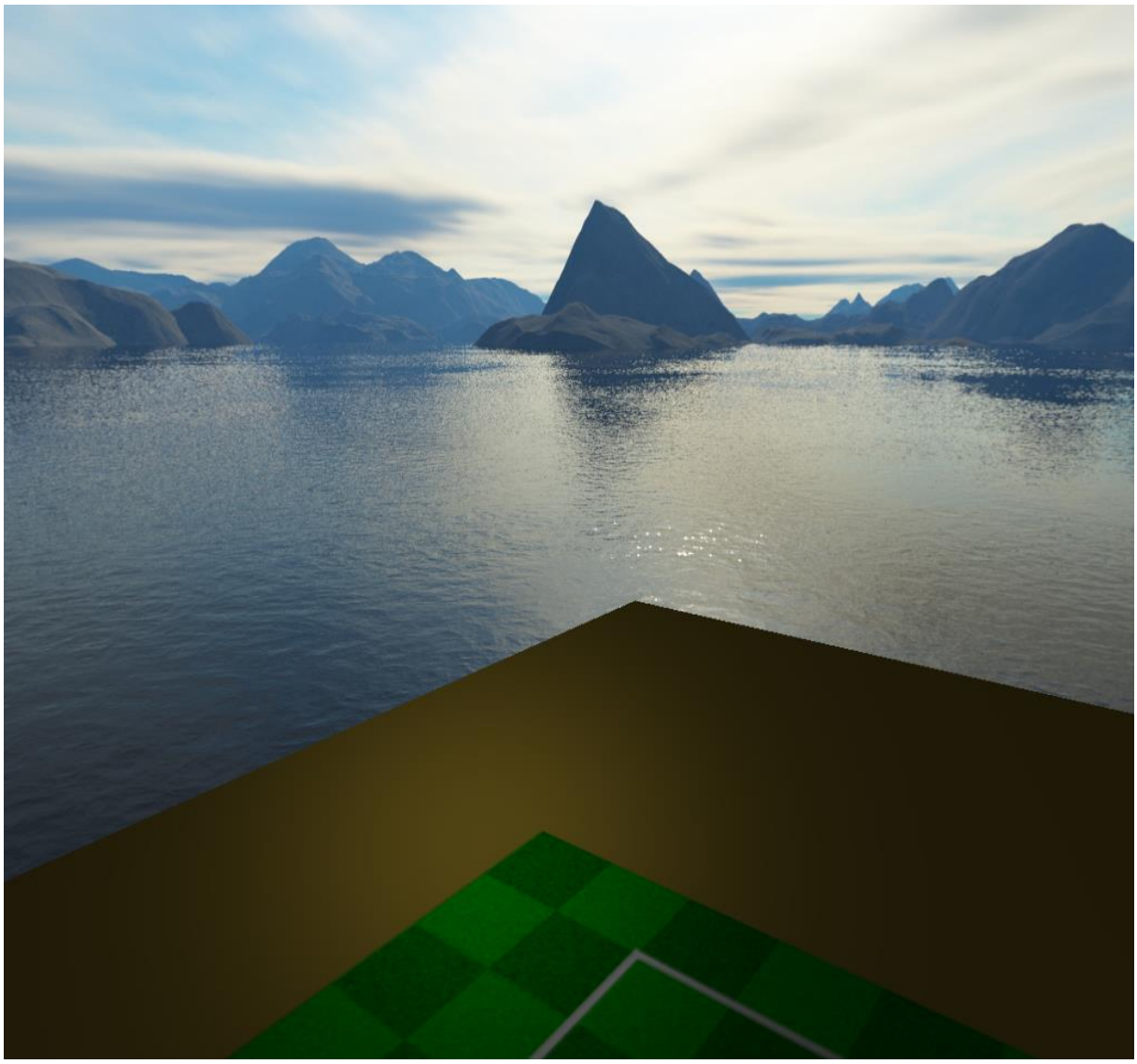

图 4-8 天空盒效果

5.屏幕特效

颜色反向的屏幕特效如图 4-9 所示：

