基于 OpenGL 的光线追踪器

在着色器 GLSL 中实现

从零开始构建的全教程

Dezeming Family

Dezeming

Art. No 0
ISBN 000–00–0000–00–0
Edition 0.0

Cover design by Dezeming Family

Published by Dezeming
Printed in China

目录

前言及简介

DezemingFamily 系列文章和电子书**全部都有免费公开的电子版**，可以很方便地进行修改和重新发布。如果您获得了 *DezemingFamily* 的系列电子书，可以从我们的网站 *[https://dezeming.top/]* 找到最新的版本。对文章的内容建议和出现的错误也欢迎在网站留言。

0.1 本文前言

将 Ray Tracing 三部曲 [1, 2, 3] 实现到 GLSL 中是一个具有一定挑战性而且很有意思的工作。本文从这三本小书和 Learning OpenGL[4] 出发，逐步构建一个基于 OpenGL 实现的光线追踪器，将整个体系构建的过程进行描述和总结。

本文的基础是 Ray Tracing 三部曲的至少前两本，以及 Learning OpenGL 的至少前三章。读者应当把这些内容学习至少两遍，否则很难掌握熟练。关于基础书籍上出现的内容，本文会告诉读者去哪里复习和了解，但为了更好地学习本文，这些基础内容应该提前掌握好。

本文会每小节都附带有源码，保证在每节之间的改动都非常小，使得读者可以轻松衔接和学习。最好的学习方式是按照本文的顺序，自己根据新的代码来填充和扩展旧代码，这样可以明白每个章节之间具体改动的代码内容。

本文的源码可以在 DezemingFamily 网站上找到。在附带的源码中，所有用到的资源都放在了 Data 目录下，一些并不是很大的工具库（例如 stb_image.h）放在了 Tool 目录下。有些用到的库，比如 assimp 或 glm、glad、glfw 则可以根据 [4] 中环境搭建部分从官网下载并使用。

本文开始写于 6 月 5 日，初版完成于 2022 年 6 月 12 日。

1. 初步构建环境

本章节搭建基础的 *OpenGL* 开发环境。从 *Learning OpenGL* 的项目开始，逐步增加一些光线追踪中可能需要的工具到光线追踪器中。本节最后我们可以实现一个最简单的基于 *Phong* 的光线追踪器，场景中只渲染一个球。

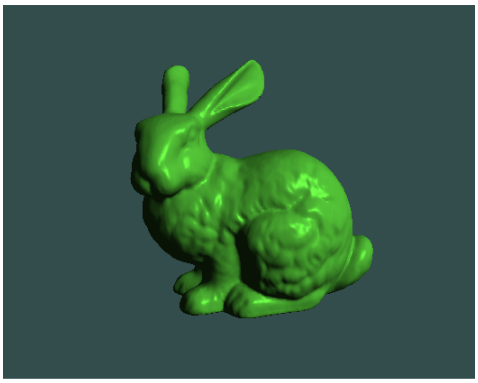
1.1 开发环境搭建

本节搭建开发环境

开发环境第一步：最基本的 OpenGL 环境搭建。我使用 Visual studio(VS) 2015 来进行开发，不过，我们的项目是可以跨平台的，而且并不限制 VS 的版本。开发环境的搭建过程可以参考 [4] 的前几个章节，我们需要用到 GLFW 和 GLAD 库。搭建好环境，能把程序 [chapter-1/Source1] 的窗口显示出来，最初的开发环境的搭建就完成了。

开发环境第二步：GL 数学库和图像加载。之后开始升级开发环境，首先加入 glm 库，为了方便，我直接将 glm 库放在了当前工程目录下。同时，需要 stb_image 这个图像解析库，在 [4] 的纹理章节中有介绍。然后我们应用 GLSL 语言，可以参考 [4] 中的“入门-着色器”一章节，在这里获得 shader.hpp 文件（我们的 Source 源码中也会附带该文件）。如果能编译运行 [chapter-1/Source2] 的源码，则说明当前第二步已经完成。

开发环境第三步：模型加载。虽然一开始我们也用不到很复杂的模型，但为了完整性，以及对 [4] 做一个回顾，所以还是要实现一下模型加载。我们照例使用在 [4] 中使用的 Assimp 库，程序代码可见 [chapter-1/Source3]，我们加载 bunny 模型并使用 phong 光照（参考 [4] 的“光照-基础光照”），编译通过后，运行可以得到下面的效果：



这说明我们的基本环境已经全都搭建完毕了。

为了接下来的工作更顺利，务必将 [4] 的前三章，即入门、光照和模型加载掌握熟练，初学者应当尽量学习两遍以上，这样才能有能力去实现光追系统。实现光线追踪器要对整个工程进行大量的更改和优化，因此前面的这些内容都是基础。到后面，大量的工作都是实现在片段着色器上实现的，主函数只需要进行调用。

1.2 初步结构设计

本节设计好的代码可以参考 [chapter-1/Source4]。

为了将光线追踪使用 OpenGL 来实现，我们需要先简单设计一下结构，方便其他工具的引入和测试。

首先，我们先要尝试在屏幕上渲染一个恰好充满整个屏幕的矩形，这样，这个矩形就可以作为我们光线追踪成像的“画布”。

定义屏幕矩形的位置坐标和纹理坐标（注意，位置值在 –1 或 1，表示标准化空间的坐标，模型-观察-投影变换以后的矩形如果长宽范围都在 –1 或 1 上，则正好充满整个成像面）：

