



西安交通大学计算机图形学实验文档

渲染部分

作者：李昌洲 袁右文 李昊东

组织：计算机图形学课题组

时间：October 18, 2024



目录

第 1 章 软光栅化渲染器	1
1.1 实验内容	1
1.2 指导和要求	1
1.2.1 光栅化渲染管线	1
1.2.2 剔除操作	3
1.2.3 Vertex Shader 的法线变换	3
1.2.4 Rasterization 的透视矫正插值	4
1.2.5 代码文件组织	6
1.2.6 要求	6
1.3 提交和验收	7
第 2 章 软光栅渲染器的并行优化	9
2.1 实验内容	9
2.2 指导和要求	9
2.2.1 现代 GPU 架构与 SIMT 模型	9
2.2.2 渲染管线在 GPU 中的运作方式	10
2.2.3 C++ 中影响多线程速度的因素	12
2.2.4 并行化需要解决的问题	12
2.2.5 要求	13
2.3 提交和验收	13
第 3 章 基于 Shadow Mapping 的阴影实现	15
3.1 实验内容	15
3.2 指导和要求	15
3.2.1 Shadow Map 的生成方法	15
3.2.2 查找 Shadow Map	15
3.2.3 要求	15
3.3 提交和验收	16
第 4 章 基于 PCSS 的软阴影效果	17
4.1 实验内容	17
4.2 指导和要求	17
4.2.1 使用 PCF 生成柔性阴影边缘	17
4.2.2 面光源与软阴影	17
4.2.3 要求	17
4.3 提交和验收	18
第 5 章 Whitted-Style Ray-Tracing	19
5.1 实验内容	19
5.2 指导和要求	19
5.2.1 主射线生成	19
5.2.2 判断是否与物体相交的注意点	20

5.2.3 阴影的判定	22
5.2.4 要求	22
5.3 提交和验收	22
第 6 章 用 BVH 加速光线求交	24
6.1 实验内容	24
6.2 指导和要求	24
6.2.1 建立 BVH 的全流程	24
6.2.2 BVH 相交的注意点	27
6.2.3 要求	28
6.3 提交和验收	28
参考文献	30

第 1 章 软光栅化渲染器

软光栅化渲染器 (编号 2.2) 这个实验中，我们将自己实现一个简略的软光栅化渲染器。

什么是软光栅化渲染器？首先我们需要理解什么是软渲染器。软渲染器就是由通用计算硬件（比如 CPU，或者 GPGPU，而不是专用的图形硬件）完成整个渲染流程，生成图像。我们编写的软渲染器是使用 CPU 的，接受的输入为顶点数据（包含位置，法向量等各种信息），输出是一帧的颜色缓冲（也可以理解成一张图片）。将帧缓冲写入文件或者用咱们的 UI 界面显示出来，就完成了整个渲染过程。

那么什么是光栅化呢？光栅化是将矢量的、解析形式的几何图元转变为二维像素阵列的过程。¹该过程包含了两部分的工作：第一部分是决定窗口坐标中哪些整型栅格区域被基本图元占用；第二部分是分配一个颜色值和深度值到各个区域。光栅化过程产生的是片元。

了解了光栅化和软渲染器的概念，我们就可以将软光栅化渲染器理解为使用光栅化的方式进行渲染的软渲染器，之后我们还会实现例如基于经典光线追踪的软渲染器，它们各有各的优点。那么这一章，当我们实现了软光栅化渲染器后，就可以动手添加自己喜欢的物体进行渲染了！

1.1 实验内容

想要实现一个软光栅化渲染器，我们首先应当明确自己的目标，也就是最终的渲染器应当是什么样子？应当具备哪些功能？在本章实验中，我们主要需要为软光栅化渲染器提供以下功能：

- 对输入的顶点信息（包括位置和法向量）进行几何变换的功能
- 三角形图元的光栅化的功能
- 使用 Phong 模型对片元着色的功能

1.2 指导和要求

1.2.1 光栅化渲染管线

我们即将实现的光栅化管线有如图 1.1 所示的几个步骤：

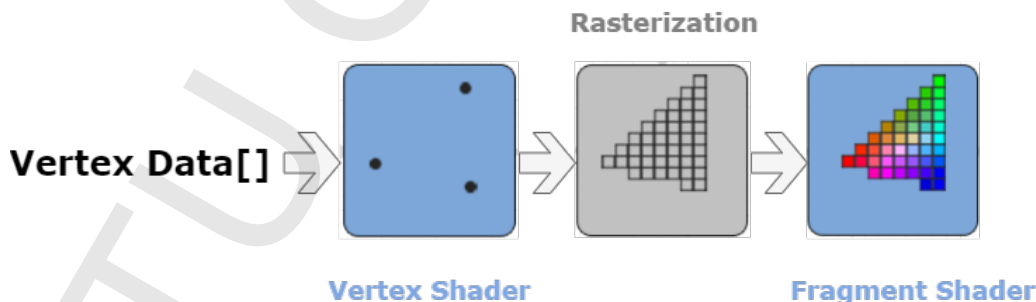


图 1.1: 光栅化流程，图片来自 [4]

- **Vertex Shader:** 作用于每个顶点，通常是处理从世界坐标到裁剪坐标（屏幕坐标）的坐标变换，我们的光栅化器主要处理顶点的位置和法线。其中输入为模型坐标系下的顶点位置和法线，输出为变换到视口空间的顶点位置（方便剪裁）和世界/相机坐标系下的法线（方便插值）。

¹如果你对信号理论有所了解，那么更简洁的说法是对连续信号采样的过程。

这里需要注意的是，我们不仅需要变换到视口空间的顶点位置，在光栅化时，我们还需要世界/相机坐标系中的顶点位置用于插值，因此，Vertex Shader 返回哪一个坐标系的顶点位置均可，只需在函数外部再计算另一个坐标系的顶点位置即可，也可以选择都在 Vertex Shader 中进行计算一并返回，具体选择哪种方式由同学们自己决定。

- **Rasterization** 主要实现了简化版的视口剔除，深度测试和透视矫正以及在相机坐标系下的坐标插值和法线插值。在 Vertex Shader 和 Fragment Shader 之间起到数据处理的桥梁作用。

这里需要注意的是，由于在透视投影时发生的非线性变换很可能将视锥体外一部分顶点坐标变化得很大，这时对于像 cube 这类面数很少的物体在插值时就会出现严重的精度损失，例如出现 z-buffer 在两个物体交界处判定错误，更严重的情况是发生整型溢出，那么会出现一部分区域出现完全错误的渲染结果，这种情况的解决方式是使用 double 型提升精度，但由于在绝大部分情况下，并不会对渲染结果出现较大的影响，因此同学们只需了解为什么出现这种情况即可，无需手动修改为双精度浮点型。

- **Fragment Shader** 作用于每个屏幕上的片元（可以近似理解为像素），计算出片元的颜色后，存储在光栅化器的 frame buffer 中。

这里需要注意的是，着色有很多种不同的方式，而不同着色方式对加载的数据要求不同，本章要求同学们使用 Phong Shading 进行着色。在着色时，同学们会用到 `GL::Material`。这个类中有四个成员，`ambient/diffuse/specular` 分别对应 Phong Shading 所用的 $k_a/k_d/k_s$ ；而 `shininess` 则对应 Phong Shading 中计算镜面反射时的指数 p ，即体现了高光的集中程度。举个例子，如果要计算镜面反射部分的渲染结果，则应当使用：

```
Vector3f specular = material.specular.cwiseProduct(attenuated_light) *
    std::pow(std::max(0.0f, normal.dot(half_vec)),
    material.shininess);
```

在 GPU 这样的专用图形硬件上，光栅化渲染管线是流水化执行的：就像 CPU 执行指令时会分为若干个阶段，让不同指令的不同阶段同时执行（即时间重叠）来提高硬件利用率；GPU 上执行光栅化渲染时，不同的运算单元会分别执行 vertex shader / rasterization / fragment shader。

需要注意的是，流水化并不完全等于并行化：如果三角形 A 先于 B 被送入流水线，那么 A 也一定先于 B 完成渲染。从逻辑顺序上来说，流水化的管线仍然是串行的。但当 A 进入片元处理阶段时，B 可以执行顶点处理或光栅化，这意味着我们可以使用更多的线程、同时处理更多的数据。本实验中你需要实现的管线正是这样一个流水化管线，需要用到四个线程：

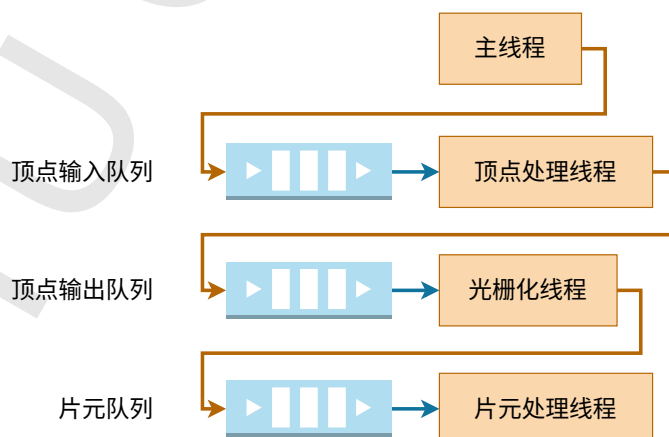


图 1.2: 光栅化渲染管线的流水结构

- 主线程（UI 线程）：构造顶点数据单元 `VertexShaderPayload`

- 顶点处理线程：对顶点进行坐标变换
- 光栅化线程：将顶点数据单元重新组装成三角形，然后将其光栅化为片元数据单元 `FragmentShaderPayload`
- 片元处理线程：对片元着色得到像素

这四个线程之间依赖三个队列交换数据，如图 1.2 所示：

- 顶点输入队列：存放主线程构造好的顶点数据单元
- 顶点输出队列：存放处理过的顶点数据
- 片元队列：存放光栅化线程输出的片元

按照这样的思路实现流水化管线，需要将任务分散到四个线程中。而线程之间是典型的生产者-消费者关系，因此你需要了解基础的多线程编程操作（使用 `std::thread`）、线程间同步操作（基于锁的同步）。

1.2.2 剔除操作

在我们即将实现的光栅化器中，Clipping 操作与传统意义的剪裁与剔除有所不同。

传统的剔除，共有三次，按照先后顺序分别是：**视锥剔除**、**视口剔除**、**背面剔除**。

- **视锥剔除**：通过 AABB 盒、OBB 盒等将物体包围起来，然后与视锥体做碰撞检测，可以直接剔除掉完全不可见的物体，运算量较小但精度也较低。
- **视口剔除**：在坐标变换后期运行，也就是在裁剪空间做透视除法之前运行（因为如果有物体在摄像机的位置，会出现 $w=0$ ，做透视除法的时候出现除零错误），这一阶段是将在视口外的物体丢弃，一部分在视口内的物体，会进行裁剪，生成新的多边形。
- **背面剔除**：在顶点着色器和片段着色器之间运行，通过计算的法线方向，剔除掉背向面

而我们的光栅化器只需要进行一次剔除操作且发生在屏幕空间内。即当某个片元变换到屏幕空间后，超出了可以显示的范围，则直接丢弃该片元即可。这样做是为了减轻此处同学们的工作量，虽然损失了一部分性能，但在尽可能保证完整效果的前提下，能够让同学们更加专注于实现和体会整个软光栅渲染器的流程。

1.2.3 Vertex Shader 的法线变换

在 Vertex Shader 中，我们需要对顶点的坐标通过左乘 `Uniforms::MVP` ($projection \cdot view \cdot model$) 变换到齐次剪裁坐标系下，但顶点法线则不能使用模型变换矩阵进行变换。如果对图形施加一个非等比缩放，模型的法线可能与模型表面不垂直。如图 1.3 所示。