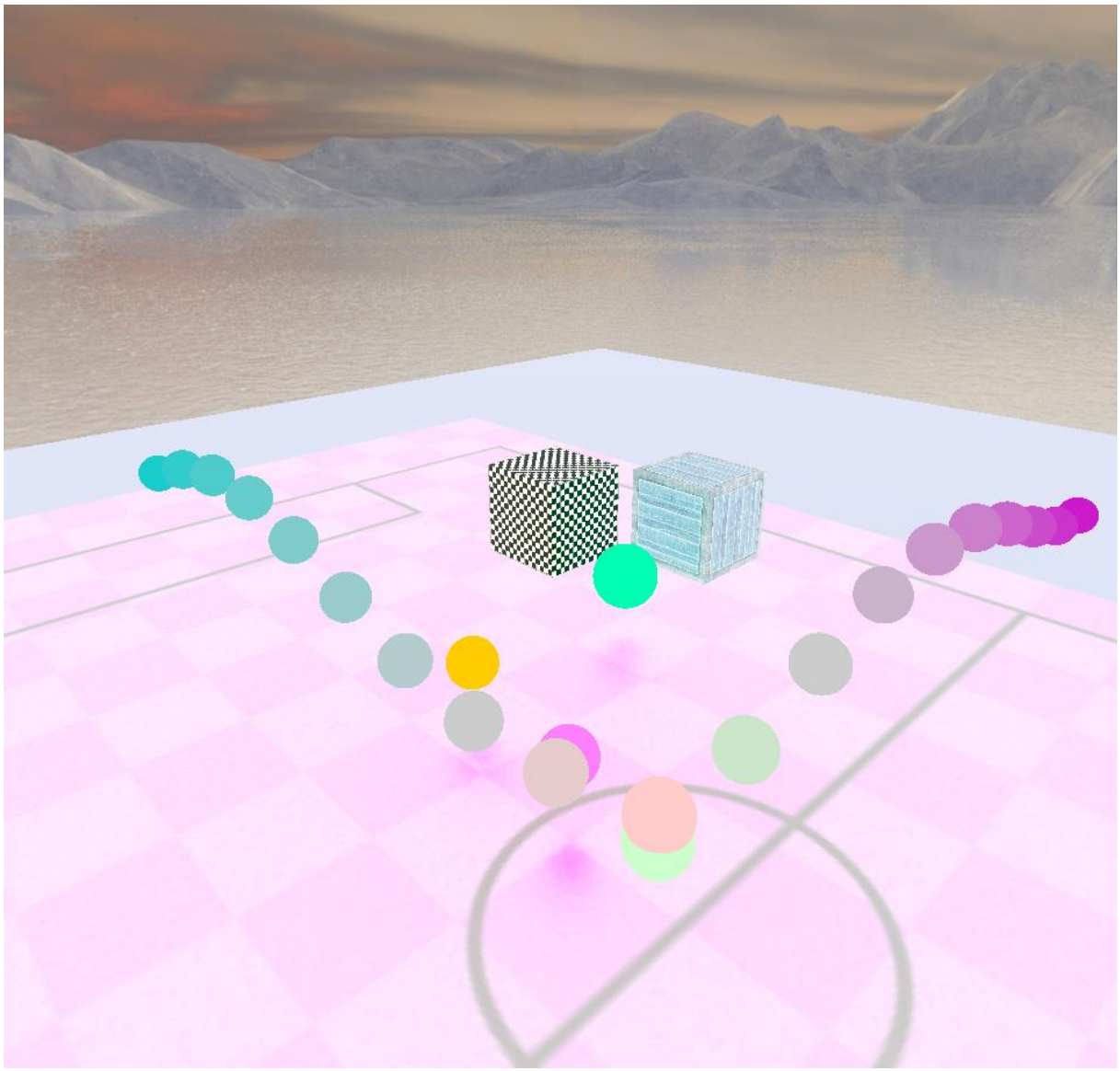

图 4-9 颜色反向特效

灰度特效如图 4-10 所示：

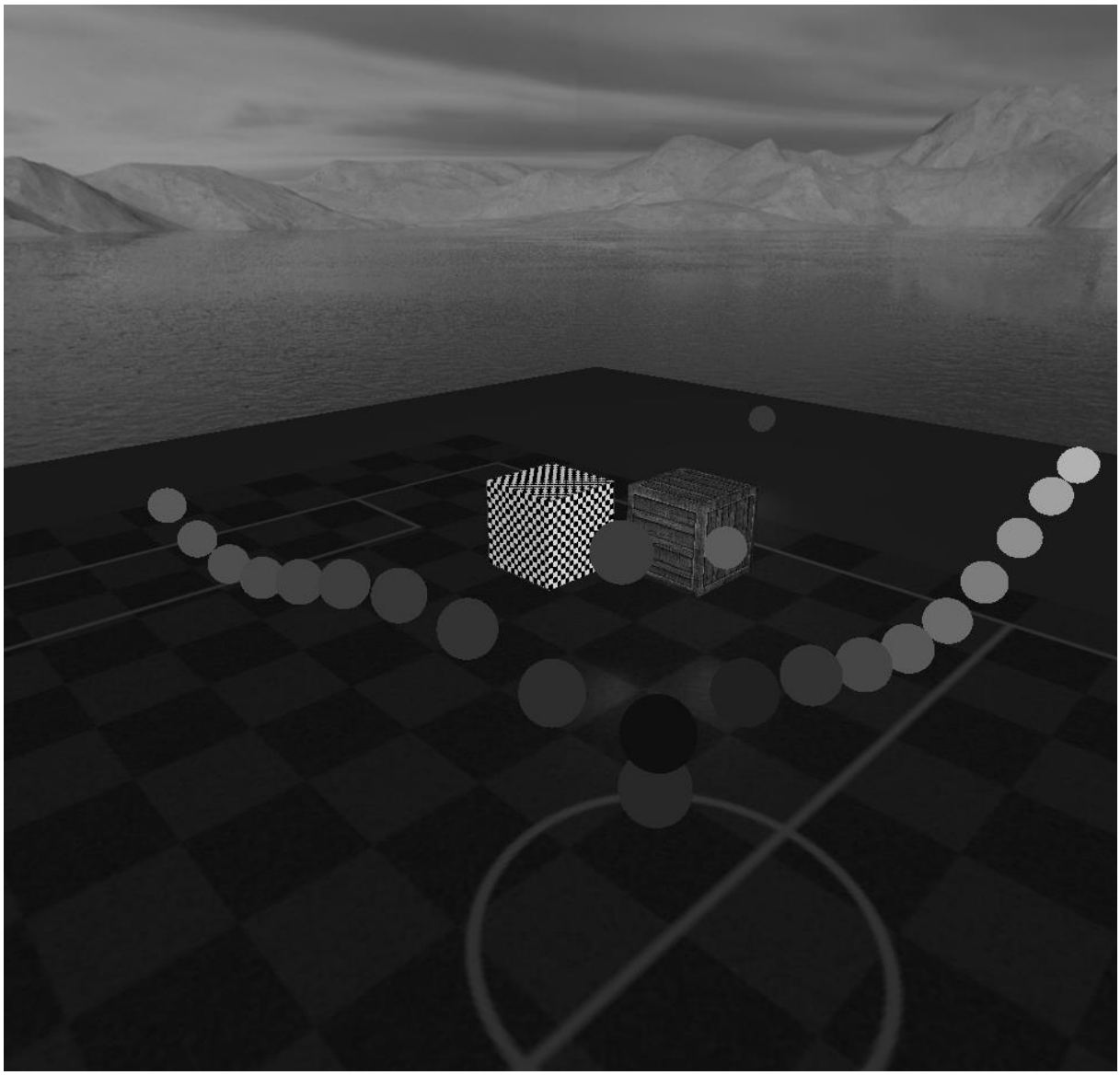