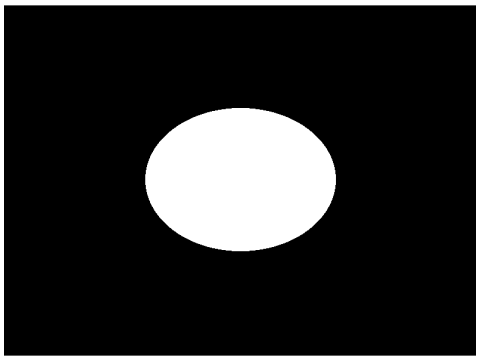
```
const float ScreenVertices[] = {
  //位置坐标(x,y)      //纹理坐标
  //三角形1
  -1.0f, 1.0f,   0.0f, 1.0f, //左上角
  -1.0f, -1.0f,  0.0f, 0.0f, //左下角
  1.0f, -1.0f,   1.0f, 0.0f, //右下角
  //三角形2
  -1.0f,  1.0f,  0.0f, 1.0f, //左上角
  1.0f, -1.0f,   1.0f, 0.0f, //右下角
  1.0f,  1.0f,   1.0f, 1.0f  //右上角
};
```

将坐标绑定到 VBO 和 VAO 中，过程可以在 [chapter-1/Source4] 找到，相关知识可以在 [4] 的“入门-你好，三角形”一章节中找到。

在 GLSL 程序中，片段着色器中将纹理坐标距离 $[0.5, 0.5]$ 小于 0.2 的区域设置为白色。

同时，在 main 函数中，深度测试就可以取消了，包括清空深度 Buffer。

显示的结果是：



1.3 随机数生成

本节设计好的代码可以参考 [chapter-1/Source5]。

我花了很长时间尝试各种低差异序列的生成方法，GLSL 的功能比 CUDA 还弱，不支持 Union 也不支持指针，所以很多以前我在 CPU 或者 CUDA 中使用的低差异序列生成方法都不再能用了。最后，我在 [6, 7] 中找到一些不错的方法，这里采用的 Thomas Wang 实现的随机序列生成方法：

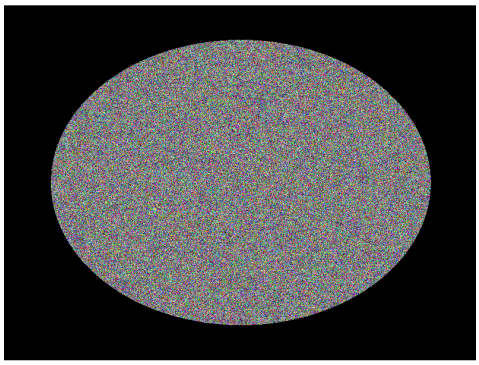
```
uint wseed;
float randcore(uint seed) {
  seed = (seed ^ uint(61)) ^ (seed >> uint(16));
  seed *= uint(9);
  seed = seed ^ (seed >> uint(4));
  seed *= uint(0x27d4eb2d);
  wseed = seed ^ (seed >> uint(15));
  return float(wseed) * (1.0 / 4294967296.0);
}
float rand() {
  return randcore(wseed);
}
```

在 glsl 的 main 函数中调用如下：

```
void main() {
  // 根据坐标将种子初始化
  wseed = uint(float(69557857) * (TexCoords.x * TexCoords.y));
  // 在圆内使用随机数填充
  if(distance(TexCoords,vec2(0.5,0.5)) < 0.4)
    FragColor = vec4(rand(), rand(), rand(), 1.0);
  else
```

```
8     FragColor = vec4(0.0, 0.0, 0.0, 1.0);
9 }
```

得到结果为：



1.4 代码封装

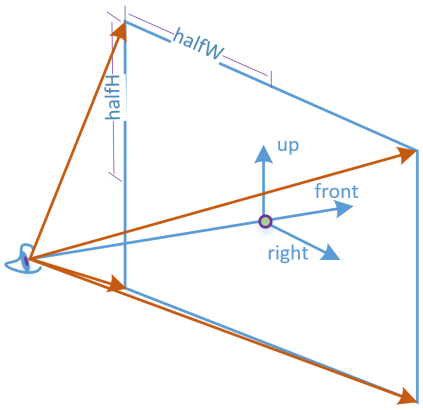
本节设计好的代码可以参考 [chapter-1/Source6]。

在开始构建本小节时，我陷入了纠结，一方面，我想先实现一个最简单的光线追踪器，渲染一个球出来；另一方面，我又想先封装一个相机类，这样后续实现会更方便。纠结了一段时间后，我觉得还是先实现相机比较重要。

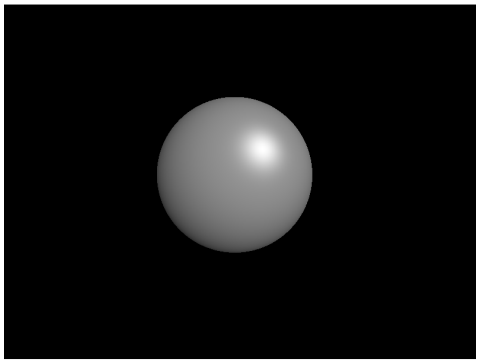
在封装相机之前，我们先建立一个屏幕类，把之前渲染的屏幕顶点封装一下。为此，我们定义了 RT_Screen.h 文件以及 RT_Screen 类，然后把屏幕顶点绑定以及渲染屏幕的函数都实现在了 RT_Screen 类中。