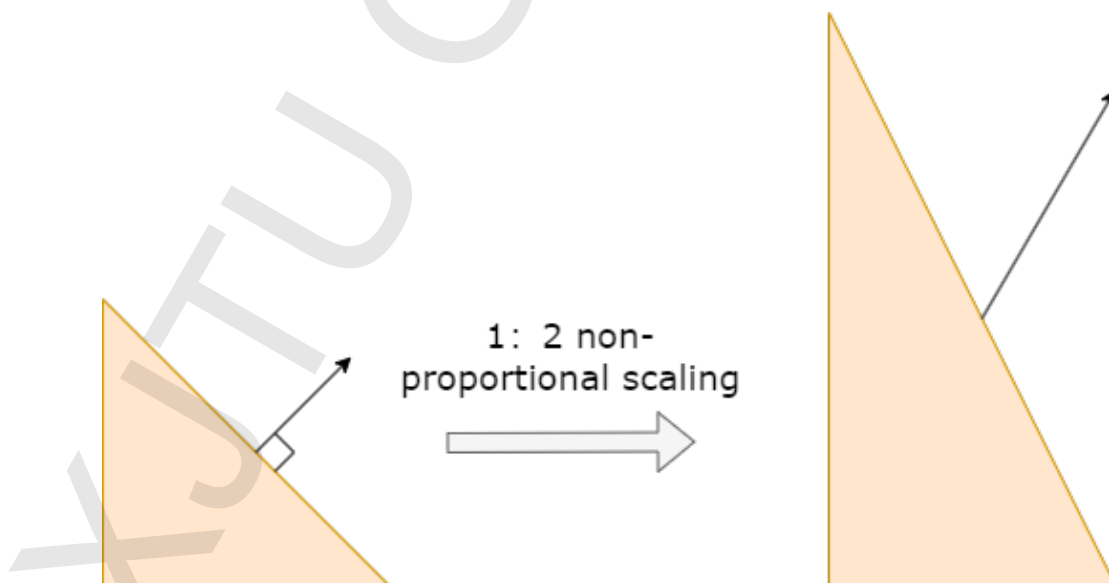


图 1.3: 不等比缩放的法线变化

如果模型变换中只包含平移、旋转和等比缩放，那么经过模型变换后法线依然与表面垂直。因为旋转变换是正交变换，不改变共面向量间的夹角；法线是向量，可以任意平移；等比缩放之后，法线只有模长改变。因此我们需要考虑模型变换中包含非等比缩放时，法线应该使用什么样的变换矩阵。

为了计算法线的正确变换矩阵，一个比较简单的方法是借助切线与法线始终相互垂直的关系。切线有不受模型变换影响的良好性质，即使是非等比缩放后，仍能保证与平面相切。我们利用这一性质来推导法线的模型变换矩阵。设法线方向为 $\vec{n}$ ，切线方向为 $\vec{t}$ ，模型变换矩阵为 M ，法线模型变换矩阵为 G ，那么有：

$$(G\vec{n})^T \cdot (M\vec{t}) = 0$$

因此有：

$$\vec{n}^T G^T M \vec{t} = 0$$

因为 $\vec{n}^T \vec{t} = 0$ ，那么只要 $G^T M = I$ ，就能满足 $(G\vec{n})^T \cdot (M\vec{t}) = 0$ ，因此得到：

$$G = (M^{-1})^T$$

1.2.4 Rasterization 的透视矫正插值

为什么需要透视矫正插值？我们先来梳理两个知识点，第一个是渲染管线中从物体坐标空间变换到屏幕坐标空间的变换流程，如图1.4

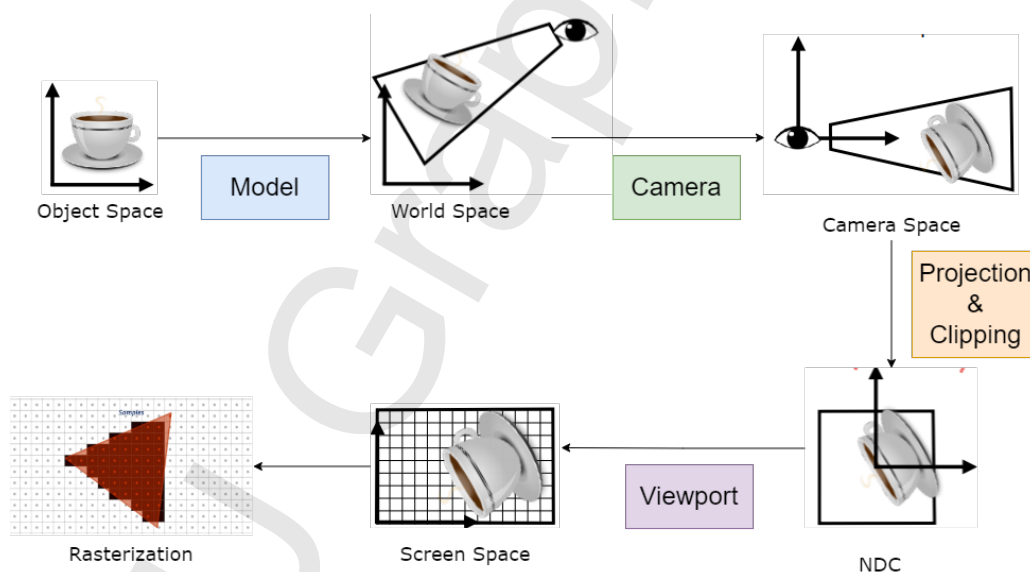


图 1.4: 物体坐标空间到屏幕空间的变换流程

看到这个流程图，不知道对同学们有没有一点启发？其实之所以要进行透视矫正插值，是因为普通的插值操作是线性插值，而我们在坐标空间变换过程中产生了非线性变换，因而线性插值是无法得到正确的插值结果的。那么在这个变换过程中，具体是哪一步发生了非线性变换呢？如果同学们还是没有思路，那么我们接着梳理第二个知识点，也就是透视投影矩阵 P ：

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

其中, t, b, l, r 分别表示视锥体近平面的上下左右边长, f, n 分别表示视锥体的远近平面距离。

这个矩阵的特别之处来自第四行的非零项 -1 , 这个 -1 与 w 相乘后, 做齐次除法之后的结果相当于在原来的坐标上都除以了 $-z$, 这带给我们了两个关键信息:

1. 坐标系的变换从原来的线性关系变成了非线性关系。自然在线性关系下的坐标系数也不适用于经过非线性变换后的三角形了。
2. 透视投影后的 w 表示的是**透视投影前的 z 值**。

有了以上的两个关键信息后, 我们就可以对透视矫正插值进行推导了。如果进行直接线性重心插值, 则形式是:

$$I = \alpha I_0 + \beta I_1 + \gamma I_2$$

其中三个重心坐标系数是基于投影变换后的三角形顶点得出。顶点的属性 I_0, I_1, I_2 是变换前三角形的顶点表示的属性, 但显然这样线性插值得到的插值结果是不对的。那么根据第一个关键信息, 我们应该明确现在非线性变化之后的插值和普通的线性插值有什么联系。

根据**Perspective-Correct Interpolation**这篇文章中的详细推导, 我们可以把投影变换后的空间看作一个与 $\frac{1}{z}$ 相关的线性空间。即我们只需要用基于屏幕空间计算出的重心坐标就能线性插值出一个基于 $\frac{1}{z}$ 空间的属性 $I_{\frac{1}{z}}$, 这个结果再乘以 $\frac{1}{z}$ 的插值结果的倒数, 即可得到透视矫正之后的正确答案。其一维数学表达式如下:

$$I_t = \left(\frac{I_1}{Z_1} + s \left(\frac{I_2}{Z_2} - \frac{I_1}{Z_1} \right) \right) * Z_t$$

$$Z_t = \frac{1}{\frac{1}{Z_1} + s \left(\frac{1}{Z_2} - \frac{1}{Z_1} \right)}$$

文章 [2] 中是直接对一维线段插值进行推导的。 Z_1, Z_2 代表的是线段两个端点在透视投影前的深度值, I_1, I_2 代表的是线段两个端点待插值的属性, Z_t 则是在屏幕坐标系中插值出的点在投影前的深度值。其实 Z_t 表达式还告诉我们一个结论, 即: **如果仅仅是为了做深度测试, 并不需要进行透视矫正, 因为深度的前后关系并不会随着透视投影变换而发生改变**。但由于我们要求同学们掌握矫正插值的思想, 所以要求同学们需要在计算 z -buffer 时, 进行透视矫正。透视矫正正在三角形中的情况可以直接将原表达式拓展, 得到三角形中的正确插值表达式:

$$I_t = \left(\alpha \frac{I_1}{Z_1} + \beta \frac{I_2}{Z_2} + \gamma \frac{I_3}{Z_3} \right) * Z_t$$

$$Z_t = \frac{1}{\alpha \frac{1}{Z_1} + \beta \frac{1}{Z_2} + \gamma \frac{1}{Z_3}}$$

其中, I_1, I_2, I_3 分别为三角形三个顶点的属性, α, β, γ 是当前插值点在屏幕坐标系下计算出的重心坐标, Z_1, Z_2, Z_3 为三个顶点在透视投影之前三维空间中的 z 值, 再由我们一开始得到的结论, 即透视投影后的 w 表示的是透视投影前的 z 值, 我们可以得到最终的表达式为:

$$I_t = (\alpha \frac{I_1}{w_1} + \beta \frac{I_2}{w_2} + \gamma \frac{I_3}{w_3}) * Z_t$$

$$Z_t = \frac{1}{\alpha \frac{1}{w_1} + \beta \frac{1}{w_2} + \gamma \frac{1}{w_3}}$$

w 是经过透视投影之后，还未使用透视除法的齐次坐标的第四维。

1.2.5 代码文件组织

渲染部分的代码全部都放在 `src/render` 目录下，软光栅化渲染器主要涉及以下几个代码文件，其文件结构如下：

```
src/
├── render/
│   ├── graphics_interface.h
│   ├── render_engine.h
│   ├── render_engine.cpp
│   ├── rasterizer_render.h
│   ├── rasterizer_render.cpp
│   ├── rasterizer.h
│   ├── rasterizer.cpp
│   ├── shader.cpp
│   ├── triangle.h
│   └── triangle.cpp
```

- `graphics_interface.h` 定义渲染管线全局公用的类型信息，包括顶点/片元数据单元、Uniforms、上下文等。
- `render_engine.h`: 渲染引擎与渲染器的头文件。Dandelion 的离线渲染工具由一个渲染引擎（最外层接口）与若干渲染器（实现渲染管线）构成。
- `render_engine.cpp`: 渲染引擎的源文件，实现调用具体渲染器的行为。
- `rasterizer_render.h/cpp` 光栅化渲染器的其他相关定义与管线实现。
- `rasterizer.h/cpp`: 光栅化阶段的定义和实现。
- `shader.cpp`: 着色器的源文件。该文件包含 vertex shader 和 fragment shader 的实现。
- `triangle.h/cpp`: 光栅化阶段组装 (primitive assembly) 成的三角形

1.2.6 要求

按照整个软光栅化渲染器的渲染流程，请对以下提到的文件中的函数空缺处进行填写，使软光栅化渲染器能得到正确的结果：

- `rasterizer_renderer.cpp`
 - `RasterizerRenderer::render`: 遍历场景，设置 Uniforms 变量并将顶点数据打包成 `VertexShaderPayload` 送入队列。
 - `VertexProcessor::worker_thread`: 从顶点输入队列中取出顶点数据，调用 vertex shader 进行坐标变换。
 - `FragmentProcessor::worker_thread`: 从片元队列中取出片元数据，调用 fragment shader 完成着色。
- `rasterizer.cpp`:
 - `Rasterizer::worker_thread`: 从顶点输出队列中取出数据，组装图元（三角形）并光栅化之。
 - `Rasterizer::inside_triangle`: 给定像素坐标 (x, y) 以及三角形的三个顶点坐标，判断 (x, y)

是否在三角形的内部。

- `Rasterizer::compute_barycentric_2d`: 给定像素坐标 (x, y) 以及三角形的三个顶点坐标, 计算 (x, y) 对应的重心坐标 (α, β, γ) 。
- `Rasterizer::rasterize_triangle`: 对当前三角形进行光栅化, 在这部分中, 深度值, 顶点坐标和法线均进行了透视矫正插值。
- `shader.cpp`:
 - `vertex_shader`: 将顶点坐标变换到投影平面, 再进行视口变换; 同时将法线向量变换到相机坐标系用于后续插值。
 - `phong_fragment_shader`: 使用 Blinn Phong 模型计算每个片元 (像素) 的颜色。

提交的实验结果包含两张渲染结果的 png 格式图片:

- 加载 `cow.dae`, 添加一个点光源并将其位置调整为 $(2.0, 2.0, 2.0)$, 同时将相机位置也调整为 $(2.0, 2.0, 2.0)$, 背景颜色任意, 其他所有参数保持默认, 你将得到类似图 1.5 的渲染结果:



图 1.5: 实验截图 1

- 加载任意一个文件 (dae 或 obj 格式), 所有参数自行设定, 渲染出你想要的效果, 比如图 1.6

 **笔记** 图 1.6 使用的模型来自网络, 其使用条款允许免费下载但不允许分发, 所以不会随测试数据下发。可用的免费模型网上有很多, 大家可以按自己的喜好挑选, 在渲染自己的模型时, 可以试着调整模型的材质、添加多个光源、调整光强和相机的视角等, 以此来获得更加满意的渲染结果。