图 4-10 灰度特效

模糊效果特效如图 4-11 所示：

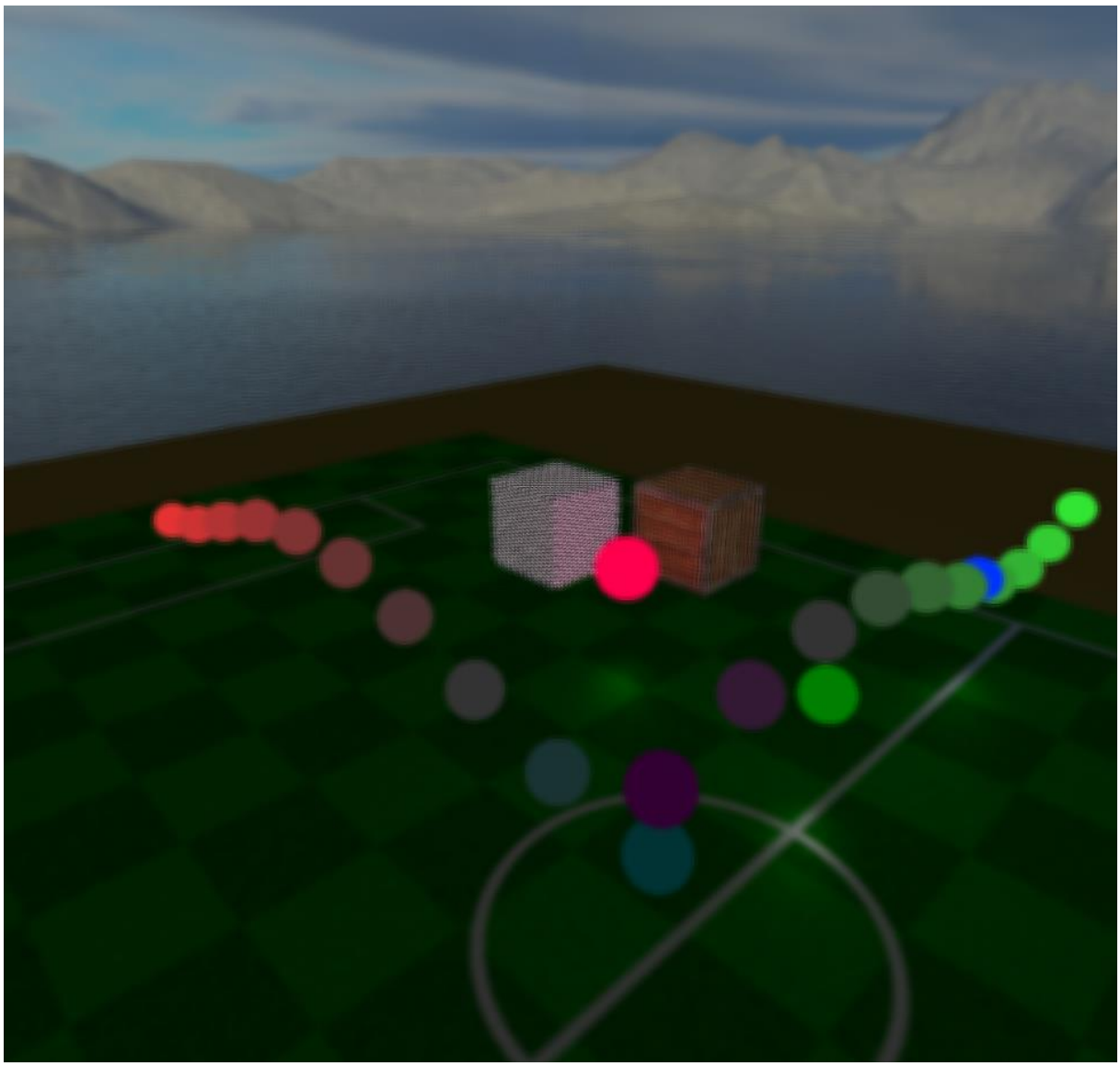

图 4-11 模糊效果特效

