TJRenderer Documentation

TJRenderer Documentation

- 1. 简介
  - 1.1 功能介绍
  - 1.2 运行环境
  - 1.3 运行说明
  - 1.4 操作说明
  - 1.5 运行示例
- 2. TGA图片与窗口绘制
  - 2.1 TGA格式图片介绍
  - 2.2 为什么选择TGA格式图片
  - 2.3 窗口绘制
- 3. Bresenham画线算法
  - 3.1 一般化
  - 3.2 优化
  - 3.3 性能对比
  - 3.4 线框渲染
- 4. 三角形栅格化和背面剔除
  - 4.1 扫描线实区域填充算法
  - 4.2 重心坐标插值算法
  - 4.3 背面剔除
- 5. 坐标系变换
  - 5.1 流水线中的坐标系变换
  - 5.2 相机矩阵的推导
- 6. 一些处理
  - 6.1 透视插值校正
  - 6.2 zbuffer
- 7. 着色器 Shader
  - 7.1 纹理贴图 Texture Mapping
    - 7.7.1 漫反射贴图 Diffuse Mapping
    - 7.7.2 法线贴图 Normal Mapping
    - 7.7.3 高光贴图 Specular Mapping
  - 7.2 纹理着色器 General Shader
  - 7.3 深度着色器 Depth Shader
  - 7.4 阴影着色器 Shadow Shader
  - 7.5 环境光遮蔽 Ambient Occlusion
- 8. 参考资料

1. 简介

1.1 功能介绍

- 画线算法

- GUI绘制
- 三角形栅格化
- 顶点变换
- 视角变换
- 透视插值
- 纹理贴图
- 着色器（顶点着色器、片段着色器）
- 阴影贴图
- 环境光遮蔽

1.2 运行环境

- Microsoft Windows 32位
- Mac OS X

1.3 运行说明

- 本项目使用 CMake 工具以支持跨平台编译，具体编译方式如下：
  - DEBUG 模式：

    ```
    $ mkdir Debug
    $ cd Debug
    $ cmake -DCMAKE_BUILD_TYPE=Debug ..
    $ make
    ```

  - RELEASE 模式：

    ```
    $ mkdir Release
    $ cd Release
    $ cmake -DCMAKE_BUILD_TYPE=Release ..
    $ make
    ```

- 如果没有 CMake 工具，Windows 平台下的 Visual Studio 可以选择导入 CMake 项目。
- 运行编译后的程序时请附加运行参数，参数为渲染所用 OBJ 文件地址；如有多个文件，则按序输入其地址，用空格隔开（文件的顺序即渲染器渲染的顺序）
  - Mac OS X terminal：

    ```
    ./Viewer ../obj/boggie/body.obj ../obj/boggie/head.obj
    ../obj/boggie/eyes.obj
    ```

  - Windows terminal:

    ```
    ./Viewer.exe ../obj/boggie/body.obj ../obj/boggie/head.obj
    ../obj/boggie/eyes.obj
    ```

  - IDE 配置（如CLion、Visual Studio等）

在项目的配置信息中，程序参数 (Program Parameters) 一栏中添加上述参数：

```
../obj/boggie/body.obj ../obj/boggie/head.obj ../obj/boggie/eyes.obj
```

1.4 操作说明

按键	说明
`A`	摄像机以模型的中心为圆心向左转动
`D`	摄像机以模型的中心为圆心向右转动
`W`	摄像机以模型的中心为圆心向上转动，为了避免视角的颠倒，移动一定角度后就不能再移动了
`S`	摄像机以模型的中心为圆心向下转动，为了避免视角的颠倒，移动一定角度后就不能再移动了
`1`	切换至常规渲染模式
`2`	切换至深度着色器渲染模式
`3`	切换至阴影着色器渲染模式
`4`	切换至环境光遮蔽着色器渲染模式
`5`	切换至线框渲染模式

1.5 运行示例

Fig 1. 常规渲染器模式

Fig 2. 深度着色器渲染模式

Fig 3. 阴影着色器渲染模式

Fig 4. 环境光遮蔽着色器渲染模式

2. TGA图片与窗口绘制

2.1 TGA格式图片介绍

TGA是由美国Truevision公司为其显示卡开发的一种图像文件格式，已被国际上的图形、图像工业所接受。现已成为数字化图像，以及运用光线跟踪算法所产生的高质量图像的常用格式。TGA文件的扩展名为.tga，该格式支持压缩，使用不失真的压缩算法，可以带通道图，另外还支持行程编码压缩。关于TGA文件的详细叙述可以在truevision的技术指南附录中找到。

2.2 为什么选择TGA格式图片

我们在渲染器读取材质图片和生成渲染图片时分别读取和生成了TGA图片，其原因如下所述：

- 编码方式简单。由于我们的目标是尽可能地减少外部库（如 OpenGL, OpenCV 等）的依赖，因此我们对于图像格式，尽可能选择那些编码方式简单的图片格式；
- 支持的图像格式多。TGA格式通过文件头的一个字段来表明它对应的文件类型，我们使用了它常用

的几种类型，即RGB/RGBA/黑白格式；
- 读取和写入图像相对容易；在网上可以找到很多关于写入和读取TGA图片的源码，因此我们不用过多地关心文件格式的问题，只需考虑计算出读取图片和写入图片时每个像素点的像素值。

2.3 窗口绘制

我们窗口绘制的代码主要参考了[Software Renderer - A shader-based software renderer written from scratch in C89]的方法，使用系统API实现了简单的图形界面，只提供了打开窗口、显示渲染画面、调整渲染视角和渲染模式这些功能。