1.3 提交和验收

提交的实验结果包含上述要求中提到的两张渲染结果的 png 格式图片, 其余提交要求参照基础部分文档的附录 A 的 A.1 部分

另外, 验收时需要解释自己实现的渲染器; 现场以 Release 模式编译并运行程序, 渲染一个指定的场景。如果能成功将物体渲染到图像中得 5 分, 物体颜色和位置正确再得 10 分。



图 1.6: 实验截图 2

第 2 章 软光栅渲染器的并行优化

本章（编号 2.3）实验中，我们将为第 1 章中的软光栅化渲染器进一步引入并行优化。完成第 1 章的实验后，我们理解了整个软光栅化渲染的流程；而这一章中，我们将通过阶段内并行进一步提高渲染性能。

2.1 实验内容

本章的实验主要是在之前实现的代码基础上进行更改，在阅读了下面对 GPU 架构及其对应渲染管线的简要介绍后，你需要自己思考并行优化策略，实现阶段内并行的软光栅化渲染器。

2.2 指导和要求

2.2.1 现代 GPU 架构与 SIMT 模型

我们在第 1 章中实现的软光栅化渲染器，所有的步骤都是交由 CPU 执行的。而在实际应用中，由于 GPU 的并行能力远高于 CPU，我们常常使用 GPU 进行一些可以大规模并行的任务以提高渲染的速度，例如在 GPU 上执行 Vertex Shader 或 Fragment Shader，本质是利用大量的内核进行大规模并行从而提高并行速度。当然，CPU 也并非一无是处，更强的单线程通用计算能力意味着可以执行更复杂的离线渲染算法，从而得到更高质量的图像。为了更好地改进我们的软光栅渲染器，我们应当先了解在 GPU 端进行的渲染管线为什么快，再将其思想应用到我们的软光栅渲染器中。

那么第一个问题是，渲染使用的 GPU 长什么样？其抽象出的架构如图 2.1 所示：

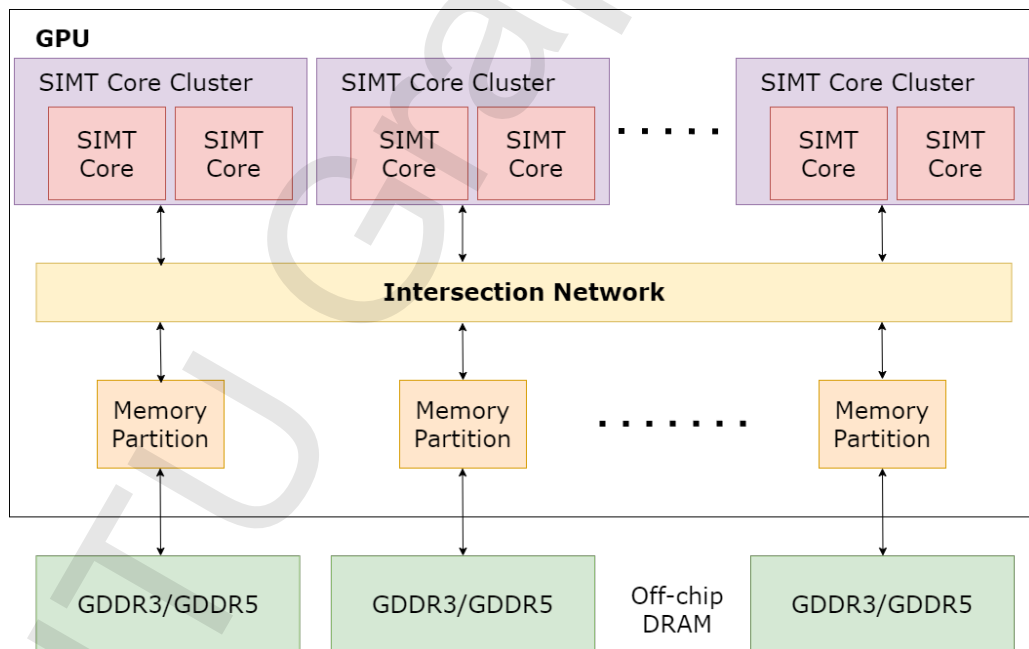


图 2.1: 现代 GPU 架构与 SIMT 模型

每个 GPU 有很多核心，英伟达称之为流式多处理器 (streaming multiprocessors, SM)。每个核都在执行单指令多线程的程序 (single-instruction multiple-thread, SIMT)。

这里出现了第二个名词——SIMT。那么 SIMT 是什么？我们知道，典型的计算模式有两种：**CPU 式的高速低延迟串行计算**以及 **GPU 式的高延迟高吞吐大规模并行计算**。对于 GPU 来说，让多个线程执行完全一致的指

令，可以仅用单路指令流控制多个执行单元，大幅度减少控制器的个数和系统的复杂度（否则成千上万的线程做不同的事，同时还需要线程间通讯/同步，将带来巨大的性能损耗）。渲染过程中的很多计算任务都有“同质”、“相互独立”的特点，适合高吞吐量的并行计算模式。比如上一章的 Vertex Shader 和 Fragment Shader，顶点或像素点非常可观，但是每个顶点或像素点上进行的计算过程是一样的。计算中数据之间的相互影响也具有“局限性”，一个数据单元的计算最多需要某个小邻域内的数据，即任务之间是弱耦合的。

针对“进行的计算是一样的”这一点，最简单的并行计算方案就是同时进行完全一致的计算，即 SIMD(单指令多数据流)。但实际情况下 shader 内可能有条件分支这样的跳转指令，GPU 上不同 SP 实际需要执行的指令未必完全相同。这种情况下，SIMT 允许分支内并行、分支间串行¹，比 SIMD 提供了更强的灵活性，也更适合编写相对复杂的 shader。

图 2.2 是英伟达在 2004 年推出的 NVIDIA GeForce 6800，能很清楚的看到整个光栅化渲染的整个流程，即从 Vertex Processing 到 Rasterization，然后是 Fragment Processing，最后是 Blending。

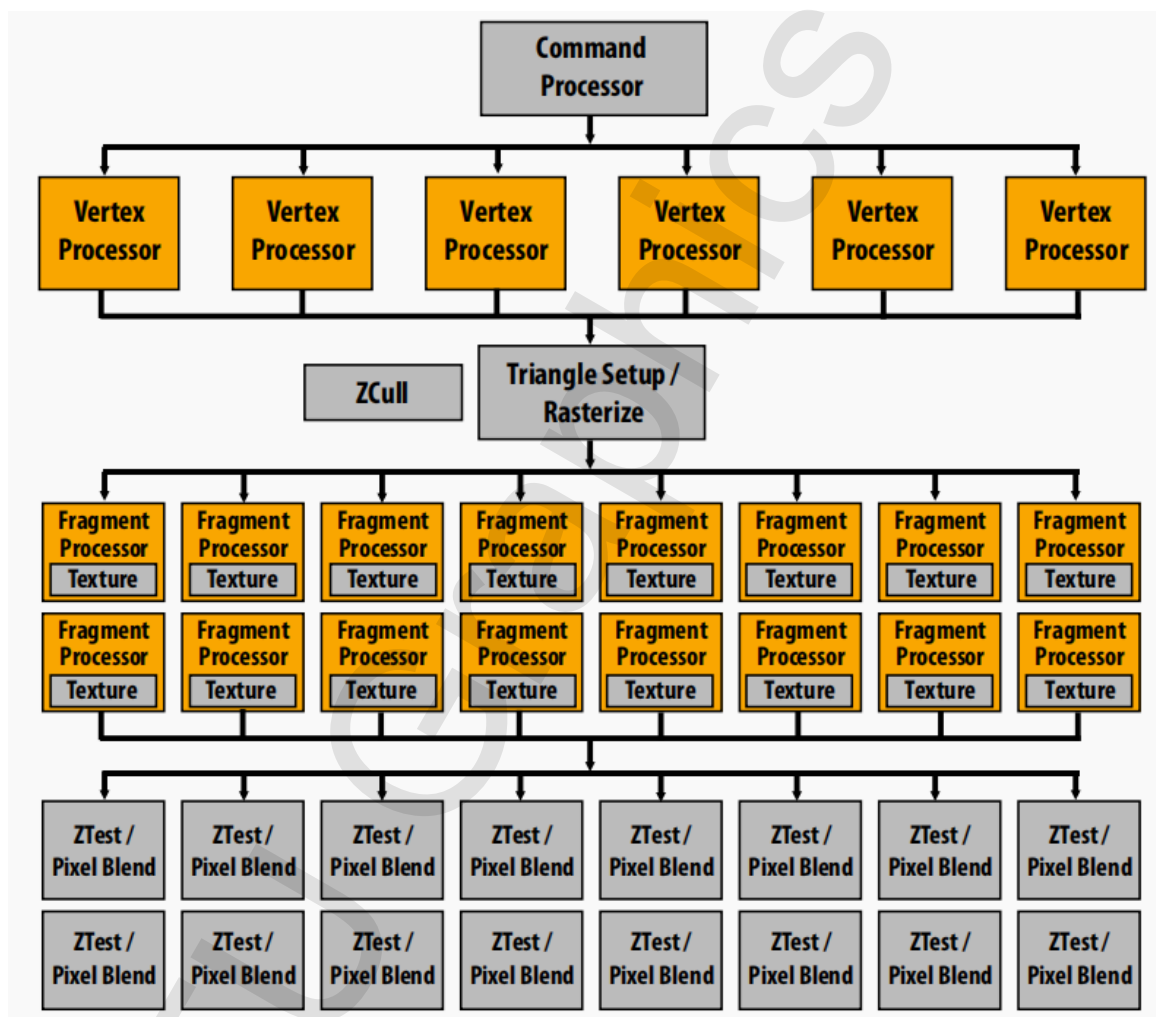


图 2.2: NVIDIA GeForce 6800，图片来自 [1]

2.2.2 渲染管线在 GPU 中的运作方式

在这一节，我们将对 Vertex Shader 和 Fragment Shader 有更深刻的认识。

区别于早期的固定管线，新的渲染管线增强了可编程能力，将一部分之前仅可通过图形 API 配置实现的功能

¹例如 if-else 条件分支，可能会有一半 SP 执行 if 分支，此时另一半执行 else 分支的 SM 会等待；所有的 if 分支都执行完以后，再执行所有的 else 分支。实际情况可能更复杂，涉及更多并行优化的知识。

能或者通过配置也无法实现的功能，以可编程的形式发放给了图形程序员，这就是 Shader。

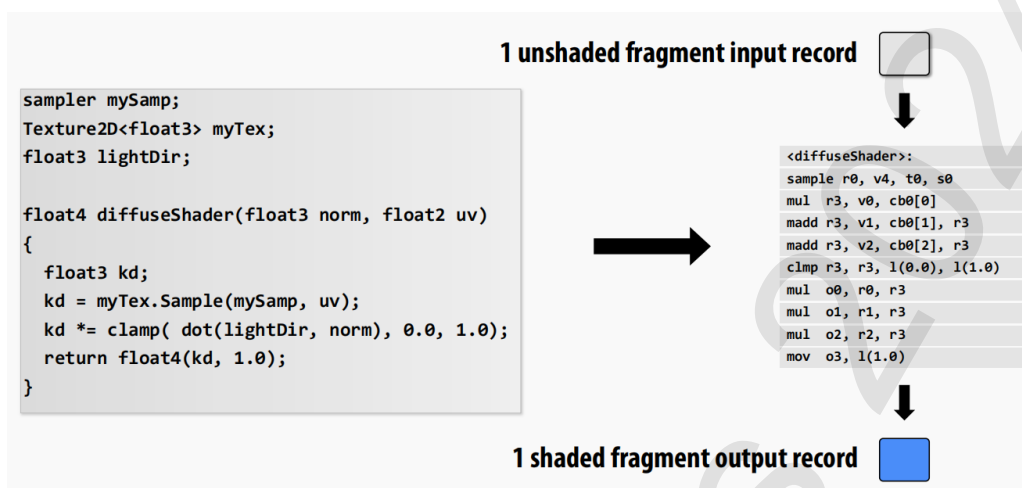


图 2.3: Fragment Shader 及其编译后的代码，图片来自 [1]

图 2.3 左边是 Fragment Shader 的一部分代码，右边是其编译之后的代码。这些指令一条一条地在 GPU 的 SP（流处理器）或 SFU（特殊函数单元）上执行。由于多个像素通常是共享同一批 Shader 指令的，所以这并不是 Shader 的最终编译结果，最终编译结果如图 2.4 所示。

