目录

[1、TriangleRotation 1](#_Toc78230690)

[2、Triangles and Z-buffering 5](#_Toc78230691)

[3、Pipeline and Shading 6](#_Toc78230692)

# 1、TriangleRotation

**目标：一个三角形矩阵通过模型变换矩阵，视图变换矩阵，投影变换矩阵相乘，经过光栅化器，画在屏幕上。实现三角形绕z轴旋转。**

# 2、Triangles and Z-buffering

**说明：—在屏幕上画出一个实心三角形， 换言之，栅格化一个三角形。**

# 3、Pipeline and Shading

**说明：在代码中添加了Object Loader(用于加载三维模型), Fragment Shader，并且支持了纹理映射。**

# 1、TriangleRotation

**目标：一个三角形矩阵通过模型变换矩阵，视图变换矩阵，投影变换矩阵相乘，经过光栅化器，画在屏幕上。实现三角形绕z轴旋转。**

float angle = 0;

bool command\_line = false;

//Eigen::Vector3f axis(1,1,1);

std::string filename = "output.png";

if (argc >= 3) {

command\_line = true;

angle = std::stof(argv[2]); // -r by default

if (argc == 4) {

filename = std::string(argv[3]);

}

else

return 0;

}

以上一段定义一些变量用于给定角度和画线，条件分支作为命令行运行的参数分支，若运行的命令行参数大于3,则将command\_line变量设为true，之后会用这个变量的值作为分支条件来确定程序是动态运行还是将旋转某角度的图像存储到本地。

之后定义了几个变量，分别是r, eyepos,pos,ind，定义了一个确定宽高参数的屏幕，人眼（相机）的坐标位置，三角形三个定点的坐标，和这三个点的索引值。通过loadposition与load\_indices将坐标点和索引值赋给光栅化器（rasterizer)，其定义在rasterizer.hpp中的rasterizer类中。

之后main函数中根据命令行参数的数目（即command\_line)变量的值来决定分支，而两个分支的主体部分并没有什么不同，区别仅仅在于是否保存图片到文件和用键盘进行实时旋转控制上。（其中键盘参数获得通过waitkey调用，返回值为a，左旋10度，返回值为d右旋10度，返回值为27（即键盘对应esc键）程序退出循环直接结束）

首先变换先要进行模型变换，本次要求只实现绕z轴的变换：

Eigen::Matrix4f get\_model\_matrix(float rotation\_angle)

{

Eigen::Matrix4f model = Eigen::Matrix4f::Identity();

// TODO: Implement this function

// Create the model matrix for rotating the triangle around the Z axis.

// Then return it.

Eigen::Matrix4f translate;

translate << cos(rotation\_angle/180.0\*acos(-1)), -sin(rotation\_angle/180.0\*acos(-1)), 0.0, 0.0, sin(rotation\_angle/180.0\*acos(-1)),

cos(rotation\_angle/180.0\*acos(-1)), 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0;

model = translate \* model;

return model;

}

之后是视图变换，本项目中已经填好，不过值得一提的是，本项目中设定相机的初始方向已经是正确的方向，所以并不需要进行旋转摆正相机，只需将相机平移到原点即可（代码是给定的就不放了）。然后是投影矩阵，首先要搞清给定的四个参数含义，eyefov代表垂直的可视角度，aspect\_ratio是宽高比，zNear和zFar分别为近远平面的z轴坐标。前文提到，透视变换即就是先透视投影变换，再做正交投影变换。所以可以先写出透视投影矩阵（其中参数只有n,f），然后通过计算出b,t,l,r的值，写出正交投影的矩阵，二者相乘可得到结果：具体代码如下：

Eigen::Matrix4f get\_projection\_matrix(float eye\_fov, float aspect\_ratio,

float zNear, float zFar)

{

// Students will implement this function

Eigen::Matrix4f projection = Eigen::Matrix4f::Identity();

// TODO: Implement this function

// Create the projection matrix for the given parameters.