边缘检测特效如图 4-12 所示：

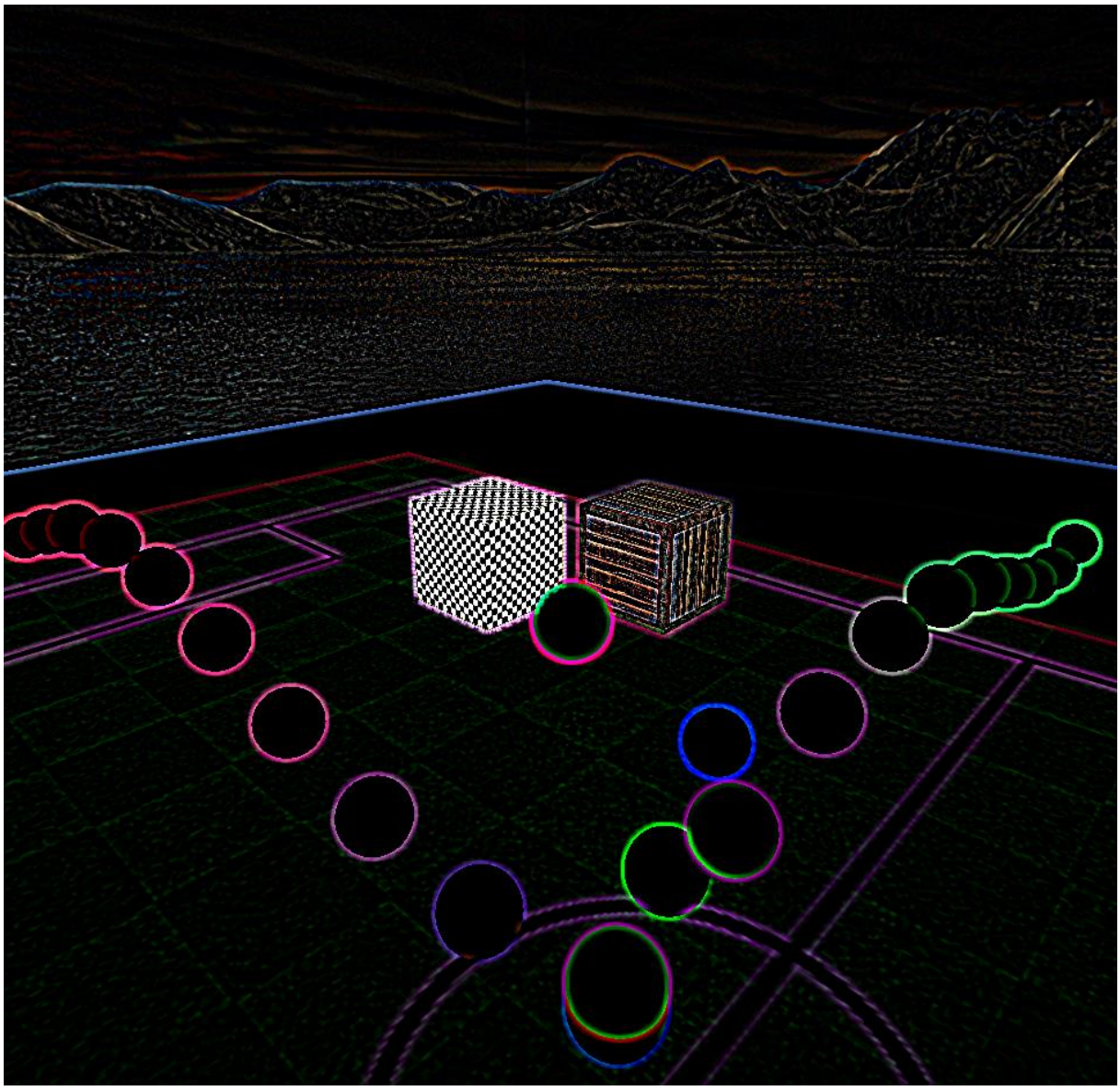

图 4-12 边缘检测特效

5. 总结

通过本次毕业设计，我将之前所学的知识融会贯通了起来，并且在查找资料的过程中，学习到了更多的编程和图形学方面的知识，了解到了计算机图形学的发展方向，自己的专业知识和动手能力都有了一个质的飞跃。