我们还把时间的计算简单地封装到了 timeRecord 类中，用来记录每帧计算的时间。

然后定义 Camera.h 头文件以及 Camera 类。该类我们定义了 fov 以及 halfW 和 halfH。我们会在着色器里手动生成相机每个像素 Ray 的方向。当然，也可以只定义矩形屏幕四个顶点的采样方向作为纹理，其他方向可以根据插值来得到（我们以后再考虑实现这种方式）：



我们按照 [1] 的方式，得到渲染中的屏幕的左下角（即片段着色器中的 leftbottom），然后根据当前纹理坐标来访问到每个像素。光线追踪并与球体相交的过程可以参考 [1] 的前四节，我们实现了一个简单的类似 Phong 的照明。本节代码实现以后的效果如下：



当然，现在封装的相机其实并不是很完备，但最起码我们当前得到的渲染器已经可以算是最简单的光线追踪了。

1.5 接下来的任务

现在还有什么问题？关于随机数有一个很重要的问题：我们每帧产生的随机数序列都是固定的——也就是说，由于我们初始种子都是固定值，导致每次渲染时使用的随机序列都是固定的序列。解决方法并不难，比如每轮渲染都设置不同的 rand1 值：

```
1   wseed = uint(float(rand1) * (TexCoords.x * TexCoords.y));
```

这样可能是可以的，也可能是不可以的：现在我们无法判断每轮渲染中产生的随机序列能够有较大差异。后面我们会再进行设计。

另外，当涉及随机采样时，如果相机或物体不动，我们希望渲染的结果逐帧累加，这个过程应该怎么实现？

其次，渲染中可能会需要大量的面片数据（比如 bunny 模型），这些数据在使用 OpenGL 管线时是绑定到 VAO 中，在顶点着色器中使用。但光线追踪则不再依赖顶点着色器，此时这些数据应该存储在哪里，以及如何在 GPU 中使用？

2. 光线追踪器的基本过程实现

本章节开始实现光线追踪过程，到本章的最后，我们就能实现一个最简单的 *Whitted-style* 的光线追踪器。

2.1 帧缓冲对象与管理

本节代码见 [chapter-2/Source1]。

多帧累积的效果我们采用 OpenGL 的帧缓冲对象 FBO（见 [4] 的“帧缓冲”一节）来实现。但是由于该部分内容相对比较难理解，所以没有提前了解过的读者也可以直接看本文。使用帧缓冲的意义如下：

- OpenGL 有一个默认的帧缓冲，就是一般我们渲染得到图像的缓冲区。
- 我们还可以自己定义其他的帧缓冲，让 OpenGL 把渲染结果放入到我们自己定义的帧缓冲区。
- 帧缓冲可以绑定到纹理上，也就是说，它可以把渲染得到的结果放入到一个纹理中。
- 我们可以在每帧的片段着色器中将历史帧积累的结果读取出来，然后累积。

我们在 RT_Screen.h 中新建了一个 ScreenFBO 类，该类负责管理帧缓冲区，包括创建（configuration 函数）、绑定当前缓冲区（Bind 函数）、解绑定（unBind 函数）、把帧缓冲区绑定到纹理（BindAsTexture 函数）以及删除帧缓冲区（Delete 函数）这些功能。

之后，我们又定义了一组新的着色器：ScreenShader。并且把以前的着色器改名为 RayTracerShader。这两个 Shader 中，RayTracerShader 负责做光线追踪，获得当前帧的结果；ScreenShader 负责将当前帧与历史帧的结果合并，并显示到屏幕上。

本节并不包括多帧积累，重点在于构建 ScreenFBO 及其方法。main 函数中的功能就是先渲染到我们自定义的 FrameBuffer 中，然后再通过采样纹理转到输出窗口的 FrameBuffer 中。

为了记录渲染的轮数，我们在相机类 Camera 中定义了 LoopNum，当相机位置和角度变化时，该值归 0。每帧在渲染前，该值都会加 1。

2.2 多帧累积

本节代码见 [chapter-2/Source2]。

一个 FBO 好像还不够，我们需要两个 FBO 来分别存储当前帧渲染结果和历史帧累积结果。为了方便起见，两个 FBO 我们定义到一个类中进行管理，也就是 RenderBuffer 类。该类中，定义了两个 ScreenFBO 对象，fbo[0] 和 fbo[1]，渲染中的顺序如下：

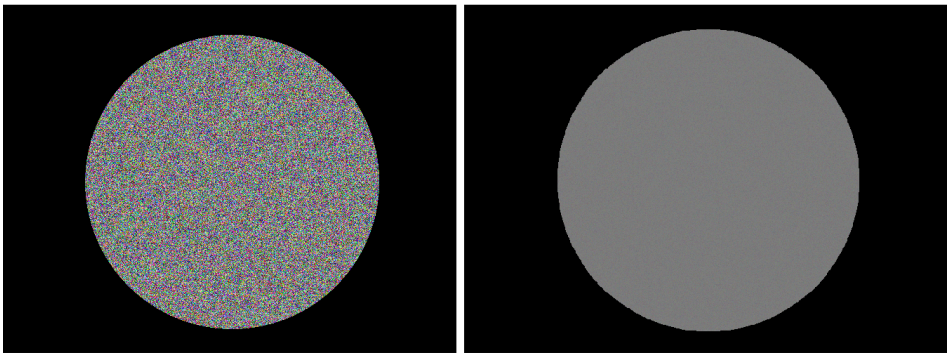
- 第 1 帧，渲染到 fbo[0]（当前帧）中。
- 第 2 帧，渲染，融合 fbo[0]（历史帧）到 fbo[1]（当前帧）中。
- 第 3 帧，渲染，融合 fbo[1]（历史帧）到 fbo[0]（当前帧）中。
- 第 4 帧，渲染，融合 fbo[0]（历史帧）到 fbo[1]（当前帧）中。