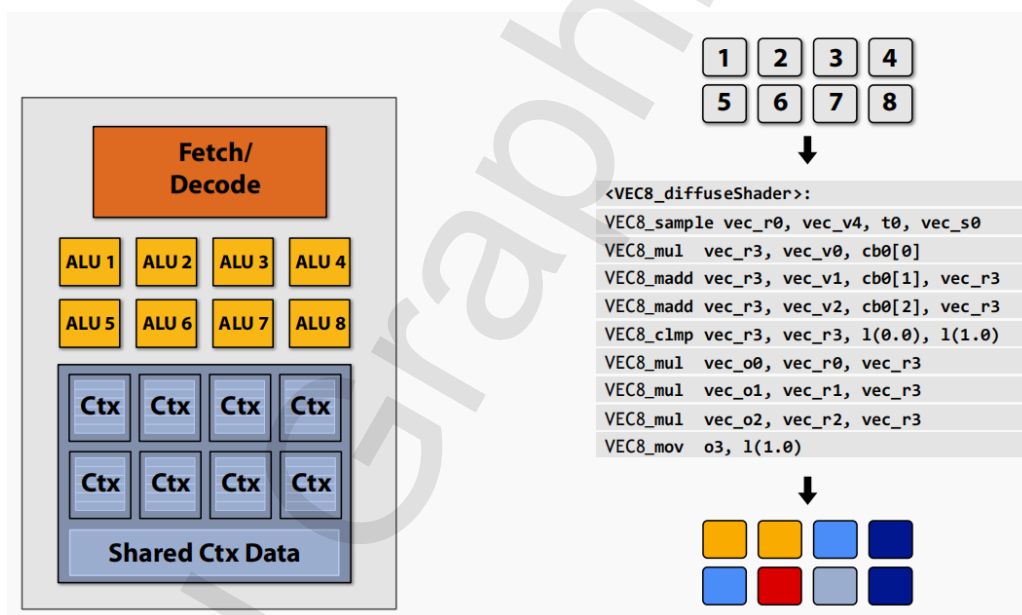


图 2.4: 多像素共享同一指令，图片来自 [1]

图 2.4 左边的结构就是在 2.2.1 中介绍的流式多处理器的部分。假设像素着色同时在 8 个 ALU（SP Core）上同步执行。而 Ctx 则用于线程切换时临时存储当前执行线程的上下文，当某一组像素在执行时被阻塞了（例如贴图、采样等耗时操作），线程调度器会保存当前执行的上下文，并迅速切换到另一组像素及指令进行执行，尽可能隐藏耗时指令带来的延迟。除了 Fragment Shader，Vertex Shader、Geometry Shader 等也会按照上述的模式进行操作，并且不同的功能也是可以并行的。

结合图 2.2 和本节的讨论，我们再对 GPU 渲染管线做一次梳理：

1. 模型数据以及渲染指令通过 CPU 端调用图形 API，将其传输到 GPU 端
2. 之后进入几何处理阶段，由 Vertex Work Distribution 将顶点数据分配到不同的流式处理器中并执行 Vertex Shader
3. 执行之后的离散顶点进行图元装配，裁剪和光栅化等功能产生离散片元

4. 生成的离散片元由 Pixel Work Distribution 分配给不同的流式处理器执行 Fragment Shader
5. 最后再次分配给流式处理器进行深度测试和 Blending

根据我们梳理出的真实渲染管线，再结合在第 1 章中我们实现的软光栅化渲染器，就能明确我们的多线程优化目标：即如何让光栅化渲染管线的每个阶段都并行起来。

2.2.3 C++ 中影响多线程速度的因素

既然明确了我们需要多线程优化的目标，那么接下来需要考虑的就是优化的方法了。我们应该从哪些方面入手去考虑多线程的优化呢？

这里只对每个因素进行简略地提及，如果同学们需要深入了解，可以自行搜索或是参考 **C++ 影响多线程速度的因素记录**

1. **缓存行同步问题/共享数据竞争问题**：当并行时，多个 CPU 可能同时读入同一个缓存行，由于每个 CPU 的数据相互独立，如果需要对缓存行进行更改，就会导致数据再每个 CPU 上不一致，这时就需要同步。比如在我们的软光栅化渲染器中，frame buffer 的写入就面临这样的问题，遇到两个线程访问临界资源时，常见的做法是加锁，实现两个线程的串行化，当然，还有另一种思路是将临界资源复制线程数量份，最后全部执行结束后再进行合并。这两种方式都会带来一定的开销，哪种方式更优需要同学们实验验证。

 **笔记** 缓存行同步并不能保证在并行时得到正确的结果，假设有线程 A 和 B 对数据 var 进行自增操作：

- A 读 var $\Rightarrow$ A 执行 $var + 1 \Rightarrow$ A 写入 var $\Rightarrow$ B 缓存行同步 $\Rightarrow$ B 读 var $\Rightarrow$ B 执行 $var + 1 \Rightarrow$ B 写入 var $\Rightarrow$ A 缓存行同步
- A 读 var $\Rightarrow$ B 读 var $\Rightarrow$ A 执行 $var + 1 \Rightarrow$ B 执行 $var + 1 \Rightarrow$ A 写入 var $\Rightarrow$ B 缓存行同步 $\Rightarrow$ B 写入 var $\Rightarrow$ A 缓存行同步

能看出问题所在吗？注意两种方式中 A 和 B 读入 var 的位置。后者的 B 在进行缓存行同步以后，并没有再执行 $var + 1$ ，而是在最开始读入的 var 上执行的自增操作，这相当于整个过程只进行了一次自增操作。所以想要真正获得正确的答案，必须保证从读入到写入整个过程加锁。

2. **任务颗粒度过小问题**：操作系统新建和销毁线程是需要时间花销的，如果线程执行的任务过于简单，管理它的代价胜于它加速的时间，那么就会得不偿失。同理，加锁操作也面临这样的问题，如果加锁和解锁的代价胜于它加速的时间，那么不如一开始就选择串行。

2.2.4 并行化需要解决的问题

在 1.2.1 节中，图 1.2 解释了当前串行的流水化管线，其中四个线程：主线程、顶点处理线程、光栅化线程、片元处理线程四个线程中，除了主线程外的另外三个线程都可以实现并行化。然而，以下的一些问题，都是在同学们进行这三个线程并行化时需要考虑的：

1. 顶点处理线程的作用是对顶点进行坐标变换，当存在多个顶点处理线程对顶点进行处理时，可能出现处理完毕顶点的乱序问题。举个例子，在处理前假如顶点输入队列的顺序为 {A,B,C,D,E,F,...}，假如有多个线程同时进行处理，则可能出现顶点输出队列的顺序为 {A,C,D,B,E,F,...}，而下一步我们会将顶点输出队列的数据单元重新组装成三角形，此时就存在一个重要问题，如何在组装三角形的环节仍然能按照正确顺序选择正确的三个顶点。已知顶点输入队列按顺序每三个顶点即为一个三角形的三个顶点，那么顶点处理线程如何能存储这样信息呢，是同学们需要解决的第一个问题。
2. 光栅化线程的作用是将顶点数据重新组装成三角形，然后将其光栅化为片元数据单元。那么如何并行化多个光栅化线程的工作呢？我们既可以分开并行化三角形组装以及光栅化，也可以将其作为一个整体进行并行化。哪一种方式更优需要同学们自行探索。
3. 片元处理线程的作用是对片元着色得到像素，这一步需要考虑的是共享数据的同步问题，这个问题已经在 2.2.3 中进行了详细的讨论。

2.2.5 要求

我们在之前的介绍中明确了优化目标是让顶点处理线程、光栅化线程和片元处理线程并行化，并行化主要涉及以下一些内容的更改 (**多线程请用 `std::thread` 实现**):

- `rasterizer_renderer.cpp` 中的 `VertexProcessor::worker_thread` 和 `FragmentProcessor::worker_thread` 这两个函数分别是顶点处理线程和片元处理线程的处理主体。
- `rasterizer.cpp` 中的 `Rasterizer::worker_thread`: 这个函数是光栅化线程的处理主体。
- `graphics_interface.h` 中的 `FrameBuffer` 以及 `Context`: 前者是 frame buffer 的类定义，后者是并行化可能需要修改的一些全局变量。

 **笔记** 除此之外，不同的并行化方式可能会需要修改一些除提到的部分之外的内容。只要保证不增减源文件，只在原有的文件中进行修改即可。

请使用 `Debug` 模式编译并运行程序，渲染一个串行流水线用时大于 4 秒的场景，并在提交时附上串行流水线和并行流水线时的运行时间截图，类似图 2.5。

2.3 提交和验收

提交的实验结果需包含上述要求中提到的 `png` 格式截图，其余提交要求参照基础部分文档的附录 A 的 A.1 部分。

另外，验收时需要现场以 `Debug` 模式编译并运行程序，渲染一个串行流水线用时大于 4 秒的场景，使用串行流水线和并行流水线两种模式并对比时间。在不计入主线程 (UI 线程) 的情况下，串行流水线使用了 3 个计算线程，渲染用时为 T_3 。你所实现的并行流水线若使用 $k (k > 3)$ 个计算线程，渲染用时为 T_k ，则最终得分比例为 $(3 \cdot T_3) / (k \cdot T_k)$ 。直观地说，你需要在至少一种情况下达到线性加速才能获得满分，例如使用 6 个计算线程时用时减少到串行版本的一半，或使用 9 个线程时用时减少到 $1/3$ 。

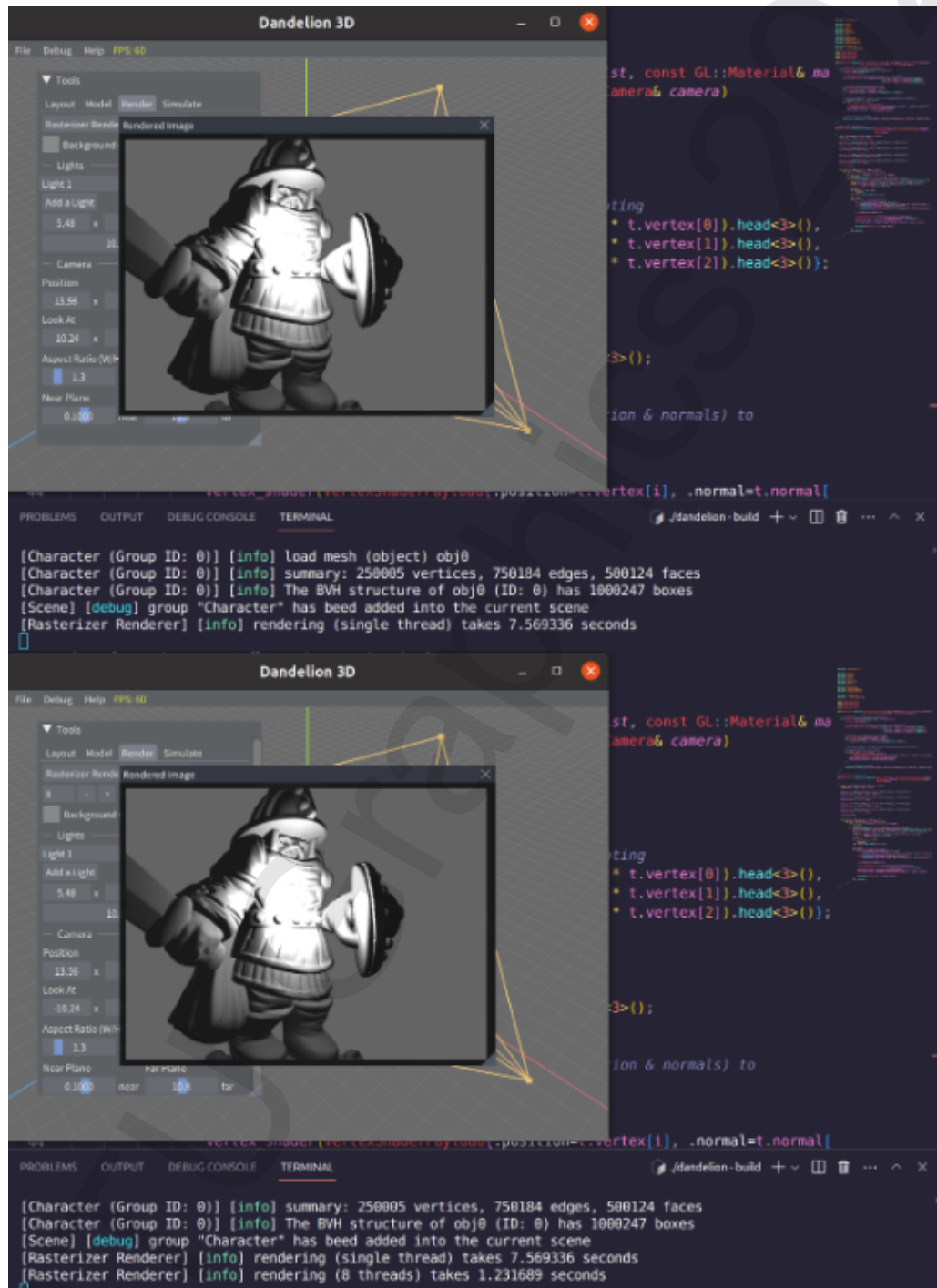


图 2.5: 实验截图

第3章 基于 Shadow Mapping 的阴影实现

阴影是增强真实感的重要手段之一。在图形学的发展历史上，曾经出现过许多种制造阴影的方法，而阴影映射 (Shadow Mapping) 已然占据主流数十年之久，至今仍是游戏等实时渲染领域最主要的阴影渲染方法。本实验 (编号 2.4) 中，你将在第 1 章光栅化管线的基础上，利用深度缓冲实现 Shadow Mapping 算法，从而制造出阴影效果。

