学号	姓名	论文规范性（10）	问题分析与调研（30）	方案创新性（20）	实验结果分析与讨论（40）	结课论文总成绩（100）
21301049	王美靖	8	24	16	35	83

计算机图形学课程论文

——计算机图形学中的光线追踪技术的实现

学　院：　软件学院

学号：　21301049

学生姓名：　王美靖

指导教师：　吴雨婷

北京交通大学

2024 年 6 月

中文摘要

光线追踪技术[1]是计算机图形学领域的重要分支，其通过模拟光的物理传播路径产生逼真的阴影、反射、折射、散射等效果。最初的光线追踪方法主要集中在静态场景的渲染上，随着计算机硬件性能的提升和算法的不断优化，光线追踪技术得到了显著的发展，逐渐扩展到动态场景和实时渲染领域。随着深度学习的兴起，光线追踪技术与机器学习的结合也成为当前的研究热点。光线追踪技术应用广泛，涵盖了电影制作、游戏开发、虚拟现实、增强现实、医学图像处理等多个领域。

总之，光线追踪技术以其强大的模拟能力和逼真的视觉效果，成为计算机图形学领域中不可或缺的一部分，并不断推动着图形渲染技术的发展和创新。

关键词：光线追踪技术

ABSTRACT

Ray tracing technology is an important branch of computer graphics, which produces realistic shadow, reflection, refraction, scattering and other effects by simulating the physical propagation path of light. The original ray tracing method mainly focuses on the rendering of static scenes. With the improvement of computer hardware performance and the continuous optimization of algorithms, ray tracing technology has been significantly developed and gradually extended to the field of dynamic scenes and real-time rendering. With the rise of deep learning, the combination of ray tracing technology and machine learning has become a research hotspot. Ray tracing technology is widely used, covering film production, game development, virtual reality, augmented reality, medical image processing and many other fields.

In short, ray tracing technology, with its powerful simulation ability and realistic visual effects, has become an indispensable part of the field of computer graphics, and constantly promote the development and innovation of graphics rendering technology.

Key words: ray tracing technology

目录

.

1. 引言

1.1 研究背景

光线追踪技术作为计算机图形学领域的重要分支，旨在模拟光在现实世界中传播和反射的物理过程，从而生成高质量的图像和视觉效果。其基本原理是通过追踪从相机或观察者发出的光线，逐步计算光线与场景中物体表面的交点以及它们的相互作用，确定光线的最终颜色和亮度。

光线追踪技术发展始于 20 世纪 80 年代，最早由 Turner Whitted 提出并实现，其基础算法为经典的 Whitted 光线追踪算法。最初的光线追踪方法主要集中在静态场景的渲染上，随着计算机硬件性能的提升和算法的不断优化，光线追踪技术得到了显著的发展，逐渐扩展到动态场景和实时渲染领域。其中的关键技术包括光线与几何体的求交算法优化、加速结构的设计和应用、全局光照算法的实现以及利用 GPU[2]并行计算加速渲染过程等。随着深度学习的兴起，光线追踪技术与机器学习的结合也成为当前的研究热点。例如，通过神经网络优化传统光线追踪算法中的采样过程，提高图像质量和计算效率。

光线追踪技术的应用广泛，涵盖了电影制作、游戏开发[3]、虚拟现实、增强现实、医学图像处理等多个领域。在电影制作中，光线追踪技术被用于创建逼真的虚拟场景和角色；在游戏领域，用于预渲染视频，提供更加沉浸的游戏体验；在医学图像处理中，光线追踪技术可以模拟 X 射线、CT 扫描等影像的物理特性，帮助医生诊断。