毕业设计作为我们大学的最后一次作业，需要我们在上面投入很多的精力和时间。毕业设计是对我整个大学学到的知识的一次总结，检验了我的独立解决问题的能力。

万事开头难，其实刚开始准备实现 MongoEnigne 的时候，不知道应该怎么入手，因为一个图形渲染引擎还是有一定的复杂度的，如果不能架构的很清楚的话，会导致开发进程的缓慢，甚至可能需要推倒重来。所有我们在刚开始的时候，没有着急去写代码，而是参考市场上现有的一些引擎的架构，比如 unity3d 引擎,egret 白鹭引擎，在参考的过程中，我们可以比较容易的抽象出来一个渲染引擎需要的模块，然后对 MongiEnigne 进行架构设计，并在开发过程中不断迭代，设计符合我们的引擎的架构方式。

在 MongoEnigne 的整个开发过程中，我们需要用到数学知识，特别是线性代数的知识，它在我们的矩阵变换中发挥着巨大的作用，需要用到物理知识，在光照模型中，我们是在对物理模型进行抽象，需要用到图形学的知识，MongoEnigne 整个实现的理论基础就是图形学的知识，需要用到 C++编程语言的知识，不仅仅是基本的语法，还有动态库、静态库、头文件的配置链接的方法。

MongoEnigne 目前的功能相对来说并不是很强大，仅仅实现了渲染引擎的部分基本功能，所以我把它称之为一个简单的渲染引擎，目前主要实现了物体顶点坐标变换，Phong 氏光照模型，三种光源的光照效果（平行光、点光、聚光），obj 模型的加载，场景树的构建，引擎资源管理器。

MongoEnigne 还有比较大的提升空间，有以下几点可以接着迭代开发增加到引擎中去：

1.阴影的添加。阴影是光线被阻挡的结果，可以使场景看起来变得更加的真实。阴影还是比较不好实现的，因为目前渲染领域还没有找到一个完美的阴影渲染算法。常用的技术就是阴影贴图，MongoEnigne 在后续会增加这个技术。

2.脚本功能的增加。可以参考 unity3d 的设计,实现每个物体都可以挂载一些脚本，脚本具有生命周期，这样我们可以更方便的单独控制每个物体，代码耦合性大大降低。

3.光线追踪技术的增加。光线追踪技术通过对光线路径的追踪，较大程度上去模拟真实场景中的光照情况，可以渲染出非常具有真实感的场景，MongoENigne 后续考虑加入离线渲染功能，就会用到这个技术。

4.场景数据的持有化。我们可以将现有场景以文件的形式储存起来，这样我们就可以更加方便的增加场景切换功能。

5.封装引擎功能由.Net 层调用。我们可以将 MongoEnigne 的功能封装成不同接口，打包变成一个动态链接库，用.Net 平台上的 C#语言来调用，这样我们就可以在.Net 平台上做架构，充分利用这个平台上的各种现有库，强大 MongoEnigne 的功能。

总之，MongoEnigne 还是具有比较大的发展空间的，需要我们不断的学习新技术，不断增强我们的学习能力，才能不断地将我们的引擎完善起来。

在这次毕业设计的整个过程中，我学习到了很多的东西，也充分地发挥了自己学习过的知识，培养了我独立思考，独立工作的能力，大大地提高了我的动手能力，也让我充分地体会到了创造的快乐和喜悦。在整个设计过程中所学到的东西是最大的收获和财富，这让我终身受益。