注意，当前帧索引的变量放在了 Camera 类中，即 LoopNum，当相机角度或者位置发生变化时，LoopNum 的值就变为 0，也就是重新开始计数。当 LoopNum 在交互响应函数中归 0 后，在渲染前会加 1，因此 LoopNum 在渲染中的最小值是 1。从上面的顺序中看出，当前帧索引为奇数时，使用 fbo[0] 进行渲染，fbo[1] 存储的是历史帧。当前帧索引为偶数时，使用 fbo[1] 进行渲染，fbo[0] 存储的是历史帧。

我们还定义了 Tool.h 文件，该文件用于在 CPU 中产生随机数。我们在 RayTracerFragmentShader 中定义了一个随机数 randOrigin，用来得到每个像素的初始随机数：

```
1  wseed = uint(randOrigin * float(6.95857) * (TexCoords.x * TexCoords.
     y));
```

本节的代码运行表现就是生成一个随机颜色填充的圆（如下图左），然后通过逐帧累加平均，得到平均以后的灰色的圆（如下图右）。



有了这些基础，我们目前应该就可以很容易地实现 [1] 中的全部内容了，但为了我们的程序结构保持简洁，我们并不着急实现多种材质和纹理，而是先实现三个金属球来看一下效果。

2.3 多个物体

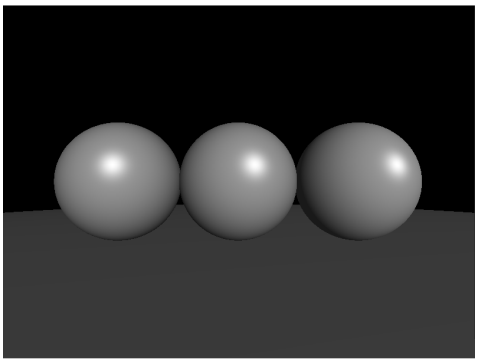
本节代码见 [chapter-2/Source3]。

我们需要先在片段着色器里定义一个 Ray 结构，包括 Ray 的原点和方向（参考 [1] 的“表面向量与多个物体”）。然后是两个 hit 函数：hitSphere 判断是否击中某个球；hitWorld 用来遍历所有的球，找到 Ray 相交最近的球。

在片段着色器中，定义了 Sphere 结构和 hitRecord 结构，其中 hitRecord 用于记录碰撞的位置以及碰撞点的法向量。

然后我们在 main 函数中将四个球的位置和半径传入到片段着色器中。

本节代码输出的结果是：



现在还缺少什么？阴影。在 [1] 中，阴影产生于 Ray 多次碰撞反弹消磨的能量，我们暂时也不定义光源产生的阴影，而是通过漫反射反弹来得到阴影。

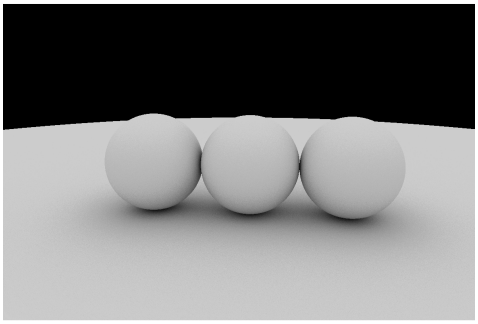
2.4 光线反弹

本节代码见 [chapter-2/Source4]。

实现光线追踪有两种思路，一是光线的递归方法（比如 [1]），二是循环方法（比如 PBRT 等较高级渲染器）。由于 GLSL 不支持递归，也不支持循环调用（A 调用 B，B 调用 A），因此，我们需要使用循环的方法。其实循环比递归更容易控制中间流程。

本节我们实现了 random_in_unit_sphere 函数和 shading 函数。第一个函数在于产生单位球内的方向，第二个函数在于通过循环来计算着色。光线反弹的方向就是交点处法向量加上单位球内的方向。

得到的结果为：



2.5 金属材质

本节代码见 [chapter-2/Source5]。

我们在 Sphere 类中增加一个变量 materialIndex 用来表示当前的球属于哪种材质；以及一个 albedo 变量，表示表面反照率。

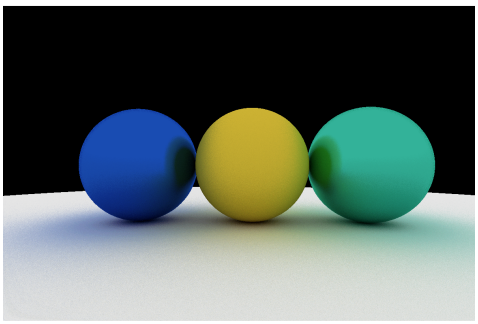
在 hitRecord 结构中定义 materialIndex 和 albedo，在 hitWorld 函数里，当检测到有碰撞时，就记录下来碰撞位置的材质索引和 albedo。

然后我们定义了两种反射类型：漫反射和金属反射。

```
vec3 diffuseReflection(vec3 Normal) {
  return normalize(Normal + random_in_unit_sphere());
}
vec3 metalReflection(vec3 rayIn, vec3 Normal) {
  return normalize(rayIn - 2 * dot(rayIn, Normal) * Normal + 0.35*
     random_in_unit_sphere());
}
```