总之，光线追踪技术以其强大的模拟能力和逼真的视觉效果，成为计算机图形学领域中不可或缺的一部分，不断推动着图形渲染技术的发展和创新。

1.2 研究目的

计算机图形学中光线追踪技术的研究目的主要包括提高图像渲染的真实感和逼真度，同时尽可能提升计算效率和优化算法。以下是研究目的：

1、继续提升视觉质量：光线追踪技术致力于生成能够模拟真实世界光学现象的图像，如阴影、反射、折射、散射等。通过计算光线与场景中物体的相互作用，产生更加逼真和细致的图像，这对于电影、游戏和虚拟现实等应用来说很重要。
2、处理更复杂的光学效果：光线追踪技术能够有效处理复杂的光学效果，如全局光照、间接光照、次表面散射等。这些光学效果在传统的实时渲染技术中难以精确实现，而光线追踪通过物理模型和数学算法，可以更准确地模拟这些复杂的光学现象，从而提升图像的真实度和表现力。
3、优化渲染效率：尽管光线追踪技术比传统的实时渲染算法更耗费计算资源，但研究更多的优化措施可以在未来继续缩短生成图像的时间，扩展光线追踪技术在实时渲染和交互应用中的可能性。
4、拓展应用领域：随着技术的进步和算法的优化，光线追踪技术将被应用于更广泛的领域。在这些领域中，高质量的图像将提供更好的视觉表达，同时对于实时交互和决策将有重要意义。
5、推动计算机图形学的发展：通过探索新的算法、深度学习方法和优化硬件等，不断拓展光线追踪技术的应用边界，以应对日益复杂和多样化的图像生成需求。

2. 相关工作介绍

以下是光线追踪技术的相关工作介绍：

1、经典光线追踪算法：由 Turner Whitted 在 1980 年提出。该算法通过追踪光线的路径，计算直射光、反射光、折射光和阴影，实现了基本的全局光照效果。虽然效果逼真，但在复杂场景中计算开销较大。

2、路径追踪算法：该算法目的是解决 Whitted 算法中对递归光线的处理限制。路径追踪算法通过随机追踪光线路径，多次采样每个像素，计算所有路径的光照贡献，从而实现更高质量的图像生成。

3、加速数据结构：BVH 等高效的加速数据结构，通过层次化的空间分割和快速相交测试，有效管理光线与场景对象的相交检测，显著提升了追踪效率。

4、实时光线追踪：随着图形处理单元[4]的性能提升，NVIDIA[5]的 RTX 技术和 Microsoft 的 DirectX Raytracing API 推动了实时光线追踪的发展，使其在游戏和虚拟现实等应用中得到广泛使用。

5、混合渲染技术：光线追踪与传统的实时渲染技术的结合被称为混合渲染。这种方法在保留实时渲染的交互性和效率的同时，通过光线追踪实现更高质量的全局光照和影像效果。

综上所述，光线追踪技术通过不断的算法创新、硬件优化，不断扩展应用范围和提升渲染质量，成为计算机图形学领域的重要研究方向之一。

3. 方法描述

下面是光线追踪的流程描述：

1、发射光线：从每个像素的视点位置发射一条光线。视点通常是相机位置，每条光线通过视点位置和相应像素的屏幕位置确定。
2、光线与场景的相交检测：对于发射的每条光线，检测它是否与场景中的物体相交。这一步是光线追踪中计算量最大的部分，需要有效的加速数据结构来快速确定与光线相交的物体。
3、计算光照效果：如果光线与场景中的物体相交，根据相交点的位置、法线和材质属性计算反射、折射和散射等光线的行为。并且考虑光照模型和光源的贡献来计算出最终的颜色值。
4、递归追踪：如果物体是镜面反射或透明的，则继续发射反射或折射光线，并递归地计算它们的贡献，直到达到追踪的最大深度或光线不再与物体相交为止。
5、阴影计算：为了确定某个点是否受到光照，光线追踪还需要检测光线与其他物体之间是否存在遮挡关系。

4. 实验设置

以下是实验设置：

1、项目环境

- 开发工具：使用 Xcode 作为集成开发环境，安装必要的插件和扩展。
- OpenGL 库和依赖项：使用 GLFW、GLAD 库来配置 OpenGL 的窗口和功能。