由于我们小组对任务进行了分工，因此窗口的显示是个单独的模块，为了降低与渲染器其它部分的耦合性，我们的窗口是将渲染后的一帧输出到窗口中，当视角移动时再在循环体里重新计算这一帧，因此卡顿现象比较明显，远达不到实时渲染的效果。

3. Bresenham画线算法

这部分的算法主要参考了[维基百科 - 布雷森汉姆直线算法]和课件上的资料。后面由于浮点运算的代价较高，故将浮点运算改用整数运算进行了优化，并测试、比较了二者的性能。

3.1 一般化

对于由 (x_0, y_0) 和 (x_1, y_1) 两点所组成的直线，直线的表达式如下：

$$y - y_0 = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

由于 x 和 y 都是整数，因此没有必要去计算每一点 x 所对应之 y 值。反之由于此线之斜率介乎于 1 至 0 之间，故此我们只需要找出当 x 到达那一个数值时，会使 y 上升 1，若 x 尚未到此值，则 y 不变。至于如何找出相关的 x值，则需依靠斜率。斜率之计算方法为

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

由于此值不变，故可于运算前预先计算，减少运算次数。

要实行此算法，我们需计算每一像素点与该线之间的误差。对于斜率小于或等于 1 的直线，误差应为每一点 x 中，其相对的像素点之 y 值与该线实际之 y 值的差距。每当 x 的值增加 1，误差的值就会增加 m。每当误差的值超出 0.5，线就会比较靠近下一个映像点，因此 y 的值便会加 1，且误差减 1。

当斜率大于 1 时，则只需在整个计算范围中交换 x 和 y。当斜率为负时，则将 `ystep` 置为 -1。

相应的代码如下：

```
void line(int x0, int y0, int x1, int y1, Image &image) {
    bool steep = abs(y1 - y0) > abs(x1 - x0);
    if (steep) {
        std::swap(x0, y0);
        std::swap(x1, y1);
    }
    if (x0 > x1) {
        std::swap(x0, x1);
```

```
        std::swap(y0, y1);
    }
    int deltax = x1 - x0;
    int deltay = abs(y1 - y0);
    float error = .0;
    float deltaerr = deltay/float(deltax);
    int ystep;
    int y = y0;
    ystep = y0 < y1 ? 1 : -1;
    for (int x = x0; x < x1; ++x) {
        steep ? image.plot(y, x) : image.plot(x, y);
        error += deltaerr;
        if (error >= .5) {
            y += ystep;
            error -= 1.;
        }
    }
}
```

3.2 优化

由于计算机处理浮点数的速度比较慢，而上述代码中 `error` 和 `deltaerr` 的计算都是浮点数的运算。此外，`error` 的值经过多次浮点数加法之后，可能有累积误差。使用整数运算可令算法更快、更准确。只要将所有以上的分数数值乘以 `deltax`，我们就可以用整数来表示它们。唯一的问题是程序中的常数 0.5 —我们可以通过改变 `error` 初始值的方法，以及将 `error` 的计算由递增改为递减来解决。新的程序如下：

```
void line(int x0, int y0, int x1, int y1, Image &image) {
    bool steep = abs(y1 - y0) > abs(x1 - x0);
    if (steep) {
        std::swap(x0, y0);
        std::swap(x1, y1);
    }
    if (x0 > x1) {
        std::swap(x0, x1);
        std::swap(y0, y1);
    }
    int deltax = x1 - x0;
    int deltay = abs(y1 - y0);
    int error = deltax / 2;
    int ystep;
    int y = y0;
    ystep = y0 < y1 ? 1 : -1;
    for (int x = x0; x < x1; ++x) {
        steep ? image.plot(y, x) : image.plot(x, y);
        error -= deltay;
        if (error < 0) {
            y += ystep;
```

```
            error += deltax;
        }
    }
}
```

3.3 性能对比

我们在程序中画了三百万根直线，并测试了以上两种画线算法所消耗的 CPU 时间，我们用 `g++ -ggdb -g -pg -O0` 命令追踪了 `line()` 函数调用执行的总时间：