3.1 实验内容

这是一个挑战任务，你需要完成的工作是：

- 稍微修改现有的渲染引擎框架，新增用于保存 Shadow Mapping 的输出缓冲。
- 修改当前光栅化渲染管线的渲染流程，实现一个 2-pass 管线，第一个 pass 生成 Shadow Mapping，第二个 pass 渲染图像。

3.2 指导和要求

3.2.1 Shadow Map 的生成方法

我们的课程或是 GAMES 101 这样的在线课程都会讲到：Shadow Map 本质上是从光源处观察时，渲染所得的深度缓冲。这是一个正确但有些模糊的说法，留下了几个细节问题：

- 我们都知道，至少需要一个观察方向和一个向上方向才能确定观察变换 (view transformation)。已知观察点在光源处，那么观察的目标点 (或者观察方向) 在哪里呢？
- 从光源处观察和从渲染视点观察，可见的场景范围肯定是不同的，如何保证第二个 pass 查找 Shadow Map 时一定能找到所有可见像素处的深度？我们应该怎样设置渲染 Shadow Map 的相机呢？

实际上，这些问题一般都不会有特别精确的解决方案。通常我们可以用一些估算的方法来大致确定渲染相机的可视范围，再增加一些余量，确保阴影相机的可视范围足够大。如果尝试渲染之后发现 Shadow Map 覆盖范围不够，那就再放大大一点。在这个实验中，如果你对估算感到头疼，也可以预先将阴影相机的可视范围调得比较大，然后限制渲染相机的可视范围不要太大。

3.2.2 查找 Shadow Map

查找 Shadow Map 的过程发生在片元着色阶段，需要将片元的世界坐标变换到阴影相机对应的图像空间，然后得到此处的阴影深度。

回忆一下课堂上讲 Shadow Mapping 时提到的精度问题——Shadow Map 的最小单位是一个像素，而这已经对应了三维空间中的一个区域。如果不加处理，查找时的误差就会制造出大量条纹，称为 shadow acne (如图 3.1)

这个问题可以通过施加一个较小的偏移量加以解决，如图 3.2 所示。

3.2.3 要求

本实验需要完成以下任务：

- 修改 `RenderEngine` 的定义，新增存储 Shadow Mapping 的第二个 z-buffer，并通过修改 `graphics_interface.h` 中的 `Uniforms` 等方法允许第二个 pass 渲染时访问 Shadow Mapping。

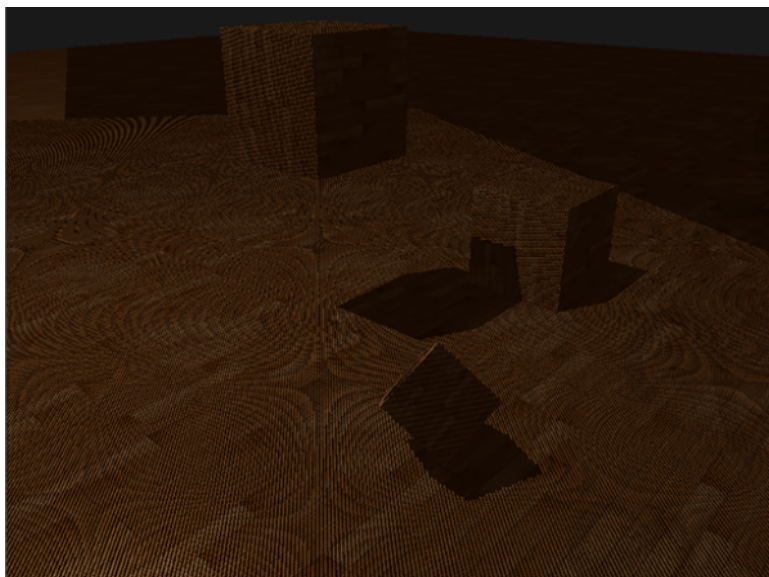


图 3.1: Shadow Mapping 精度有限导致的误差现象 [5]

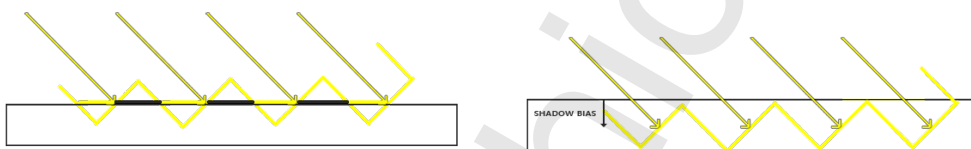


图 3.2: 通过施加偏移消除 shadow acne[5]

- 修改 `RasterizerRenderer::render` 方法，实现 2-pass 渲染。
- 实现用于生成 Shadow Mapping 的片元着色器，并修改当前的 Phong Shading 着色器使之在着色时先查找阴影。

3.3 提交和验收

提交的实验结果包括一张带有正确阴影效果的 png 格式截图，其余提交要求遵循默认规定。本实验不允许增减源代码文件，只能修改现有的文件。

验收时需要解释自己实现 2-pass 渲染的方式和在两个 pass 间传递 Shadow Mapping 的方式；并现场以 Release 模式编译运行程序，渲染 *cow.dae* 放置于 *cube.obj* 上的场景（只需要一个光源，光源位置要能产生阴影）。能产生形状正确的阴影得 3 分，无明显的 shadow acne 等细节问题再得 2 分。

第4章 基于 PCSS 的软阴影效果

第3章中实现的 Shadow Mapping 虽然能生成形状位置基本合理的阴影，但在朴素的 Shadow Mapping 算法中，一个片元只有“在阴影中”与“不在阴影中”两种可能，最终画面上会产生边界锐利的阴影（即“硬阴影”）。然而日常生活中，我们所见到的光源一般是面光源，它们照射物体所产生的阴影边缘是柔和的，这种阴影称为“软阴影”。本实验（编号 2.5）中，你将更进一步地利用 Shadow Map，基于 PCSS (Percentage Closer Soft Shadow) 技术生成更加真实的阴影效果。

4.1 实验内容

这是一个挑战任务，你需要完成的工作是：

- 实现面光源的设置和调整。
- 实现 PCSS 算法，通过多次采样 Shadow Map 估算片元处的可见性，从而生成合理的软阴影。

4.2 指导和要求

4.2.1 使用 PCF 生成柔性阴影边缘

PCF (Percentage Closer Filtering) 是柔化阴影边缘的算法，它的思路非常直接：如果我们扩大在 Shadow Map 上的采样范围，就能知道片元的邻域有多大比例处于阴影中，进而确定这个片元是在阴影中心还是边缘。这样一来，可见性就不再是二值的，而是一个从 0（完全在阴影内部）到 1（完全不在阴影中）的连续值。所用的采样半径越大，阴影边缘的柔化（模糊）范围就越大。

4.2.2 面光源与软阴影

从光学角度讲，点光源会产生硬阴影，而有体积的光源会产生软阴影，可以用“本影”和“半影”的理论加以解释。在阴影范围内，如果某个点处完全看不到光源，那么这一点就是“全黑”的（只被环境光照射）；而如果某个点能看到部分光源，那么这一点处就是偏暗的。“全黑”的点组成了影子中心的部分，称为本影；偏暗的点组成边缘，称为半影。

从影子中心向边缘移动时，光源可见的部分越来越多，半影也越来越亮。直到光源完全可见时，就脱离了阴影范围。根据半影形成的原因，可以推出一个现象：物体离阴影投射表面越近的部分，所投射的半影范围越小，如图 4.1 中钢笔笔尖的影子就比较接近硬阴影。

因此，光源的大小 w_{light} 、物体到光源的距离 d_{blocker} 和阴影投射表面到光源的距离 d_{receiver} 共同决定了半影的大小。用相似三角形来处理，可以得到半影的大小为 [6]

$$w_{\text{penumbra}} = \frac{d_{\text{receiver}} - d_{\text{blocker}}}{d_{\text{blocker}}} \cdot w_{\text{light}}$$

根据计算出的半影大小控制 PCF 的采样范围，就可以产生图 4.1 中的阴影效果了。

4.2.3 要求

更详细的 PCSS 介绍可以参考 GAMES 202 课程 [6] P3/P4 部分。本实验需要完成以下任务：



图 4.1: 真实的阴影 [6]

- 将现有的 `light` 提升为基类，通过继承分别实现点光源 `PointLight` 与面光源 `AreaLight`，并在 UI 界面上增加编辑面光源属性的接口。
- 在第 3 章 2-pass 管线的基础上实现 PCF 和 PCSS 算法，生成合理的软阴影效果。

4.3 提交和验收

提交的实验结果包括一张带有正确阴影效果的 png 格式截图，其余提交要求遵循默认规定。本实验不允许增减源代码文件，只能修改现有的文件。

验收时需要解释自己是如何实现面光源设置、如何实现 PCF 采样和半影范围计算的，并现场以 Release 模式编译运行程序，渲染 `cow.dae` 放置于 `cube,obj` 上的场景（只需要一个面光源，光源位置要能产生阴影）。能设置面光源并产生柔性阴影得 5 分，半影范围正确再得 5 分。

第 5 章 Whitted-Style Ray-Tracing

在前两章中，我们对软光栅化渲染器进行了研究。这一章 (编号 2.6)，我们将对 Whitted-Style 光线追踪进行研究，Whitted-Style 光线追踪是利用折射定律和菲涅尔方程来计算光线与不同材质 (Relection / Reflection and Refraction / Diffuse or Glossy) 物体表面相交时的反射和折射光线方向。那么这二者有什么联系和区别呢？

我们知道，在软光栅化渲染器计算片元着色时，我们使用的是 Blinn-Phong 光照模型，**这种模型无法处理全局效果**。也就是说，Blinn-Phong 光照模型是局部的，渲染出的结果是不考虑真实的阴影情况的。在实现软光栅化渲染器的时候，我们给环境光加上了一个常数颜色，这样的作法是不严谨的，那么真实情况是怎样的呢？就需要引出全局光照的概念。

这里会有几个容易混淆的名词：**直接光照和局部光照、间接光照和全局光照**。首先，点光源与平行光源这种理想光源发出光线到物体表面产生的光照我们将其称为直接光照，而只考虑直接光照效果我们称之为局部光照。与此相对应，在真实世界中，不仅需要考理想光源带来的光照效果，光线还在玻璃、镜子亦或是不那么光滑的物体表面之间发生弹射，这些现象也是光照效果的重要组成部分，我们称之为间接光照，那么考虑了间接光照效果我们称之为全局光照。因此，我们可以说 Whitted-Style 光照模型是全局的。

既然理解了 Whitted-Style 光线追踪是什么以及引入它的原因，那么接下来，就让我们开始更深入地探索 Whitted-Style 光线追踪的实现吧！

5.1 实验内容

在本章中需要实现的 Whitted-Style 渲染器主要可以分为三个模块：

1. **发射一条主射线 primary ray**: 从摄像机成像平面的每个像素发射一条主射线。
2. **判断是否与物体相交**: 本章中使用的是最简单粗暴的遍历方式，即遍历 mesh 中的每一个三角形，判断当前光线是否与该三角形相交，如果相交，则返回交点的表面属性，否则返回 `std::nullopt`。
3. **递归光线跟踪**: 根据物体表面属性，判断是否需要发射二次射线 secondary ray (同学们不需要考虑折射，只需要考虑反射情况，其中 `material.shiness < 1000` 时为 diffuse，否则为 relection)，如果需要发射，则继续递归，直到不需要反射或达到设定的反射次数阈值。