2、项目配置

- 创建项目：在 Xcode 中创建一个新的 C++项目。
- 配置 tasks.json： 配置 tasks.json 文件编译和构建 C++项目。

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build",
            "type": "shell",
            "command": "g++",
            "args": [
                "-std=c++11",
                "main.cpp",
                "-o",
                "raytracing",
                "-lglfw",
                "-ldl",
                "-lGL",
                "-lX11",
                "-lpthread",
                "-lXrandr",
                "-lXi"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "problemMatcher": ["$gcc"],
            "detail": "Generated task by VS Code."
        }
    ]
}
```

- 配置 launch.json：配置 launch.json 文件启动调试会话。

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "raytracing",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/raytracing",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ],
            "preLaunchTask": "build",
            "miDebuggerPath": "/usr/bin/gdb",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ],
            "logging": {
                "engineLogging": true
            }
        }
    ]
}
```

3、实验参数设置

- 编写和加载着色器：编写 GLSL 着色器文件。

opengl 〉 opengl 〉 fragment_shader 〉 intersectSphere()

```
36
37 void main()
38 {
39     vec3 dir = normalize(rayDir);
40     float closest_t = 10000.0;
41     vec3 hitColor = vec3(0.0);
42
43     for (int i = 0; i < numSpheres; ++i)
44     {
45         float t;
46         if (intersectSphere(cameraPos, dir, spheres[i].center, spheres[i].radius, t))
47         {
48             if (t < closest_t)
49             {
50                 closest_t = t;
51                 vec3 hitPoint = cameraPos + t * dir;
52                 vec3 normal = normalize(hitPoint - spheres[i].center);
53                 vec3 lightDir = normalize(lightPos - hitPoint);
54                 float diffuse = max(dot(normal, lightDir), 0.0);
55                 hitColor = spheres[i].color * diffuse;
56             }
57         }
58     }
59
60     FragColor = vec4(hitColor, 1.0);
61 }
```

- 场景设置：在 C++代码中定义场景参数，并传递给着色器进行渲染。

opengl 〉 opengl 〉 C⁺ main 〉 main()

```
141 unsigned int createShaderProgram(const char* vertexPath, const char* fragmentPath)
142 {
143     std::string vertexCode = loadShaderSource(vertexPath);
144     std::string fragmentCode = loadShaderSource(fragmentPath);
145
146     const char* vShaderCode = vertexCode.c_str();
147     const char* fShaderCode = fragmentCode.c_str();
148
149     unsigned int vertex, fragment;
150     int success;
151     char infoLog[512];
152
153     vertex = glCreateShader(GL_VERTEX_SHADER);
154     glShaderSource(vertex, 1, &vShaderCode, NULL);
155     glCompileShader(vertex);
156     glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
157     if (!success)
158     {
159         glGetShaderInfoLog(vertex, 512, NULL, infoLog);
160         std::cerr << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog <<
              std::endl;
161     }
162
163     fragment = glCreateShader(GL_FRAGMENT_SHADER);
164     glShaderSource(fragment, 1, &fShaderCode, NULL);
165     glCompileShader(fragment);
166     glGetShaderiv(fragment, GL_COMPILE_STATUS, &success);
167     if (!success)
168     {
169         glGetShaderInfoLog(fragment, 512, NULL, infoLog);
170         std::cerr << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog <<
              std::endl;
```

加载、编译和链接顶点着色器和片段着色器，创建一个完整的着色器程序。

```
    // 顶点数据
    float vertices[] = {
        -0.5f, -0.5f, 0.0f,
         0.5f, -0.5f, 0.0f,
        -0.5f,  0.5f, 0.0f,
         0.5f,  0.5f, 0.0f,
    };

    unsigned int VBO, VAO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);

    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
```

设置顶点数据

```
    // 设置 uniform 变量
    glUseProgram(shaderProgram);
    glUniform3f(glGetUniformLocation(shaderProgram, "cameraPos"), 0.0f, 0.0f, 3.0f);
    glUniform3f(glGetUniformLocation(shaderProgram, "lightPos"), -2.0f, 2.0f, 2.0f);

    struct Sphere {
        float center[3];
        float radius;
        float color[3];
    } spheres[1] = {
        {{ 0.0f,  0.0f,  0.0f}, 0.5f, {1.0f, 0.0f, 0.0f}}
    };
    glUniform3fv(glGetUniformLocation(shaderProgram, "spheres[0].center"), 1,
        spheres[0].center);
    glUniform1f(glGetUniformLocation(shaderProgram, "spheres[0].radius"),
        spheres[0].radius);
    glUniform3fv(glGetUniformLocation(shaderProgram, "spheres[0].color"), 1,
        spheres[0].color);
```