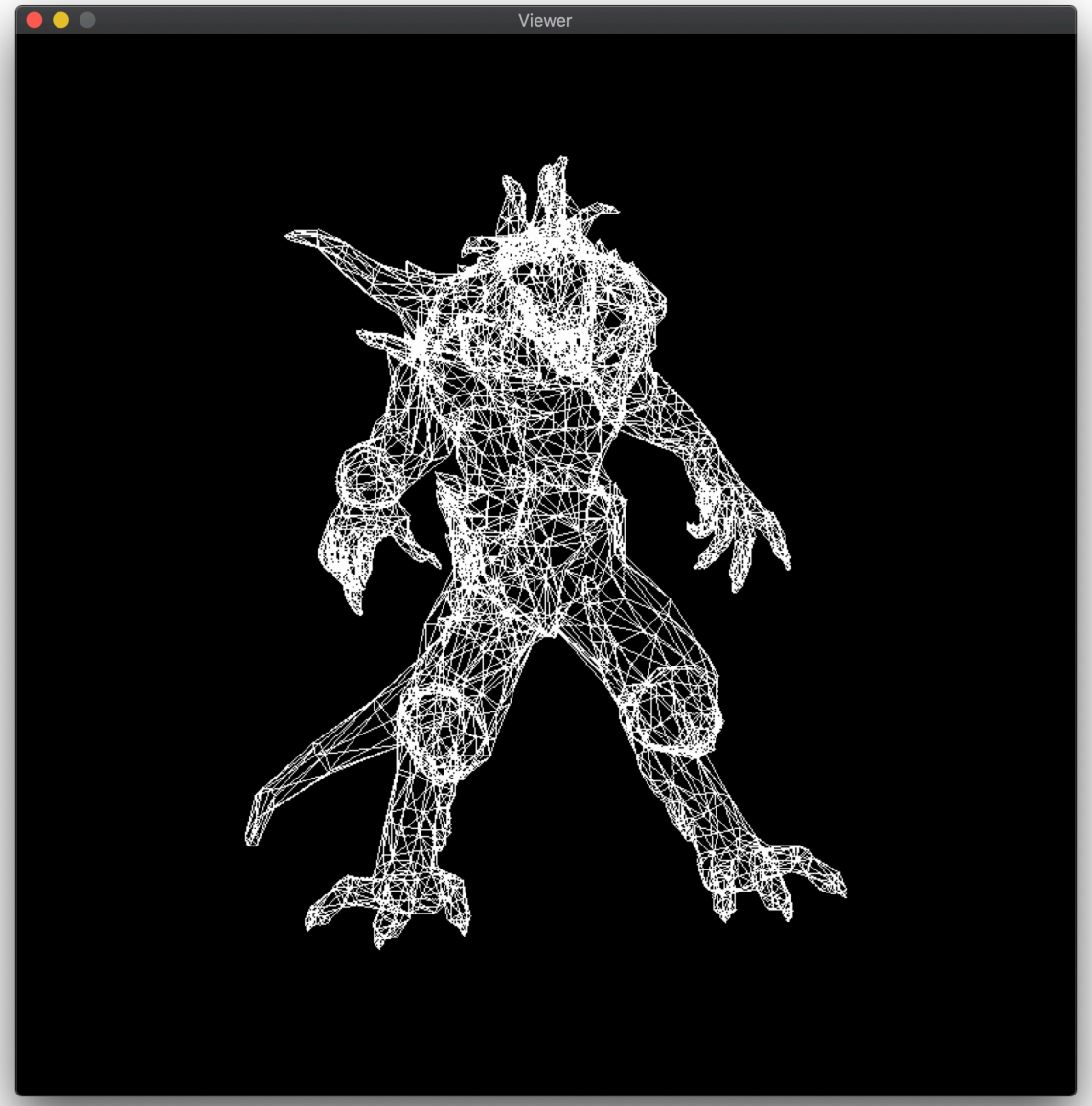
算法	line() 执行时间	line() 调用时间占总时间比例
浮点数运算版本	0.47s	59.65%
整数运算版本	0.41s	53.27%

可以看出优化后的版本稍有提升，即使在关闭所有优化 (-O0) 时，二者的差距也并不大，可见浮点数运算在画线算法中并非性能瓶颈，在开启 -O3 编译优化之后，这些差距就基本被抹除了。

3.4 线框渲染

完成上述画线算法后，我们就可以创建一个线框渲染器了。我们使用 wavefront obj 格式文件来保存模型，该格式文件使用 `v x y z` 来表示一个几何顶点，用顶点索引 `f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3` 等方式描述表面单元，此处不做详细介绍。

只需在程序中读取每一个模型文件，对于每一个模型文件再读取其中的每一个三角面，将它的三个顶点全部投影到 xy 坐标平面上（只取其三维坐标中的 x 和 y 值），将三个投影后的坐标使用上面的画线算法即可绘制出 xy 坐标平面上的线框图。



4. 三角形栅格化和背面剔除

4.1 扫描线实区域填充算法

对于三角形的扫描线实区域填充算法，其原理与课上讲的多边形扫描转换算法（新边表与活性边表）有些相似，只是需要考虑的情况相比多边形的填充而言更少，实现起来也相对而言更加简单。对于普通的三角形而言，只需考虑将三个顶点的按其大小顺序进行排序，然后根据中间的那个顶点将三角形分为上下两部分（对于三角形有一边与水平线平行的情况，则可以认为该三角形只含上半部分或者下半部分）。随后只需求出水平扫描线与两个三角形之间的交点，在两个交点之间填充像素即可。

由于这部分原理简单，因此略去实现的代码和结果。因为我们项目中实际使用的是后面讲到的算法。

4.2 重心坐标插值算法

首先可以在维基百科 - 重心坐标页面查看重心坐标的定义以及三角形重心坐标的求解方法。

给定一个二维的三角形 ABC 和一个点 P，这四个点的坐标都由笛卡尔坐标系给出。我们的目标是找到点 P 关于三角形 ABC 的重心坐标。这意味着我们寻找三个值 $(\lambda_1, \lambda_2, \lambda_3)$，使得 P 的坐标可由下式表示：

$$P = \lambda_1 A + \lambda_2 B + \lambda_3 C$$

易证得 $\lambda_1 + \lambda_2 + \lambda_3 = 1$，证明过程比较容易，通常可以使用面积法进行证明。假设我们将三个权重 $(1-u-v, u, v)$ 分别赋值给顶点 A, B 和 C，那么我们就可以说点 P 相对于斜基底 (oblique basis) $(A, \overrightarrow{AB}, \overrightarrow{AC})$ 有坐标 (u, v)：

$$P = A + u\overrightarrow{AB} + v\overrightarrow{AC}$$

所以，我们有向量 $\overrightarrow{AB}, \overrightarrow{AC}$ 和 $\overrightarrow{AP}$，我们需要找到两个实数 u, v 满足以下约束：

$$u\overrightarrow{AB} + v\overrightarrow{AC} + \overrightarrow{PA} = \overrightarrow{0}$$