// Then return it.

Eigen::Matrix4f m;

m << zNear, 0, 0, 0,

0, zNear, 0, 0,

0, 0, zNear + zFar, -zNear \* zFar,

0, 0, 1, 0;

float halve = eye\_fov/2\*MY\_PI/180;

float top = tan(halve) \* zNear;

float bottom = -top;

float right = top \* aspect\_ratio;

float left = -right;

Eigen::Matrix4f n, p;

n << 2/(right - left), 0, 0, 0,

0, 2/(top - bottom), 0, 0,

0, 0, 2/(zNear - zFar), 0,

0, 0, 0, 1;

p << 1, 0, 0, -(right + left)/2,

0, 1, 0, -(top + bottom)/2,

0, 0, 1, -(zFar + zNear)/2,

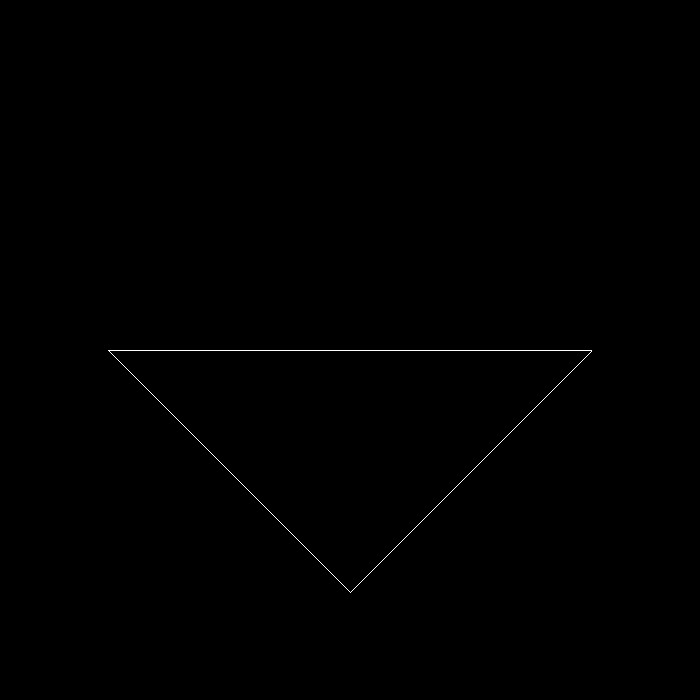
0, 0, 0, 1;

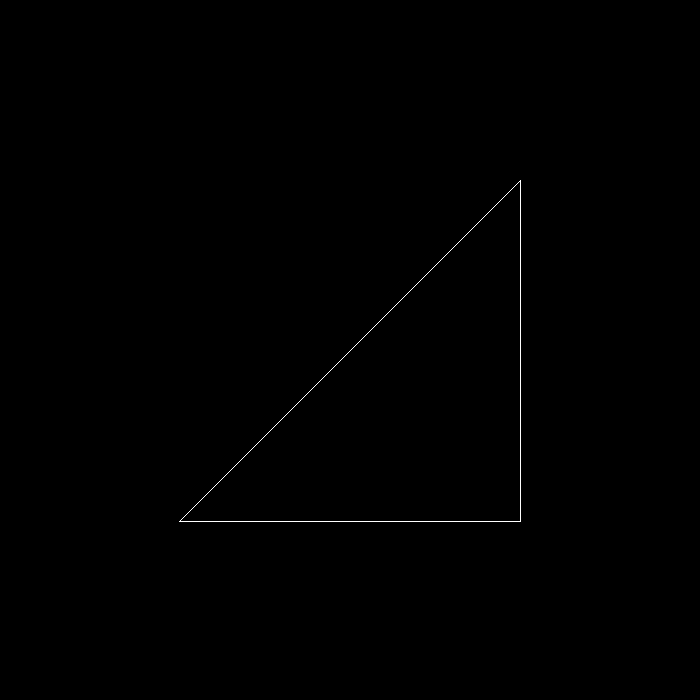
projection = n \* p \* m;