设置相机位置、光源位置和场景中的球体属性

```
    while (!glfwWindowShouldClose(window))
    {
        processInput(window);

        glClear(GL_COLOR_BUFFER_BIT);

        glUseProgram(shaderProgram);
        glBindVertexArray(VAO);
        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteProgram(shaderProgram);

    glfwTerminate();
    return 0;
}
```

处理用户输入并在每个帧中绘制四边形，模拟光线追踪。

5. 实验结果与分析

光线追踪技术的效果受多个因素影响，这些因素直接影响最终渲染的效果。

1、场景复杂度：场景中的对象数量、复杂度和几何形状会显著影响性能。更复杂的场景需要更多的光线和相交测试，导致渲染时间增加。

2、光照模型：光照模型决定了如何模拟光的传播和交互。精确的光照模型能够产生更真实的光影效果，例如镜面反射、折射和阴影。

3、材质属性：物体的表面属性（如反射率、折射率和表面粗糙度）会对反射光线的行为产生直接影响。具有高反射率或折射率的材质会导致更复杂的光线路径和更明显的反射效果。

4、光源：光源的类型、位置和强度对最终渲染结果具有重要影响。设置正确的光源能够产生更逼真的光照效果。

5、渲染算法的优化程度：光线追踪的实时性能取决于算法的优化程度，包括加速结构、光线与物体相交的快速判断算法以及并行处理的使用。

在高质量渲染设置下，路径追踪的渲染时间较长。通过实验测试，在增加光线样本数和最大追踪深度时，渲染时间呈指数增长。表展示了不同参数设置下的渲染时间和图像质量，可以看出增加光线样本数和追踪深度能够显著提高图像质量，但同时也带来了更高的计算开销。

样本数	最大追踪深度	渲染时间	图像质量（PSNR）
16	5	15s	25.8
64	5	60s	30.2
256	5	240s	35.7
64	10	120s	32.5
256	10	480s	38.1

6. 结论

光线追踪技术作为计算机图形学中的重要分支，通过模拟光线在场景中的传播路径，能够精确模拟光影、反射、折射等光学效果，因此在现实感图像生成和视觉效果增强方面具有广泛应用前景。本次实验基于 Xcode 和 OpenGL 环境，使用光线追踪算法实现了简单的场景渲染，包括球体的阴影和漫反射效果。实验结果表明，通过调整球体的位置、大小和材质属性，可以直观地观察到光线追踪技术在模拟真实光影过程中的表现。

然而，光线追踪技术仍面临着渲染速度慢、计算复杂度高等挑战，特别是在处理复杂场景和动态光照时表现不足。此外，该项技术对于硬件的要求较高也限制了其在实时应用中的推广。

综上所述，光线追踪技术在图形学和计算机视觉领域的发展前景广阔，但仍需要进一步的技术改进和应用研究支持其在更多实际场景中的应用。

7.参考文献

[1] 徐汝彪，刘慎权．光线追踪技术综述[J]. 计算机研究与发展，1990, 27(5):9.DOI:CNKI:SUN:JFYZ.0.1990-05-003.

[2] 王红斌．基于 GPU 的高效光线追踪技术研究与实现[D]. 长春理工大学 [2024-06-23].DOI:10.7666/d.y1662411.

[3]张健浪.游戏的未来:光线追踪技术步入实用化[J].新电脑, 2007(007):031.

[4] 曹小鹏．图形处理器关键技术和光线追踪并行结构研究[D]. 西安电子科技大学 [2024-06-23].DOI:CNKI:CDMD:1.1016.214232.

[5]齐健.NVIDIA在中国正式发布Quadro V100,实时光线追踪和AI技术成为热点[J].智能制造, 2018(5):2.DOI:CNKI:SUN:JSFY.0.2018-05-028.