这是一个简单的向量等式，或者我们可以将它写成一个线性方程组：

$$\begin{cases} u\overrightarrow{AB}_x + v\overrightarrow{AC}_x + \overrightarrow{PA}_x = 0 \\ u\overrightarrow{AB}_y + v\overrightarrow{AC}_y + \overrightarrow{PA}_y = 0 \end{cases}$$

可以再将这个线性方程组改写为矩阵的形式：

$$\begin{cases} \begin{bmatrix} u & v & 1 \end{bmatrix} \begin{bmatrix} \overrightarrow{AB}_x \\ \overrightarrow{AC}_x \\ \overrightarrow{PA}_x \end{bmatrix} = 0 \\ \begin{bmatrix} u & v & 1 \end{bmatrix} \begin{bmatrix} \overrightarrow{AB}_y \\ \overrightarrow{AC}_y \\ \overrightarrow{PA}_y \end{bmatrix} = 0 \end{cases}$$

这意味着我们只需寻找一个同时与 $(\overrightarrow{AB}_x, \overrightarrow{AC}_x, \overrightarrow{PA}_x)$ 和 $(\overrightarrow{AB}_y, \overrightarrow{AC}_y, \overrightarrow{PA}_y)$ 正交的向量 $(u, v, 1)$，那么也就是求这两个向量的叉积。因此重心坐标插值算法的核心代码就是求该叉积：

```
Vec3 barycentric(Vec2 *pts, Vec2 P) {
    Vec3 u = cross(Vec3(pts[2][0] - pts[0][0], pts[1][0] - pts[0][0], pts[0][0] - P[0]),
                   Vec3(pts[2][1] - pts[0][1], pts[1][1] - pts[0][1], pts[0][1] - P[1]));
    // 对应u[2]=0的情况，也即三角形退化的情况
    if (std::abs(u[2]) < 1) {
        return Vec3(-1, 1, 1);
    }
    return Vec3(1.f - (u.x + u.y) / u.z, u.y / u.z, u.x / u.z);
}
```

因此算法的步骤如下：

1. 对于平面上的三角形，求出它的矩形包围盒的边界；
2. 对于包围盒中的每一个像素点 P，调用上述 `barycentric()` 函数，计算叉积；
3. 若函数返回的向量中有任意一值小于 0，则表明点 P 不在三角形内部，反之则画点 P；

4.3 背面剔除

目前主流的实现背面剔除的算法都是通过丢弃所有三角形表面法线与相机到三角形矢量的点积大于或等于 0 的三角形。这种情况下，三角形面的朝向与光照方向相同，这意味着光线从三角形的背部照射过去，因此属于不可见部分。

而三角形表面法线矢量则可以通过其两边向量的叉乘求得。这就要求在进行多边形建模时保证场景良好，该算法依赖于三角面顶点的顺序，像 OpenGL 等库往往会根据顶点的绕序来指定哪个方向为正面。