5.2 指导和要求

5.2.1 主射线生成

主射线生成的函数 `generate_ray()` 在 `DANDELION/src/utlis/ray.cpp` 中。我们都知道，定义一条射线，只需要给出其端点和方向。而想要确定其世界坐标系下的端点坐标和方向向量，首先需要确定的是坐标系。如图 5.1:

大家看到这张图的时候有没有一种熟悉感呢？是不是感觉很像在光栅化中的 frustum？其实计算主射线就是一个逆光栅化的过程，在图 5.1 中，以相机建立坐标系，遵循右手定则，**成像平面在-Z 方向**，图中小一点平面的是归一化的成像平面，归一化成像平面距离相机中心的距离为 1，即 $depth = 1$ ，而大一点的是物理成像平面，其长宽和最终成像设置的长宽相等。另一个需要注意的点是像素坐标系 UV，可以看到，在成像平面上，**U 轴与相机坐标系的 X 轴平行且方向相同，V 轴与相机坐标系的 Y 轴平行但方向相反**，即原点 O 位于图像的左上角，另外，相机中心对应的是像素平面的中心，因此，在计算对应的相机坐标系坐标时，**应当减去的是像素平面中心的坐标而非原点坐标**。明确了坐标系之间的关系，那么成像平面与真实的世界坐标系中坐标的关系就应当是：

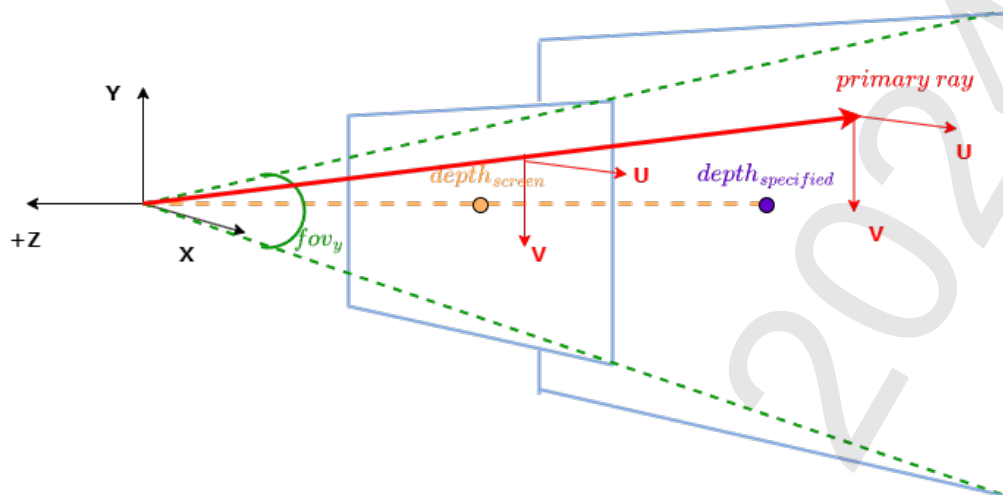


图 5.1: 主射线生成示意图

```
Vector2f pos((float)x+0.5f, (float)y+0.5f);
Vector2f center((float)width/2.0f, (float)height/2.0f);
Matrix4f inv_view = camera.view().inverse()
Vector4f view_pos_specified = Vector3f(pos.x()-center.x(),
    -(pos.y()-center.y()), -depth, 1.0f);
Vector4f world_pos_specified = (inv_view * view_pos);
```

最后还有一个问题，我们虽然知道发射射线经过的像素坐标，但是这个像素坐标实际对应的是物理成像平面的像素坐标，而这个深度我们是未知的，但是我们的归一化成像平面的深度是已知的，即 $depth = 1$ ，但长宽又不知道，那么我们就需要一个额外的信息，那就是视场角 fov 。对于相机来说， fov 是已知的，因为归一化成像平面 $depth$ 已知，我们就能通过视场角计算出归一化成像平面的宽度（也就是高度），也就能够得到归一化成像平面和物理成像平面的比值，进而就可以通过比值，将任何物理成像平面的计算结果，转换到归一化成像平面中了，即：

```
float fov_y = radians(camera.fov_y_degrees);
float ratio = image_plane_height / (float)height;
Vector4f view_pos = ratio * view_pos_specified;
```

5.2.2 判断是否与物体相交的注意点

在一条主射线发出后，我们接下来就是要判断这条射线与哪个物体相交。而判断与哪个物体相交，实质上是判断与哪个物体中的三角形面片相交，当与多个三角形面片相交时，应当选取光线最先相交的那一个。

本章中求解三角形面片相交使用的是简单粗暴地遍历场景中的每一个三角形，但在这个过程中，会有一些细节需要同学们注意。

首先假如我们使用 M-T 算法 (Fast, Minimum Storage Ray/Triangle Intersection) 求解光线与三角形的交点：当我们建立线性方程组求解相交结果时，要保证行列式不为 0，即保证系数矩阵是可逆的。当稀疏矩阵不可逆时，对应的是光线平行于当前遍历三角形面片的情况。其次，当我们得到了求解结果时，应当首先判定 t 是否 < 0 ，我们的光线是一条射线而非直线， $t < 0$ 的情况应当被直接剔除。

 **笔记** 当我们进行大小比较时，由于我们进行的是浮点运算，由于精度损失，常常会出现并不真正等于我们想要的值，有时候会稍大一点或者稍小一点，这个时候需要我们放宽条件，采用一个小小的偏移值 `epsilon` 来解决这个问题。这点在渲染器的编写过程中比较常见。如：

```
// Matrix A is not invertible, indicating the ray is parallel with the
// triangle.
if (std::fabs(det) < eps) {
    return std::nullopt;
}
//the direction of the ray is not correct
if (t < eps) {
    return std::nullopt;
}
// Ensure result.t is strictly less than the constant `infinity`.
if (result.t - infinity < -eps) {
    return result;
}
```

在求得交点后，同样由于精度损失，计算得到的相交点坐标往往并不会正中理论上的点位置，而是会向外或者向内一点，直接使用会产生噪声值，这时也是通过使用法线方向偏移值 `epsilon` 来解决这个问题。但需要注意的是：在考虑反射和折射时，偏移方向会有所不同。如图5.2

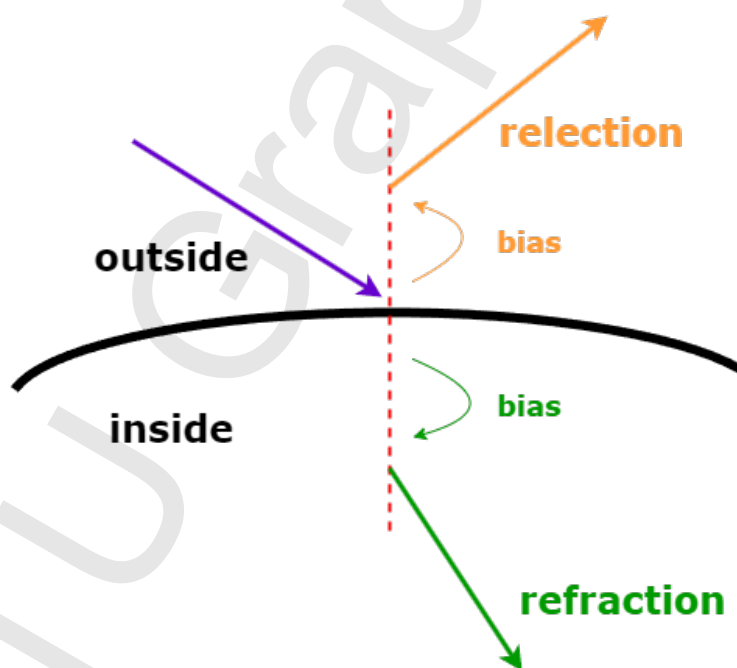


图 5.2: 相交点偏移

从图5.2我们可以发现，如果出射光线是朝外侧发射的，则交点需要向外偏移；如果出射光线向内侧发射，则交点需要向内侧偏移，那么如何判断是朝外侧还是内侧呢？可以使用出射光线方向与法线点乘，小于零为内侧，大于零为外侧。在本章中，我们只实现反射，不考虑折射。

5.2.3 阴影的判定

在本章的开头我们就明确了 Whitted-Style Ray-Tracing 是全局光照模型。因此，如何判断阴影呢？其实和递归光线一样，当我们获得了一个光线与物体的交点之后，我们认定：**当这个点被其他物体挡住所有光线直射时，它就在阴影里**。因此，我们只需要从交点发射一根光线打向光源，用以判断是否被其他物体所遮挡，但是这里与我们的递归光线追踪有一些逻辑上的差异：

- 在递归光线时，我们判断光线与物体相交时，才会进行后续的工作，如反射或者着色；而判断阴影时，当射线与物体相交时，说明被遮挡了，而未相交才应该进行着色。同学们在实现时需要注意区分。
- 计算的相交结果的 t 值不仅需要和光线递归时一样 >0 ，还应当小于和光源之间的距离。
- 阴影只在需要着色时需要发射射线进行判定，而光线递归是在每一次递归都需要进行。

 **笔记** 细心的同学肯定会意识到，我们判定阴影的方式其实是不严谨的，因为虽然被物体挡住，但是当我们考虑间接光照时，其实所有弹射光的物体均可以被当作光源，而非只有我们手动添加的光源才能称为光源。而本章的 Whitted-Style 渲染器显然具有局限性，不仅在阴影上不够严谨，同学们在实验中会发现，还有例如反射不全、diffuse 物体的反射没有被考虑等问题，这些问题都有各种解决办法，包括渲染部分的选做实验路径追踪渲染器。感兴趣的同学可以自行谷歌并创造出满意的结果。尽管如此，Whitted-Style 仍然可以称为一个完整且生动的渲染器。

5.2.4 要求

本章的实验需要完成以下提到的几个函数：

- `whitted_renderer` 中的 `WhittedRenderer::fresnel`：该函数用于使用菲涅尔定理计算反射光线的剩余量
- `whitted_renderer` 中 `WhittedRenderer::trace` 挖空的部分：在本章中，该函数应当调用 `DANDELION/src/utils/ray.cpp` 中的 `naive_intersect` 函数进行与三角形面片的求交
- `whitted_renderer` 中的 `WhittedRenderer::cast_ray`：该函数实现了 Whitted-Style Ray-Tracing 的光线递归
- `utils/ray.cpp` 中的 `WhittedRenderer::naive_intersect`：该函数按顺序遍历 `mesh` 中的所有面片，判断是否与当前光线相交，最终取交点中 $t>0$ 且 t 值最小的点作为结果返回

在之前的实验中，我们都是默认使用的 `ray.cpp` 的静态链接库版本，在本章中，同学们需要取消掉 `CMakeLists.txt` 中对 `src/utils/ray.cpp` 的注释，将 `target_link_libraries(\${PROJECT_NAME})` 中的 `dandelion-ray-debug` 和 `dandelion-ray` 这两行注释掉，以 Release 模式编译并运行程序，渲染一个自定义场景，要求场景中必须能体现出 Whitted-Style Ray-Tracing 的阴影和反射，类似图 5.3

5.3 提交和验收

提交的实验结果需包含上述要求中提到的 png 格式截图，其余提交要求参照基础部分文档的附录 A 的 A.1 部分。

另外，验收时需要解释自己实现的渲染器；现场以 Release 模式编译并运行程序，渲染一个自己准备的能体现出 Whitted-Style 渲染器特色的场景。非镜面物体的位置和颜色正确得 10 分，阴影、反射现象正确再得 10 分。