在 shading 函数里判断 rec.materialIndex 来决定使用哪个反射函数。

本节代码运行得到的结果是：



2.6 本章小结

由于这个程序结构还不完善，为了防止结构变得混乱，我们就暂时不考虑实现玻璃材质了。

本章我们已经构建了一个非常基础的光线追踪器，并且能够产生还算可以的图像效果——最重要的是，这些内容都是在 GLSL 上实现的，比在 CPU 上实现要更有趣。

但是现在很多结果都比较零散，比如材质。下一章，我们会考虑对代码再进一步封装，以及实现将大规模的模型数据提前存储到 GPU 中供我们使用。

3. 模型数据存储

本章节将包含大量面片的模型导入到 *GLSL* 环境中，尽管没有加速结构会比较慢，但我们还是以先实现基本功能为主，优化方法放在后面进行。

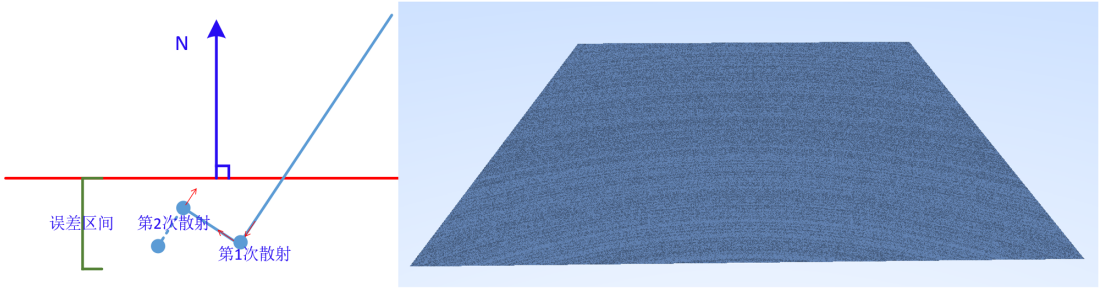
3.1 光线与三角形求交

本节代码见 [chapter-3/Source1]。

网上有很多种关于光线和三角形求交的方法，在 PBRT 中也有一个效率很高且有效控制误差的方法，将 Ray 变换到三角形空间中。我们这里采用的方法是 [9] 里的方式，整个过程分为两步：

- 计算 Ray 与三角形所在平面的交点。
- 判断 Ray 与三角形所在平面的交点是否在三角形内部。

该过程实现在了 hitTriangle 函数里。但是如果按照流程 [9] 实现，由于精度控制的问题，有时候计算出的交点会跑到面片后面，从而与面片多次相交，导致出现下面的这种效果：

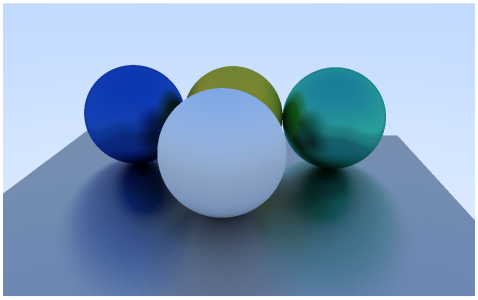


上图中的红线表示三角形平面。求交时，Ray 与平面相交有一个误差区间，在这个区间内，虽然根据法向量都是朝上散射，但距离在误差范围内，就可能来回跳变，一直卡在附近，比如第 2 次散射，计算距离以后甚至还更靠下了（但还是在误差区间内）。于是，在程序中，判定与三角形求交中，我们令 Ray 前进的长度减一个非常小的值，比如 0.00001。

我们定义了 Triangle 类，该类包括了顶点、每个顶点处的法向量以及每个顶点的纹理坐标，但我们暂时先不使用法向量和纹理坐标。

在 hitWorld 中，我们做两次循环，先遍历与 Shpere 求交，再遍历与 Triangle 求交，如果判定与 Triangle 求交，则最近点在 Triangle 上（只有求交距离小于当前最小距离才会判定为求交）。在 hitRecord 中进行记录，我们定义的两个三角形的法向量都是朝上的（作为地板）。

本小节得到的结果显示如下：



3.2 导入大量数据

本节代码见 [chapter-3/Source2]。

当我们的场景包含大量三角面片数据时，就需要一个有效的存储结构，我们可以存储在纹理中。注意，由于 FrameBuffer 一直绑定的都是纹理 0，所以三角面片需要加载到其他纹理编号中。

我们要存储的数据类型是 GL_FLOAT,因为存储的值属于单色纹理,所以需要用到 GL_RED 表示源图格式（单色 float 图），以及数据类型为 GL_R32F：

```
1  glGenTextures(1, &ID);
2      glBindTexture(GL_TEXTURE_2D, ID);
3  // 注意width为dataSize, height为1
4      glTexImage2D(GL_TEXTURE_2D, 0, GL_R32F, dataSize, 1, 0, GL_RED
           , GL_FLOAT, data);
```

一般 Obj 物体都会把顶点放在一起，然后定义每个面的顶点编号。因此我们需要两个纹理，一个用来存顶点与顶点法向量、纹理坐标；另一个用来存三角面片的三个顶点的编号。