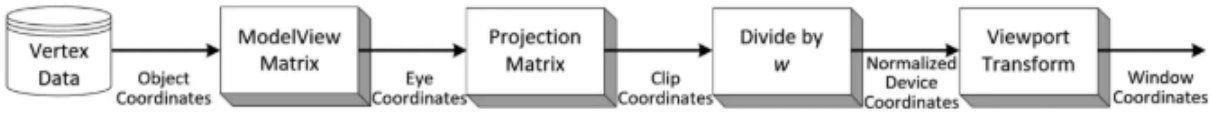
下图展示了逆时针绕序和顺时针绕序对模型进行背面剔除得到的图像：

逆时针绕序（正面）

顺时针绕序（背面）

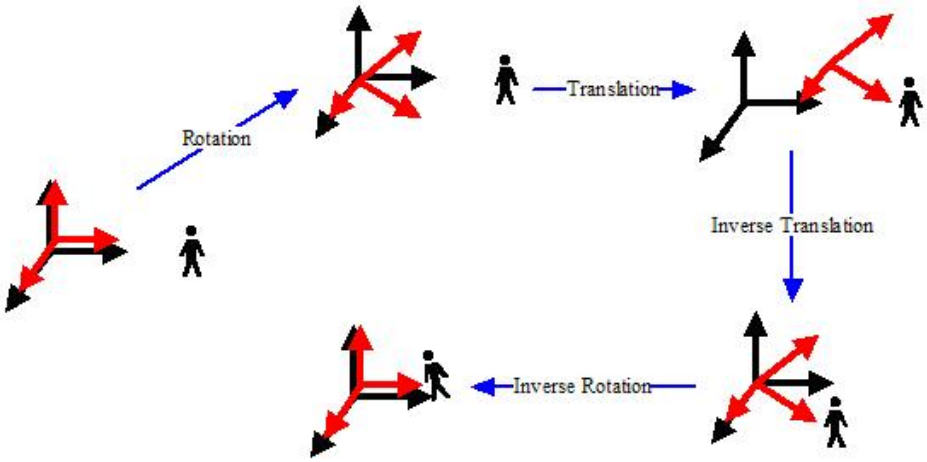
5. 坐标系变换

5.1 流水线中的坐标系变换



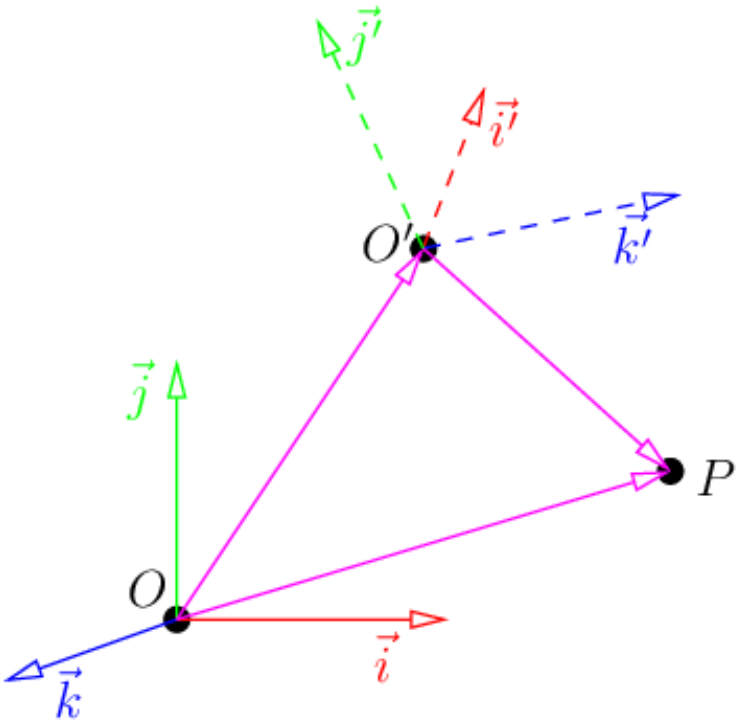
如图所示，当导入模型时，拥有的坐标系为模型坐标系，我们需要将其变换为统一的世界坐标系。因为我们可以用不同的视角进行观察，在不同的视角下，一个在世界坐标系中不动的物体在相机坐标系中是变化的，因此需要通过相机矩阵变换求得其在相机坐标系下的坐标。物体经过投影矩阵后进入裁剪坐标系，这里对其在可视区域进行一定的剪裁，再经过透视之后进入标准设备坐标系，最后通过平移和缩放来得到在屏幕上所显示的坐标。

5.2 相机矩阵的推导



如图所示，从直觉上可以看出，当变换后的相机坐标系同物体一起逆变换到世界坐标系后，即可得到物体在变换后的相机坐标系中的坐标。

下面我们来进行简单证明



如图，我们认为O是世界坐标系的原点，O'是相机坐标系的原点，P为物体

那么一定存在变换C使得

$$[i', j', k'] = [i, j, k] \times C$$

对于$\overrightarrow{OP}$，我们有

$$\overrightarrow{OP} = \vec{i}x + \vec{j}y + \vec{k}z = \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

由向量加法，得

$$\overrightarrow{OP} = \overrightarrow{OO'} + \overrightarrow{O'P} = \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} \begin{bmatrix} O'_x \\ O'_y \\ O'_z \end{bmatrix} + \begin{bmatrix} \vec{i}' & \vec{j}' & \vec{k}' \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

替换，得

$$\overrightarrow{OP} = \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} \left(\begin{bmatrix} O'_x \\ O'_y \\ O'_z \end{bmatrix} + C \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \right)$$

因此，我们有

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} O'_x \\ O'_y \\ O'_z \end{bmatrix} + C \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = C^{-1} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} - \begin{bmatrix} O'_x \\ O'_y \\ O'_z \end{bmatrix} \right)$$

现在只需要求出从世界坐标系变换到相机坐标系的C的逆即可

对于C，我们有

$$C = TR$$

其中T为平移矩阵，R为旋转矩阵

因此C^{-1}为

$$C^{-1} = (TR)^{-1} = R^{-1}T^{-1}$$

又因为R为正交矩阵

所以

$$C^{-1} = R^T T^{-1}$$

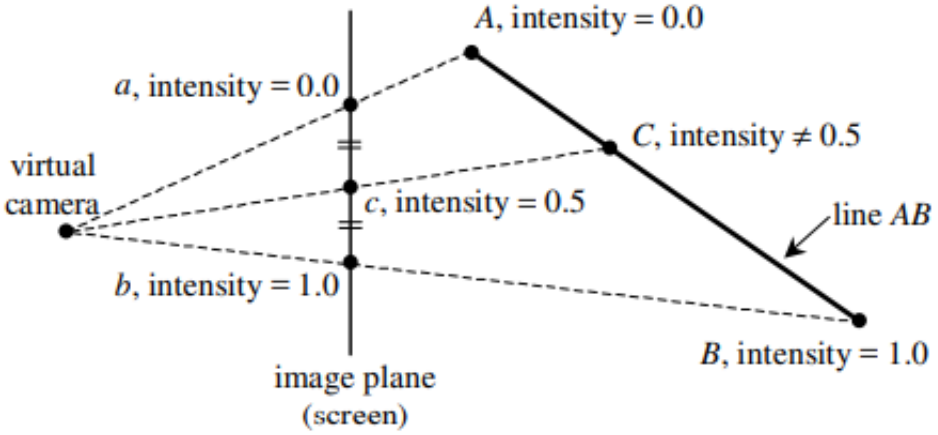
$$C^{-1} = \begin{pmatrix} Ux & Uy & Uz & 0 \\ Vx & Vy & Vz & 0 \\ Nx & Ny & Nz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -Tx \\ 0 & 1 & 0 & -Ty \\ 0 & 0 & 1 & -Tz \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} Ux & Uy & Uz & -U \bullet T \\ Vx & Vy & Vz & -V \bullet T \\ Nx & Ny & Nz & -N \bullet T \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