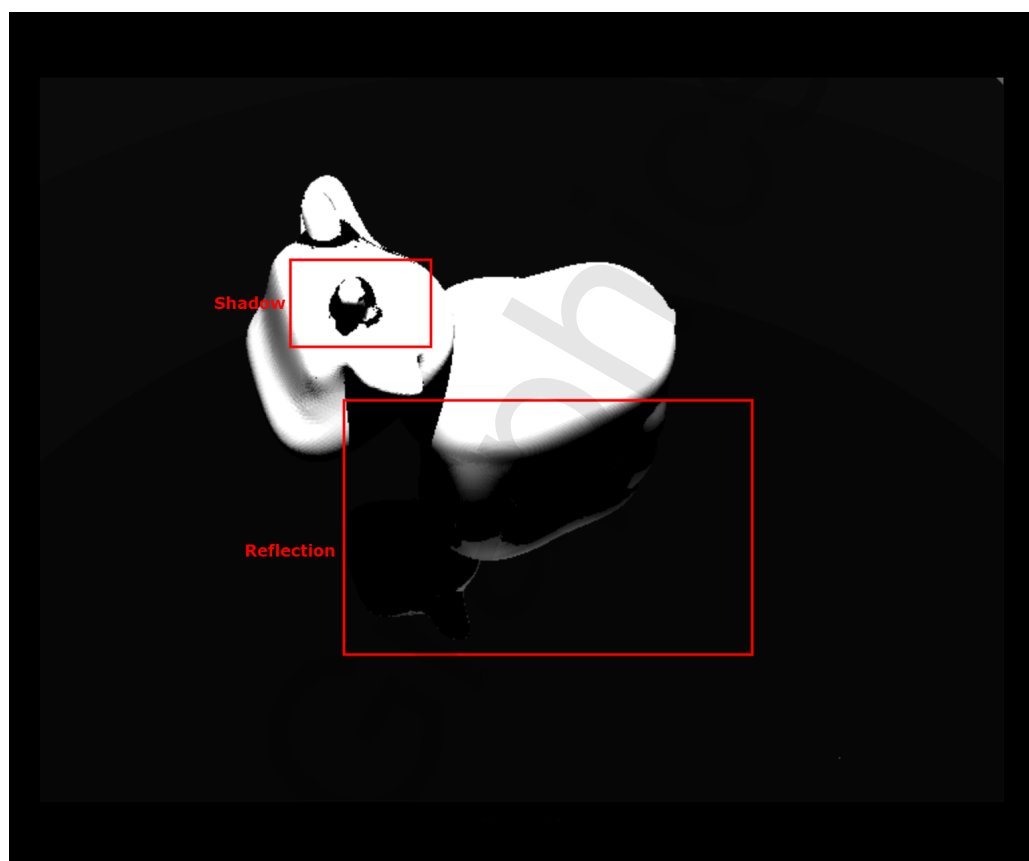


图 5.3: 实验截图

第6章 用BVH加速光线求交

本章(编号 2.7)中,我们将使用几何数据结构加速光线求交。在上一章中,我们实现了 Whitted-Style Ray-Tracing 渲染器。我们会发现 Whitted-Style 渲染器的耗时在 debug 模式下非常长,使用单线程渲染一个稍微复杂的场景,就可能需要半个小时以上的时间。事实上,我们完全可以让这个时间减少很多。仔细分析一下我们实现的光线求交算法。我们采用的是暴力遍历所有物体的所有三角形面片来判断是否与当前射线相交。但实际上,很多相交的求解都是没有意义的,从直观上讲,我们的射线一定只可能命中在它的方向附近有限区域内的物体,那么我们就得想办法表示“一定区域内”这个概念,在这个需求下,有一系列的算法优化,其中一个就是 BVH (Bounding Volume Hierarchies 层级包围盒)。

具体来说, BVH 的核心思想就是使用体积略大而几何特征简单的包围盒来近似描述复杂的几何对象,且这种包围盒是嵌套的。我们只需要对包围盒进行递归的相交测试,就可以越来越逼近实际对象。说到这里,同学们应该会自然而然地想到树的层级结构。没错, BVH 就是通过树结构来进行实现的。了解了 BVH 大概是什么,接下来就让我们更进一步地从 BVH 的建立和使用去感受它的简洁和高效叭。

6.1 实验内容

在本章中,我们需要将在第三章中 Whitted-Style 渲染器中 `WhittedRenderer::trace` 函数中调用的 `naive_intersect` 函数替换为 `object->bvh->intersect` 函数。

同学们需要重点理解和掌握的是 BVH 的建立,射线与 BVH 节点的相交以及射线与 AABB(Axis-Aligned Bounding Box 轴对齐包围盒)的相交。这里注意区分射线与 BVH 节点的相交和射线与 AABB 的相交:前者重点在于递归 BVH 树,求解最后相交的结果;后者则只需要判断当前光线与 AABB 是否相交。

本章涉及的知识点较少,接下来我们会带着大家梳理一遍整个 BVH 的建立,并对 BVH 求交中的一些注意点进行说明。只要大家能够清楚地理解 BVH 的结构以及其求交原理,就都能轻松地为自己的 Whitted-Style 渲染器实现出一个合格的 BVH 加速结构。

6.2 指导和要求

6.2.1 建立 BVH 的全流程

BVH 本质上就是一棵二叉树,我们都知道二叉树的建立过程以及搜索方法, BVH 与之保持一致,同学们只需要在过程中体会以下几点:

- BVH 树的节点的含义是什么?
- BVH 划分左右节点的依据是什么?
- BVH 树的每个节点需要存储哪些数据?

那么接下来就让我们一起来梳理一遍 BVH 树的建立全流程:

Step 1: 创建一个根节点

Step 2: 将场景中所有的物体用一个轴对齐包围盒(后续均用 AABB 表示)包住

Step 3: 将这个初始的 AABB 分配给创建的根节点, 整个流程第 1-3 步如图6.1

在理解图6.1时, 要注意以下几个细节:

- 图中的每个形状表示的就是叶子节点的 AABB 要包裹的物体, 这个物体的定义很抽象, 有的时候可以是“真正的物体”, 比如一只猫, 一个茶壶等; 有的时候可能是一组面片, 比如一只猫的耳朵, 一个茶壶的盖子; 也可能是一个面片, 比如一个三角形面片或是长方形面片。细分到什么程度全由开发

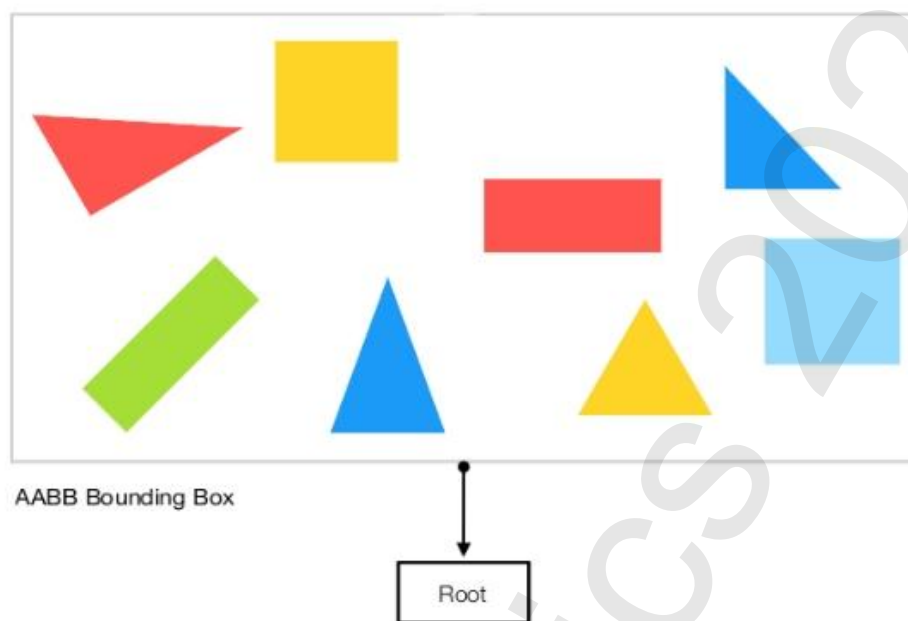


图 6.1: BVH 建立的第 1-3 步，图片来自 [3]

者自己决定，在本章实验中，我们是将三角形面片作为叶子节点的 AABB 包裹的物体的。

- 图中的物体看着相互之间毫无联系，但事实上，相近的物体常常具有某些共同的属性。比如在我们的渲染器中，由于物体实际是三角形面片，那么相邻的面片通常都归属于同一个 object(为了和 BVH 中抽象的物体区分，提到真正的物体时都会使用 object) 的同一个 mesh。为什么需要说明这一点呢？我们的 BVH 树不是只有叶子节点，如果我们明确了他们之间的联系，我们就能更好地理解非叶子节点是什么。
- 理解了上面提到的两点，那么最后还有明确一下根节点是什么。在我们的整个实验框架中，BVH 被认为是 object 的一个属性，所以根节点应当是一个 object，用不严谨但通俗易懂的说法就是：场景中有多少个 object 就有多少个 BVH 树。因此，假如我们需要使用 BVH 来进行求交，应当使用 `object->bvh->intersect` 这样的写法。这样划分 BVH 的好处是我们的模型是可以实时被缩放旋转的，假如我们建立的是场景的 BVH，那么一个物体的平移缩放旋转，就会导致整个场景的 BVH 重新建立，而 BVH 的建立是相对较为耗时的，尤其是较为复杂的场景。而假如我们以物体为单位建立 BVH，只要我们在 model 坐标系中进行求交等操作，BVH 就不需要随着物体的平移缩放旋转，重新建立整个场景的 BVH，即整个 BVH 树的结构不会发生任何改变。

Step 4: 寻找当前 AABB(x,y,z) 中的最长轴，并且按最长轴分量从小到大进行排序，如图 6.2

Step 5: 寻找一个中间点将整个 AABB 沿这个最长轴一分为二。这里肯定会有同学对中间点或是一分为二的定义有疑惑。我们选择的是排序之后序列中，排在中间的那个索引值作为划分，也就是说，当中间的物体发生重叠时，并不会出现无法划分左右或者同时划分为左右的现象，因为我们划分的依据是按最长轴的坐标分量排序后的索引值，这个索引值与物体一一对应，不会出现共享或者冲突。

Step 6: 对一分为二之后的两边，各自新创建新的 AABB 包裹住各自新划分的物体。

Step 7: 将新创建的 AABB 传递给当前 BVH 树新创建的左右子节点。

流程第 5-7 步如图 6.3

这里需要注意的是，虽然逻辑上是划分之后新创建的 AABB，但是假如同学们在递归建树时采用的是以下这种写法：

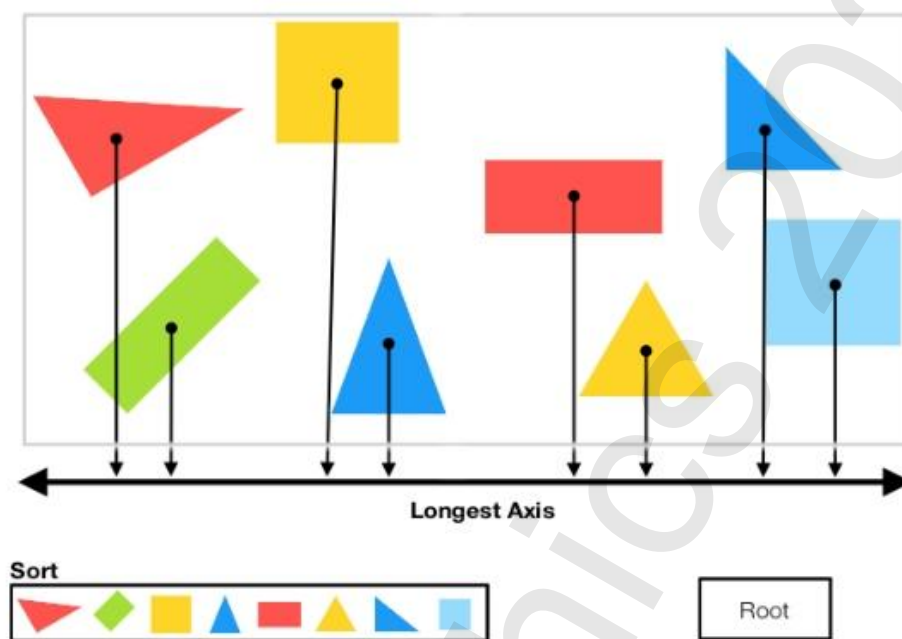


图 6.2: BVH 建立的第 4 步, 图片来自 [3]