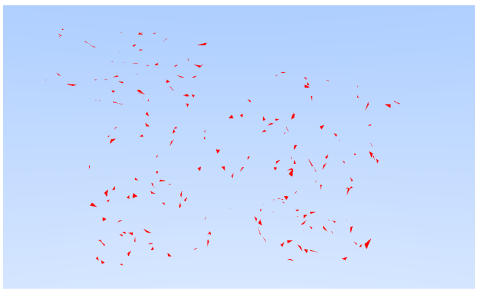
有的模型可能会需要好几张纹理图（比如 [4] 里孤岛危机主角的纳米服），我们暂时不考虑这种情况（实际上，我们使用的 Bunny 和 Dragon 模型都是没有纹理的），因为在具体的游戏或应用中都会统一定义自己的模型文件格式，将其他格式，比如.max 或者.obj 转换到自己的格式，我们就不搞这么麻烦了。

我们定义了 ObjectTexture.h 文件，该文件中的 getTexture 函数用来把模型中的顶点数据生成纹理。现在纹理的分配如下：

```
1  // 纹理0: Framebuffer
2  // 纹理1: MeshVertex
3  // 纹理2: MeshFaceIndex
```

尽管我们使用的是二维纹理,本节我们只考虑将数据生成一维,由于纹理最大维度仅有 2048*2080,所以我们只能往里面存几千个面片（大家可以尝试一下，如果存储的浮点数过多，那么 GLSL 并不会报错，但纹理生成会失败，读取的数据都是 0.0）。在着色器程序里，我们击中三角形就显示红色，否则显示背景色。

本节代码的渲染结果如下：



如果大家根据第一章的第一节模型加载部分显示一下中国龙的话，就能看到现在显示的大致轮廓还是符合中国龙的特征的。

我们下一节实现二维坐标访问，通过定义二维坐标，就能把整个龙的数据都加载进去了。

3.3 数据存储为二维

本节代码见 [chapter-3/Source3]。

修改 getTexture 函数来生成二维纹理。我也不知道纹理长宽最大可以是多少维，为了简单，我们用平方根来确定长宽：

```
1  float vert_x_f = sqrtf(dataSize_v);
2  int vert_x = ceilf(vert_x_f);
3  int vert_y = ceilf((float)dataSize_v / (float)vert_x);
4
5  float face_x_f = sqrtf(dataSize_f);
6  int face_x = ceilf(face_x_f);
7  int face_y = ceilf((float)dataSize_f / (float)face_x);
```

注意以前的 OpenGL 版本要求纹理长宽需要是 2 的整数次幂，比如 1024*1024。现在新版本已经没有这个要求了。

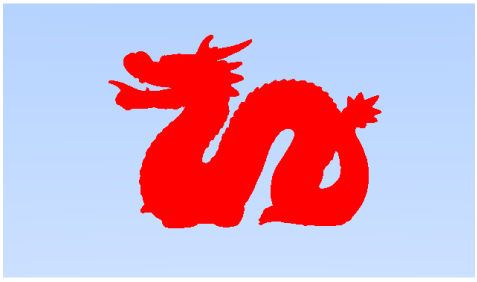
由于我们现在还没有变换矩阵，我们一开始就将中国龙的坐标设置的小一点，并整体下移 0.2 个单位：

```
1  vertexArray[v_index * (8) + 0] = 0.04 * data[i].vertices[j].Position
       .x;
2  vertexArray[v_index * (8) + 1] = 0.04 * data[i].vertices[j].Position
       .y - 0.2;
3  vertexArray[v_index * (8) + 2] = 0.04 * data[i].vertices[j].Position
       .z;
```

在 GLSL 里访问二维纹理的方式如下：

```
float At(sampler2D dataTex, float index) {
  float row = (index + 0.5) / textureSize(dataTex, 0).x;
  float y = (int(row) + 0.5) / textureSize(dataTex, 0).y;
  float x = (index + 0.5 - int(row) * textureSize(dataTex, 0).x) /
     textureSize(dataTex, 0).x;
  vec2 texCoord = vec2(x, y);
  return texture2D(dataTex, texCoord).x;
}
```

本节代码的渲染结果如下：



3.4 法向量计算

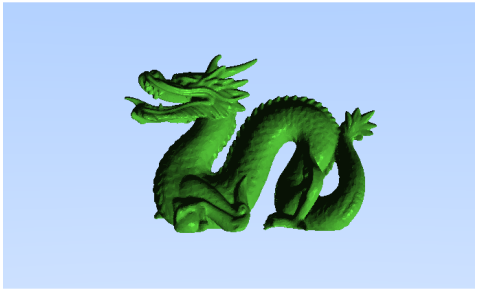
本节代码见 [chapter-3/Source4]。

由于现在运行一帧耗费的时间太长，我们也很难指望渲染出什么好看的场景。但至少我们可以先尝试计算一下法向量，因为我们的 Obj 模型并没有提供法向量，我们给每个顶点留出的 8 个浮点空位中只有顶点位置有用，其他相当于预留的功能。

法向量可以通过叉积来完成，注意三角形的三个顶点 v0，v1 和 v2 从正面看的话，v0-v1-v2 的顺序是逆时针（这也是一般的模型中约定俗成的），由此可以得到计算法向量的方法（在 GLSL 中计算）：

```
vec3 getTriangleNormal(Triangle tri) {
        return normalize(cross(tri.v2 - tri.v0, tri.v1 - tri.v0));
}
```

本节代码的渲染结果如下：



对于有纹理的模型，纹理插值还需要计算重心坐标（相当于 Ray 与面片的交点距离三个顶点的距离比值）。由于暂时我们的模型是没有纹理的，所以中心坐标就先不进行计算了。