在本项目中，为了方便实现，我们将摄像机固定在Z轴，通过改变目标位置，来实现变换。

6. 一些处理

6.1 透视插值校正

错误的透视插值

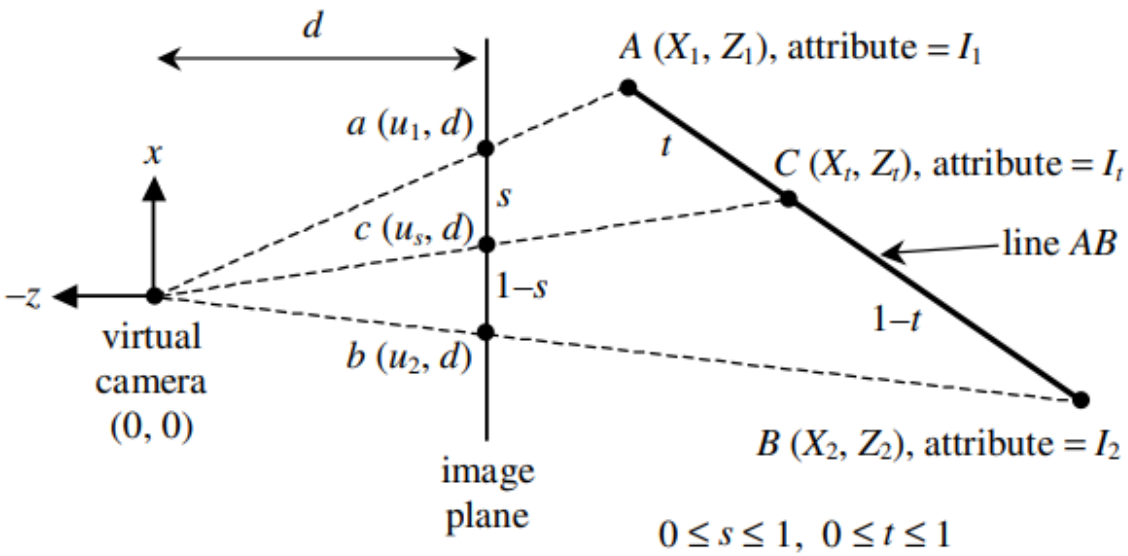


假设c为a，b的中点。如果我们简单对其进行插值，则c的强度为0.5，与初始图元并不一致。

因此，直接进行线性插值得到的结果并不全是正确的。

但是我们可以利用一些函数，来得到正确的结果。

透视插值校正证明



如图，s为在image plane上的插值参数，t为在图元上的插值参数。

根据相似关系，有如下变换

$$\frac{X_1}{Z_1} = \frac{u_1}{d} \Rightarrow X_1 = \frac{u_1 Z_1}{d}$$
$$\frac{X_2}{Z_2} = \frac{u_2}{d} \Rightarrow X_2 = \frac{u_2 Z_2}{d}$$
$$\frac{X_t}{Z_t} = \frac{u_s}{d} \Rightarrow Z_t = \frac{dX_t}{u_s}$$

根据插值关系，有如下等式

$$
\begin{aligned}
u_s &= u_1 + s\left(u_2 - u_1\right) \\
X_t &= X_1 + t\left(X_2 - X_1\right) \\
Z_t &= Z_1 + t\left(Z_2 - Z_1\right)
\end{aligned}
$$

对Z_t进行代入

$$
Z_t = \frac{d\left(X_1 + t\left(X_2 - X_1\right)\right)}{u_1 + s\left(u_2 - u_1\right)}
$$

$$
\begin{aligned}
Z_t &= \frac{d\left(\frac{u_1 Z_1}{d} + t\left(\frac{u_2 Z_2}{d} - \frac{u_1 Z_1}{d}\right)\right)}{u_1 + s\left(u_2 - u_1\right)} \\
&= \frac{u_1 Z_1 + t\left(u_2 Z_2 - u_1 Z_1\right)}{u_1 + s\left(u_2 - u_1\right)}
\end{aligned}
$$

又由于

$$
Z_t = Z_1 + t\left(Z_2 - Z_1\right)
$$

因此有

$$
Z_1 + t\left(Z_2 - Z_1\right) = \frac{u_1 Z_1 + t\left(u_2 Z_2 - u_1 Z_1\right)}{u_1 + s\left(u_2 - u_1\right)}
$$

简化后，得

$$
t = \frac{s Z_1}{s Z_1 + (1 - s) Z_2}
$$

将t代入Z_t方程，有

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

又因为，对于在图元上的属性插值，我们有

$$
I_t = I_1 + t\left(I_2 - I_1\right)
$$

将t代入并化简

$$
I_t = \left(\frac{I_1}{Z_1} + s\left(\frac{I_2}{Z_2} - \frac{I_1}{Z_1}\right)\right) / \left(\frac{1}{Z_1} + s\left(\frac{1}{Z_2} - \frac{1}{Z_1}\right)\right)
$$

利用Z_t方程进行化简

$$
I_t = \left(\frac{I_1}{Z_1} + s\left(\frac{I_2}{Z_2} - \frac{I_1}{Z_1}\right)\right) / \frac{1}{Z_t}
$$

因此，我们得到在投影平面上的属性插值可以由一些变换得到。

透视插值校正代码

对于实际的实现，我们参考了OpenGL SPEC 2.0中的公式

对于一个点p，有

$$p = ap_a + bp_b + cp_c$$

通过下式进行计算

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a/w_a + b/w_b + c/w_c}$$

代码为

```
Vec3f c = barycentric(proj<2>(pts[0] / pts[0][3]), proj<2>(pts[1] / pts[1][3]), proj<2>(pts[2] / pts[2][3]), proj<2>(P));
Vec3f clip = Vec3f(c[0]/pts[0][3],c[1]/pts[1][3],c[2]/pts[2][3]);
clip = clip / (clip[0]+clip[1]+clip[2]);
```