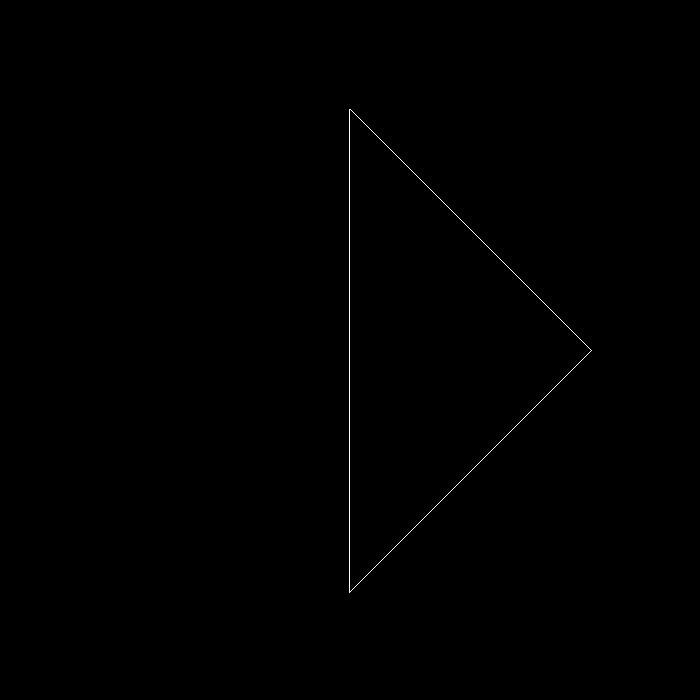
return projection;

}

结果演示：







# 2、Triangles and Z-buffering

**说明：—在屏幕上画出一个实心三角形， 换言之，栅格化一个三角形。**

rasterize\_triangle(const Triangle& t)

该函数的内部工作流程如下：

1. 创建三角形的 2 维 bounding box。

2. 遍历此 bounding box 内的所有像素（使用其整数索引）。然后，使用像素中

心的屏幕空间坐标来检查中心点是否在三角形内。

3. 如果在内部，则将其位置处的插值深度值 (interpolated depth value) 与深度

缓冲区 (depth buffer) 中的相应值进行比较。

4. 如果当前点更靠近相机，请设置像素颜色并更新深度缓冲区 (depth buffer)。

代码如下：

#include static bool insideTriangle(float x, float y,const Vector3f\* v)

{

std::array arr3f = {

Vector3f(v[0].x()- v[1].x(), v[0].y()-v[1].y() , 1.0f),

Vector3f(v[1].x()- v[2].x(), v[1].y()-v[2].y() , 1.0f),

Vector3f(v[2].x()- v[0].x(), v[2].y()-v[0].y() , 1.0f),

};

std::array point3f = {

Vector3f(x- v[1].x(), y-v[1].y() , 1.0f),

Vector3f(x- v[2].x(), y-v[2].y() , 1.0f),

Vector3f(x- v[0].x(), y-v[0].y() , 1.0f),

};

float i1 = ( arr3f[0].y() \* point3f[0].x() ) - (arr3f[0].x() \* point3f[0].y() );

float i2 = ( arr3f[1].y() \* point3f[1].x() ) - (arr3f[1].x() \* point3f[1].y() );

float i3 = ( arr3f[2].y() \* point3f[2].x() ) - (arr3f[2].x() \* point3f[2].y() );

return (i1>0 && i2>0 && i3>0 ) ||(i1<0 && i2<0 && i3<0) ? true : false;

}

bool rst::rasterizer::z\_bufffun(int x, int y, float z\_buff){

if ( z\_buff < depth\_buf[get\_index(x,y)] ){

depth\_buf[get\_index(x,y)] = z\_buff;

return true;