3.5 本章小结

本章我们将包含大量数据的模型导入到了纹理中，我们也感受到了没有使用加速结构渲染的速度之慢。

下一章开始，我们会定义和使用 BVH 树，构建一个实时的光线追踪器。

4. 加速结构

本章节用一整个章节来实现和优化加速结构——*BVH(Bounding volume hierarchies)* 树。

4.1 构建思路

简单的构建 BVH 树的方法可以参考 [2] 的内容。但是放到 GLSL 中就没有那么简单了，比如，三角面片要存储在纹理里面，以及 BVH 数要“拍平”为一个数组 [8]。而且，调试也会变得非常困难。但所幸，在 GPU 能调试的东西，在 CPU 也可以模拟出来，所以我们先在 CPU 中模拟整个流程，确保无误以后，再转移到 GPU 中。

在多个复杂模型的场景中，一个可选的过程是首先对每个模型来构建树，然后把多个模型的树再进行合并，这样可以更好地降低复杂性。不过我们构建 BVH 树的方法在使用中效果很好，所以也没有必要人为控制过多因素。

因为构建和调试 BVH 树的过程比较复杂，所以我们将先在 CPU 中构建测试，然后再转移到 GPU 中。

4.2 构建 BVH 树

本节代码见 [chapter-4/Source1]。

建树的方式我们参考的是 PBRT[11] 的 EqualCounts 方法，如果有兴趣可以参考《PBRT 系列 3-代码实战-形状和加速器》，如果只是想进行移植，那么就不需要花太多精力了解本节的内容。

4.2.1 封装结构

我们目前只考虑给三角面片构建 BVH 树，这是 GLSL 里不支持多态，所以不能把所有基元都使用统一的接口。一个可选方案是，在 BVH 的 Node 叶节点中做标记，比如如果是球就标记 0，如果是三角就标记 1 等，然后记录在球数组和三角数组中的索引。但由于这会增加复杂性，所以我们一开始并不考虑。

我们本节新建了两个头文件：shape.h 和 BVHTree.h

shape.h 中定义了包围盒类 Bound3f、三角形类（从以前的 ObjectTexture.h 头文件中移到了 shape.h 中）以及在 CPU 的 Ray 类。还有一些关于求交的程序，以及定义一个函数 getTriangleBound，用来返回三角形的包围盒。

4.2.2 基本类的定义

首先定义 BVH 树的节点 BVHNode 类。该类要么会作为内部节点，要么会作为叶节点，因此有两个初始化函数。由于可能一个叶节点可能包含多个基元（三角面片、球体都是基元），比如有时候好几个基元的包围盒质心都会在同一个点上，所以有 firstPrimOffset 和 nPrimitives 这两个变量。以及记录划分两个子包围盒的轴 splitAxis。

然后定义“拍平”以后的树的节点 LinearBVHNode。注意，用数组表示树时，由于左子节点恰好在当前节点后面，所以不用记录左子节点的位置，只需要记录右子节点的位置即可。

然后是基元对应于 BVH 构建的信息：BVHPrimitiveInfo。该类包含有基元的索引 primitiveNumber 以及基元的包围盒 bound 和包围盒的重心坐标 centroid。

4.2.3 递归建树和“拍平”操作

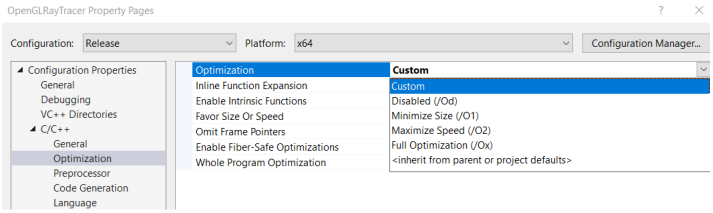
递归建树的过程其实并不难，由于 PBRT 已经写得很详细了，这里不再赘述。这两个函数分别是 recursiveBuild 和 flattenBVHTree。调用 flattenBVH 就能构建一个数组。

IntersectBVH 是与 BVH 求交的程序，也不再赘述。

为了证明我们构建的 BVH 树是有效的，我们实现了一个 BVHTest 程序，它用于产生相机 Ray，然后与包围盒求交，只要与三角面片相交，就画红色：

由于 CPU 渲染比较慢，为了更好地把握进度，程序会打印出当前正在渲染的像素。Visual Studio 中使用 Debug 模式比 Release 模式要慢得多。

有一点可能需要注意，我调试时总是内存报错，研究了很久都没有办法，后来突然想到或许是编译或者运行优化的缘故，关闭了 Visual studio 的优化以后就好了：



4.3 转到 GLSL 风格

本节代码见 [chapter-4/Source2]。

我们现在需要把符合 C++ 风格的树结构和遍历操作修改为 GLSL 风格，当前的操作还是在 CPU 上进行，但 GLSL 风格不能再使用指针等操作了。

我们把 LinearBVHNode 中的 Bound3f 写成下面的形式（int 类型都表示为了浮点数）：

```
struct LinearBVHNode {
        vec3f pMin, pMax;
        float nPrimitives;
        float axis;
        float childOffset; //第二个子节点位置索引 或 基元起始位置索引
};
```

我们还定义了下面的两个变量和数组：

```
int nodeNum; // 节点数
float *NodeArray; // 存节点的数组
int meshNum; // Mesh数
float *MeshArray; // 存Mesh的数组
```

这些数组都是用于导入到纹理中的。NodeArray 的大小为 9 乘以 nodeNum 个浮点数（pMin 和 pMax 分别占 3 个浮点数，以及剩下三个 float 类型的数组）。