6.2 zbuffer

当我们观察空间任何一个不透明的物体时，只能看到该物体朝向我们的那些表面，其余的表面由于被物体所遮挡我们看不到。

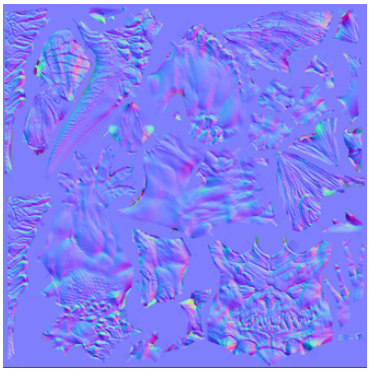
若把可见的和不可见的线都画出来，对视觉会造成多义性。要消除二义性，就必须在绘制时消除被遮挡的不可见的线或面，习惯上称作消除隐藏线和隐藏面，简称为消隐。

zbuffer即为一个较为常用的消隐算法。

```
1. 初始化一个和帧缓存一样大小的存储空间
2. 将zbuffer的各单元初始值置为最小值
3. 当要改变某个像素的颜色值时，首先检查当前多边形的深度值是否大于该像素原来的深度值
4. 如果大于原来的z值，说明当前多边形更靠近观察点，用它的颜色替换像素原来的颜色
```

7. 着色器 Shader

7.1 纹理贴图 Texture Mapping

diffuse	 normal	specular

7.7.1 漫反射贴图 Diffuse Mapping

漫反射贴图表现出物体表面的反射和表面颜色。换句话说，它可以表现出物体被光照射到而显出的颜色和强度。我们通过颜色和明暗来绘制一幅漫反射贴图，在这张贴图中，吸收了比较多的光线的部分比较暗，而表面反射比较强的部分，吸收的光线比较少。

核心代码：

```
float intensity = _intensity_value * bar;
Vecf2 uv = _uv_value * bar;
color = model->diffuse(uv)*intensity;
```

7.7.2 法线贴图 Normal Mapping

法线贴图就是在原物体的凹凸表面的每个点上均作法线，通过RGB颜色通道来标记法线的方向，可以把它理解成与原凹凸表面平行的另一个不同的表面，但实际上它又只是一个光滑的平面。对于视觉效果而言，它的效率比原有的凹凸表面更高，若在特定位置上应用光源，可以让细节程度较低的表面生成高细节程度的精确光照方向和反射效果。

核心代码：

```
Vec2f uv = _uv_value * bar;
Vec3f = proj<3>(_MIT_transform *
                embed<4>(model->normal(uv))).normalize();
float intensity = std::max(0.f, n*l);
color = model->diffuse(uv) * intensity;
```

7.7.3 高光贴图 Specular Mapping

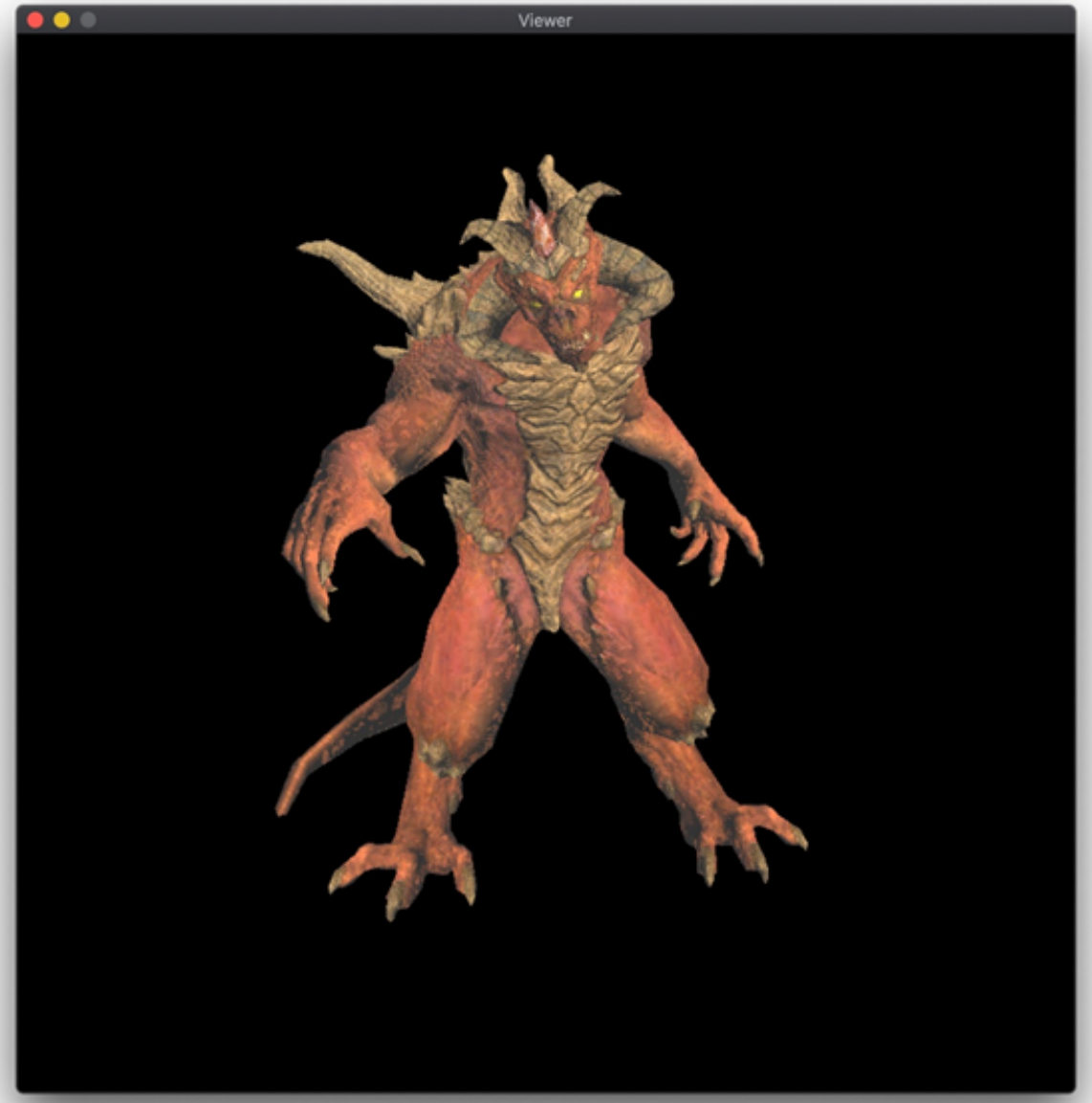
高光贴图是用来表现当光线照射到模型表面时,其表面属性的.(如金属和皮肤、布、塑料反射不同量的光)从而区分不同材质。高光贴图主要表现镜面反射和物体表面的高光颜色。