参考文献

[1] WolfganEEngel. Direct3D 游戏编程入门教程. 北京：人民邮电出版社，2002.7
[2] 张继开. 三维图形引擎技术的研究. [硕士学位论文]. 北京：北方工业大学，2004
[3]乔林，费广正. OpenGL 程序设计. 北京：人民邮电出版社，2000：61-367
[4]傅晟，彭群生. 一个桌面型虚拟建筑环境实时漫游系统的设计与实现，计算机学报， 1998,21(9)：793-799
[5]祁利文，韩崇昭. 一种新的基于环境纹理映射的浓淡方法，中国图象图形学报，2000. 5C2)： 124—127
[6] 毛玉姣，王梅，陈远. 虚拟现实技术及其应用，图书情报知识. 1997. 12
[7]汪成为，高文，王行仁. 灵境(虚拟现实)技术的理论、实现及应用. 北京：清华大学出版 社，1996. 1-243
[8] 蒋燕萍. 虚拟环境漫游中的关键技术：[硕士论文]，北京：北方工业大学，2003
[9]Mason Woo,Jackie Neider,Tom Davis etc. The Official Guide to Learning OpenGLVersion 12 北 京：中国电力出版社，2001. 133-398
[10]付凯. 基于可编程图形硬件的实时图形技术研究：[硕士学位论文]，武汉：武汉理工大学， 2005
[11]Donald Heam,M. Pauline Baker. 计算机图形学(蔡士杰，吴春，孙正兴等译). 北京：电子工业 出版杜，1998. 302-320，462-465
[12]Fletcher Dunn，Ian Parberry. 3D math primer for graphics and game development. Texas：Wordware Publishing,2002. 1-67，83-193
[13]J. MailoLetc. Intcractive texture mapplng. 1n Computer Graphics,J. T. Kajiya,Ed，2002(27)
[14]S. Mafoank,O D Faugeras. A Theory of Self2cah'bratinn of a Moving camera[S]. IntJoumal of Computer Vision,1992. 8(2)：123—151
[15]O D Faugeras。Q T Luong,S J Maybank. Camera Self cah'bration：Theory and Experiments[C]. Proc of the 2rid European Condon Comp. Vision,LecNotes in Comp. Science 588. 1992. 321-334
[16]J,A. Ferwerda,s. N. Pattanaik,EShirley,D. P. Greenberg. A Model of Visual Masking for Computer Graphics. In Proceedings of SIGGRAPH 97,Computer Graphics Proceedings，Annual Conference Series,1997. 2(8)
[17]N. Grecne and M. Kass. Hiemchical Z-Bttffer Visibility. In Proceedings of SIGGRAPH93, Computer Graphics ProceedingsAnnual Confcrencs

[18]Franklin C. Crow. Shadow-algorithms for computer graphics. Proceedings of the 4th annual conference on Computer graphics and interactive techniques. San Jose,California,1977. 242-248
[19]Jukka Kainulainen. Stencil Shdadow Volumes. Helsinki University ofTechnology,2002. 4-15
[20]ASsARSSON U,AKENINE-MOLLER T. A Geometry-based Soft Shadow Volume创90riⅡm using Graphics Hardware. ACM Transactions on Graphics, 2003(22): 511-520
[21]David也Ededy. 3D Game E丑gine Design: a Practical Approach to Realtime Computer Graph.PRENTICE HALLPublishing,2000. 1—121
[22]Akeley,K Reality Engine Graphies. In Computer Graphics(SIGGRAPH’93 Promxlings),1993.(3): 109-116
[23]SegalJ C. Korobkin,R. van Widenfelt,J. Foran,EHaeberlL Fast shadows effects using texture mapping and lighting,In Computer Graphics(SIGGRAPH92 Proceedings),1992. (2): 249—252 [24]J. H. Clark,Hierarchical geometric models for visibleⅡ algofithms,Communications of theACM,1976. (19): 547-554
[25]Rubin S M. There presentation and display of scenes with a wide range of detail. The Graphics and Image Processing,1982. (1外: 291-299
[26]J. Schroeder,J. A. Zarge, W. E. Lorenson. Decimation of triangle meshes. Computer Graphics, 1992. (2): 65-70
[27]GTurt: Re-tiling polygonal surfaces. Computer Graphics, 1992. (2): 55-64

致谢

在此毕业论文完成之际，我首先要向导师程建老师致以衷心的感谢!感谢程建老师的帮助，程老师深厚的专业素养和认真的人生态度深深感染了我。

感谢我的同学，他们在我遇到困难的时候所给予的无私帮助，同学之间宽松、愉快的共事经历将永远留在我心底，并激励我在今后的学习和生活中更加积极乐观地面对一切困难和挑战。

谢谢我的家人和好友，他们的关爱和支持永远是我前进的最大动力。在任何时候，他们都给予我最大的鼓励和支持。感谢他们!