在 IntersectBVH 函数中改动较多，大家可以参考源码。

虽然我们还保留了一些类和类访问，但已经和 GLSL 非常接近了，现在我们可以很容易地将代码迁移到 GLSL 上。

4.4 转移到 GLSL 上

本节代码见 [chapter-4/Source3]。

现在有一个好处：我们不再需要面片索引了。因为现在的三角形数组就是按照一个个的面片来排列的，这样虽然比较占用空间，但使用起来方便多了。

关于我们是否需要给数组预留一个法向量和纹理，我认为还是留着吧，虽然挺浪费空间的（相当于每个三角形占用的空间为 3*3 个 float 顶点坐标，3*3 个 float 法向量坐标，3*2 个纹理坐标，共 24 个 float 值），但留着或许有用呢（虽然在本文中没有什么用）。

我们在 BVHBuildTree 函数中计算纹理的长宽，这样为了申请足够大的数组（大小等于 meshNumX*meshNumY），输入给二维纹理数组如果不够大，就会报内存访问错误。

实现在 GLSL 中的部分和 CPU 上差不多，因此不再赘述。本节的代码实现目标是在 GPU 中访问通过加速结构来访问三角形，与三角面片相交就显示为红色。本节是我花费调试时间最多的一章，通宵了一整天才调试成功，这是因为有些微小的 Bug 很难察觉，但会直接导致结果错误，在 GLSL 里也是很难跟踪。

本节代码的显示效果和第三章第三节的效果一样，所以就不再展示了。

4.5 应用加速结构的渲染

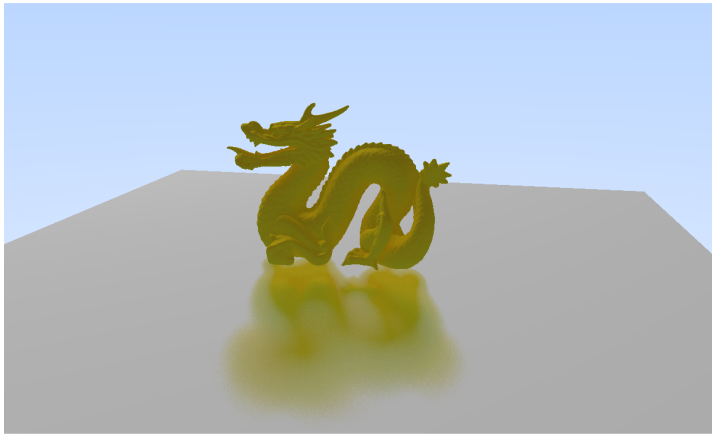
本节代码见 [chapter-4/Source4]。

我们在 GLSL 的 Ray 结构中加上一个 hitMin 变量，hitMin 表示当前 ray 就能撞上物体前进最近的距离（撞上当前已经测试求交过的物体）。

GLSL 里的 hitWorld 函数和 IntersectBVH 函数都进行了相关的修改。因为目前我们还没有光源采样，所以我们仍然按照 [1] 的方法，碰撞的次数越多就越暗。同时，如果第一次与物体相交，但反射的时候没有与物体相交，则做一个简单的 diffuse 光照（见 shading 函数）：

```
1  if (i == 1) {
2    vec3 lightPos = vec3(-4.0, 4.0, -4.0);
3    vec3 lightDir = normalize(lightPos - rec.Pos);
4    float diff = 0.5 * max(dot(rec.Normal, lightDir), 0.0) + 0.5;
5    color *= vec3(diff, diff, diff);
6  }
```

本节代码显示效果如下（源码效果为左图，稍加修改就能得到右图）：



在我的 1080Ti GPU 上测试，大概每秒钟 60 到 70 帧左右。

4.6 本章小结

事实上，本来本文我总共规划了七八章，包括材质和物体的管理和优化、面光源与重要性采样、甚至包括物理材质渲染的内容，但我发现，写着写着就越来越像流水账了，因此就此打住——从某种意义上来说，本文也该结束了。

虽然我们已经实现了一个看起来还凑合的光线追踪器，但仍然留下了大量需要改进的地方，比如我们将模型、地板和球分为了三部分，但其实它们都应该被统一使用包围盒加速。想要改进也不难——先在 CPU 中定义一个基元类，设定一个变量用于记录当前物体的类型（是球体还是三角形），然后生成包围盒；同时 GPU 中也需要记录基元类型，比如这样：

```
1  struct PrimitiveType {
2          int type;
```

```
3         int materialIndex;
4 };
```

材质的管理也是一个富有挑战的任务，比如制作一个材质列表纹理，然后物体基元记录使用的哪种材质，以及 albedo 是什么值等。

虽然后续任务也会比较艰辛，但我们目前已经拥有了一切基础，最困难的部分——BVH 树也已经调试成功，接下来的优化就会相对容易很多了。

参考文献

[1] Shirley P. Ray Tracing in One Weekend[J]. 2016.

[2] Shirley P. Ray Tracing The Next Week[J]. 2016.

[3] Shirley P. Ray Tracing The Rest Of Your Life[J]. 2016.

[4] https://learnopengl-cn.github.io/

[5] https://zhuanlan.zhihu.com/p/51387524

[6] http://holger.dammertz.org/stuff/notes_HammersleyOnHemisphere.html#warren01

[7] https://zhuanlan.zhihu.com/p/326270572

[8] https://zhuanlan.zhihu.com/p/364043906

[9] https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/

[10] https://www.cs.uregina.ca/Links/class-info/315/WWW/Lab5/InternalFormats_OGL_Core_3_2.html

[11] Pharr M, Jakob W, Humphreys G. Physically based rendering: From theory to implementation[M]. Morgan Kaufmann, 2016.