核心代码：

```
Vec2f uv = _uv_value * bar;
Vec3f n = proj<3>(_MIT_transform *
                  embed<4>(model ->normal(uv))).normalize();
Vec3f l = proj<3>(_M_transform  *embed<4>(light_dir)).normalize();
Vec3f r = (n*(n*l*2.f) - l).normalize();
float spec = pow(std::max(r.z, 0.0f), model->specular(uv));
float diff = std::max(0.f, n*l);
TGAColor c = model->diffuse(uv);
for (int i = 0; i < 3; i++)
    color[i] = std::min<float>(40 + c[i] * (7.2*diff + .6*spec), 255);
```

7.2 纹理着色器 General Shader

纹理着色器中，片段着色器根据漫反射贴图、法线贴图和高光贴图进行着色。



7.3 深度着色器 Depth Shader

深度着色器中，以光源为相机视角，片段着色器以深度值进行着色。

核心代码：

```
Vec3f p = varying_tri * bar;
color = TGAColor(255, 255, 255)*(p.z / depth);
```

7.4 阴影着色器 Shadow Shader

阴影着色器需要先调用深度着色器，对阴影区域以 shadowbuffer 进行记录，在第二次着色时，进行 shadow mapping。

核心代码：

```
Vec4f shadow_position = _shadow_trasnform * embed<4>(_tri_value*bar);
shadow_position = shadow_position / shadow_position[3];
int screen_position = int(shadow_position[0]) + int(shadow_position[1])*width;
float shadow = .3 + .7*(shadowbuffer[screen_position] < shadow_position[2] +
43.34);
Vec2f uv = _uv_value * bar;
Vec3f n = proj<3>(uniform_MIT*embed<4>(model->normal(uv))).normalize();
Vec3f l = proj<3>(uniform_M  *embed<4>(light_dir)).normalize();
Vec3f r = (n*(n*l*2.f) - l).normalize();
float spec = pow(std::max(r.z, 0.0f), model->specular(uv));
float diff = std::max(0.f, n*l);
TGAColor c = model->diffuse(uv);
for (int i = 0; i < 3; i++) color[i] = std::min<float>(40 + c[i] * shadow *
                                                        (7.2*diff + .6*spec),
255);
```

7.5 环境光遮蔽 Ambient Occlusion

假设每个像素点都向相邻的八个方向反射等强度的环境光，以 zbuffer 记录每个像素点的光强，将其看作高度图，通过计算每个像素在八个方向上的的最大斜率，估算立体角，以立体角为参数进行着色。

核心代码:

```
for (int x = 0; x < width; x++) {
    for (int y = 0; y < height; y++) {
        if (ambient_zbuffer[x + y * width] < -1e5) continue;
        float total = 0;
        for (float a = 0; a < M_PI * 2 - 1e-4; a += M_PI / 4) {
            total += M_PI / 2 -
                max_elevation_angle(ambient_zbuffer,Vec2f(x, y),
                                    Vec2f(cos(a), sin(a)));
        }
        total /= (M_PI / 2) * 8;
        total = pow(total, 100.f);
        ambient.set(x, y, TGAColor(total * 255, total * 255, total * 255));
    }
}

float max_elevation_angle(float *zbuffer, Vec2f p, Vec2f dir) {
    float maxangle = 0;
    for (float t = 0.; t < 1000.; t += 7.) {
        Vec2f cur = p + dir * t;
        if (cur.x >= width || cur.y >= height || cur.x < 0 || cur.y < 0) return maxangle;

        float distance = (p - cur).norm();
        if (distance < 7.f) continue;
        float elevation = zbuffer[int(cur.x) + int(cur.y)*width]
            - zbuffer[int(p.x) + int(p.y)*width];
        maxangle = std::max(maxangle, atanf(elevation / distance));
    }
    return maxangle;
}
```

8. 参考资料

- 知乎 - 如何开始用 C++ 写一个光栅化渲染器？
- truevision的技术指南附录 - TGA Image Format
- GitHub - GLFW Source Code
- Software Renderer - A shader-based software renderer written from scratch in C89
- 维基百科 - 重心坐标
- 维基百科 - 布雷森汉姆直线算法
- CSDN Blog - SoftRenderer&RenderPipeline（从迷你光栅化软渲染器的实现看渲染流水线）
- CSDN Blog - Shader中贴图知识汇总：漫反射贴图、凹凸贴
- CSDN Blog - threejs- z-fighting 问题
- OpenGL Tutorial - Tutorial 16 : Shadow mapping
- OpenGL Tutorial - Tutorial 13 : Normal Mapping