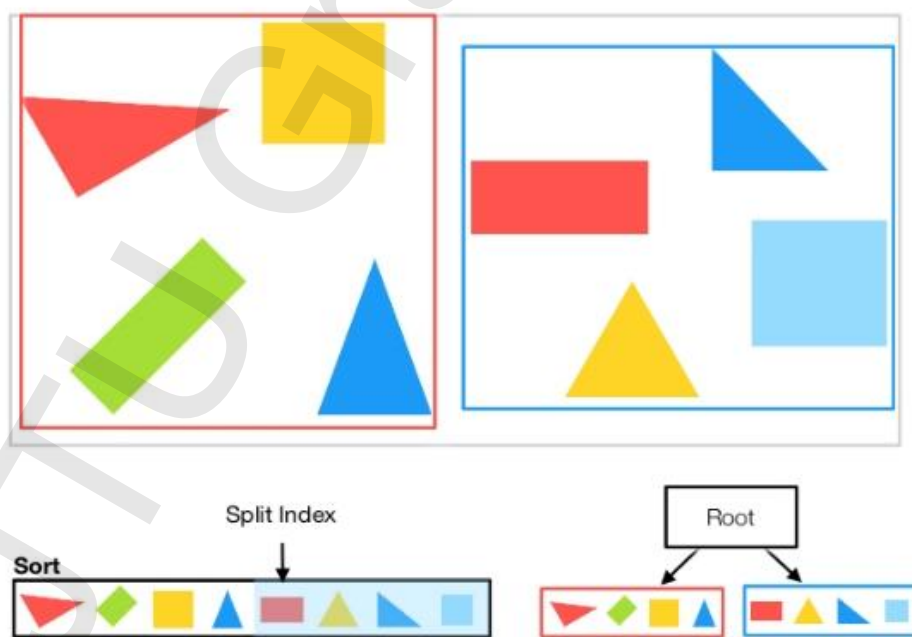


图 6.3: BVH 建立的第 5-7 步, 图片来自 [3]

```
node->left = recursively_build(left_faces_idx);
node->right = recursively_build(right_faces_idx);
node->aabb = union_AABB(node->left->aabb, node->right->aabb);
```

那么事实上的代码逻辑应该是在一开始建立根节点时，就有为每一个面片建立一个 AABB，即：

```
for(size_t i=0; i<faces_idx.size(); i++){
    aabb = union_AABB(aabb, get_aabb(mesh, faces_idx[i]));
}
```

Step 8: 之后就是继续在左右子树上重复第 5-7 步的操作，如图 6.4 和 6.5

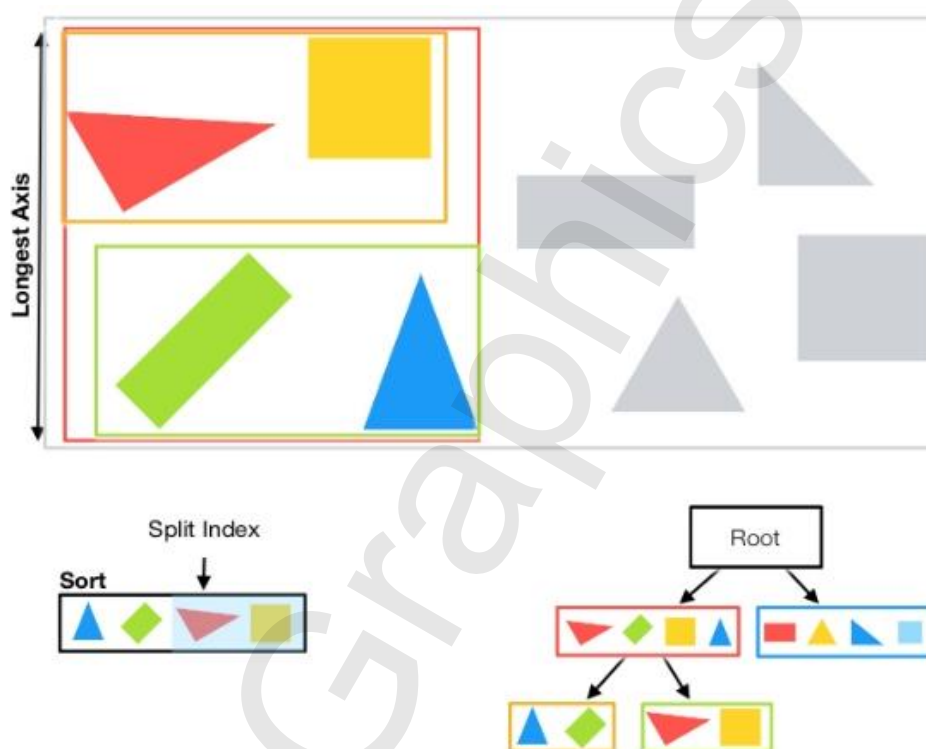


图 6.4: 对左子树重复第 5-7 步，图片来自 [3]

那么递归的边界条件是什么呢？在本章实验中，我们决定当 `faces_idx.size()==1` 时，不再继续细分，即将节点中的三角形面片数有且仅有一片作为递归的边界条件。在实际应用中，这个边界条件是可以自行设定的，比如面片数小于等于一个指定值，或者递归的深度大于等于某个指定值等。

6.2.2 BVH 相交的注意点

在前面一节，我们已经梳理清楚了 BVH 的建树过程，接下来使用射线与建立好的 BVH 求交也就只需正常地遍历 BVH 树的层次结构即可。但是 BVH 求交的过程有一些关键点需要同学们注意，否则可能得到错误的结果：

- 分清 `AABB::intersect` 函数与 `BVH::ray_node_intersect` 函数还有 `ray_triangle_intersect` 函数的区别，不清楚的同学可以通过阅读[开发者文档：类说明](#)，来分辨这三个函数的作用
- `ray_triangle_intersect` 是发生在 model 坐标系中的，原因在 4.2.1 中已经详细说明。但在该函数外

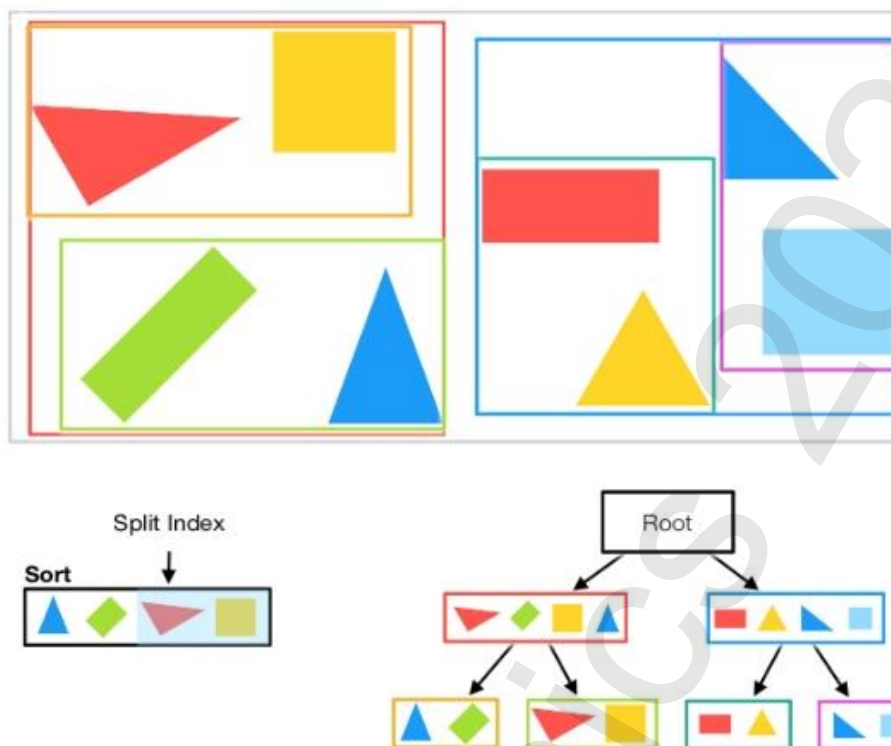


图 6.5: 对右子树重复第 5-7 步, 图片来自 [3]

部, 我们都采用相机坐标系, 于是就存在对返回结果坐标系的变换, 这里同学们需要特别注意法线的从模型坐标系到相机坐标系的变换。

- 还有一个需要注意的地方是在 `AABB::intersect` 函数中, 由于我们的最终包裹物体仅为一个三角形面片, 因此常常会出现 AABB 非常扁平的情况, 这种情况由于精度损失, 可能出现 $t_{enter} - t_{exit} > 0$ 的情况, 同学们可以自行思考解决方法。

6.2.3 要求

本章的实验代码均在 `DANDELION/src/utlis/` 目录下, 请完成下面提到的几个函数:

- `aabb.cpp` 中的 `AABB::intersect`: 该函数用于判断当前射线是否与当前 AABB 相交
- `bvh.cpp` 中的 `BVH::recursively_build`: 该函数用于递归建立 BVH
- `bvh.cpp` 中的 `BVH::ray_node_intersect`: 该函数用于使用发射的射线与当前节点求交, 并递归获取最终的求交结果
- `ray.cpp` 中的 `ray_triangle_intersect`: 该函数用于求解一个射线与三角形面片的求交结果

在之前的实验中, 我们都是默认使用的 `bvh.cpp`、`aabb.cpp`、`ray.cpp` 的静态链接库版本, 在本章中, 同学们需要取消掉 `CMakeLists.txt` 中对 `src/utlis/ray.cpp`、`src/utlis/aabb.cpp`、`src/utlis/bvh.cpp` 的注释, 将 `target_link_libraries()` 中的 `dandelion-ray-debug`、`dandelion-ray`、`dandelion-bvh-debug` 以及 `dandelion-bvh` 这四行注释掉, 以 Release 模式编译并运行程序, 添加 `cow.dae` 模型, 并在工具栏中勾选 `Debug⇒Debug Options⇒Show BVH`, 获得结果并得到类似图 6.6 的截图。

6.3 提交和验收

提交的实验结果需包含上述要求中提到的 png 格式截图, 其余提交要求参照基础部分文档的附录 A 的 A.1 部分。

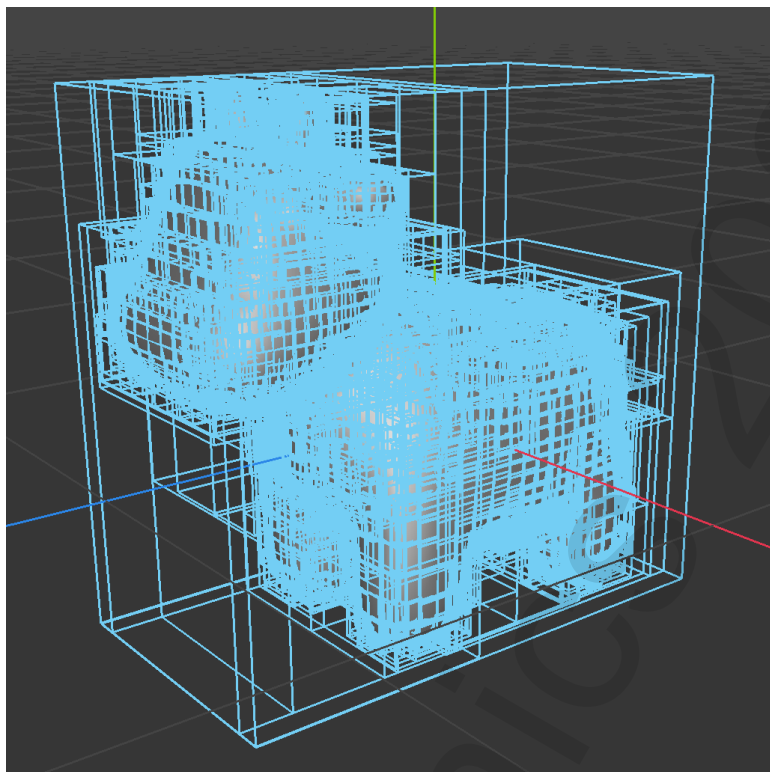


图 6.6: 实验截图

另外，验收时需要解释自己构造 BVH 的过程；现场以 **Release** 模式编译并运行程序，添加 *cat.obj* 模型并展示。能绘制出正确的 BVH 得 5 分，渲染结果正确再得 10 分。

参考文献

- [1] Kayvon Fatahalian. *Graphics and Imaging Architectures*. Tech. rep. CMU, 2011.
- [2] Kok-Lim Low. *Perspective-Correct Interpolation*. Tech. rep. Department of Computer Science University of North Carolina at Chapel Hill, 2002.
- [3] Harold Serroano. *Visualizing the Boundary Volume Hierarchy algorithm*. 2016. URL: <https://www.haroldserrano.com/blog/visualizing-the-boundary-volume-hierarchy-collision-algorithm>.
- [4] Joey de Vries. *Hello Triangle*. published under CC BY 4.0 license. 2014. URL: <https://learnopengl.com/Getting-started/Hello-Triangle>.
- [5] Joey de Vries. *Learn OpenGL - Shadow Mapping*. 2014. URL: <https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>.
- [6] 闫令琪. *GAMES 202 高质量实时渲染*. 2021. URL: <https://www.bilibili.com/video/BV1YK4y1T7yY/>.