}else{

return false;

}

}

void rst::rasterizer::rasterize\_triangle(const Triangle& t) {

auto v = t.toVector4();

int ytop, ybottom, xl, xr;

xr = std::max( std::max(v[0].x(), v[1].x() ) , v[2].x() );

xl = std::min( std::min(v[0].x(), v[1].x()) ,v[2].x() );

ybottom = std::max( std::max(v[0].y(), v[1].y() ) ,v[2].y() );

ytop = std::min( std::min(v[0].y(), v[1].y() ) ,v[2].y() );

for (int y = ytop; y < ybottom; y++ ){

for(int x = xl; x < xr; x++){

if (insideTriangle( (float)x+0.5f,(float)y+0.5f, t.v ) ){

float al, be, ga, z\_interpolated ;

auto p = computeBarycentric2D( x,y,t.v );

al = std::get<0>(p);

be = std::get<1>(p);

ga = std::get<2>(p);

float k = 1.0f / (al / v[0].w() + be/v[1].w() + ga/v[2].w() );

z\_interpolated = al\* v[0].z()/v[0].w() + be\* v[1].z()/v[1].w() + ga \* v[2].z()/v[2].w() ;

z\_interpolated = z\_interpolated\*k;

Eigen::Vector3f point = Eigen::Vector3f(x, y, z\_interpolated);

if ( z\_bufffun(x,y,z\_interpolated) ){

set\_pixel(point, t.getColor());

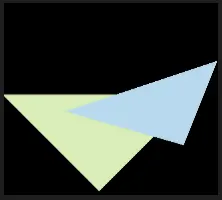
}

}

}

}

}



# 3、Pipeline and Shading

**说明：在代码中添加了Object Loader(用于加载三维模型), Fragment Shader，并且支持了纹理映射。**

代码结构：

Aabb 双for循环

If 判断点是否在三角形内

计算点在三角形各顶点系数，求出z值，即深度。

If 判断当前点是否是最浅

+法向量

+颜色

+纹理颜色

创建fragment\_shader\_payload实例

颜色=fragment\_shader(实例)；

着色（点，颜色）

rasterize\_triangle(const Triangle& t) in rasterizer.cpp:

在此处实现法向量、颜色、纹理颜色的插值。

void rst::rasterizer::rasterize\_triangle(const Triangle& t, const std::array& view\_pos)

{

auto v = t.toVector4();

//top xl =min bottom xr=max

int xr = std::max( std::max(v[0].x(), v[1].x() ) , v[2].x() );

int xl = std::min( std::min(v[0].x(), v[1].x()) ,v[2].x() );

int ybottom = std::max( std::max(v[0].y(), v[1].y() ) ,v[2].y() );

int ytop = std::min( std::min(v[0].y(), v[1].y() ) ,v[2].y() );

//AABB

for (int y = ytop; y <= ybottom; y++ ){

for(int x = xl; x <= xr; x++){

if (insideTriangle( (float)x+0.5f,(float)y+0.5f, t.v ) ){

auto p = computeBarycentric2D( (float)x+0.5f,(float)y+0.5f,t.v );

float al = std::get<0>(p);

float be = std::get<1>(p);

float ga = std::get<2>(p);

float k = 1.0f / (al / v[0].w() +be/ v[1].w()+ ga/v[2].w() );

float z\_interpolated = al\* v[0].z()/v[0].w() + be\* v[1].z()/v[1].w() + ga \* v[2].z()/v[2].w() ;

z\_interpolated = z\_interpolated\*k;

if ( z\_interpolated < depth\_buf[get\_index(x,y)] ){

depth\_buf[get\_index(x,y)] = z\_interpolated;

Eigen::Vector3f color = al \* t.color[0] + be\*t.color[1] + ga\*t.color[2];

Eigen::Vector3f normal = al \* t.normal[0] + be\*t.normal[1] + ga\*t.normal[2];

Eigen::Vector2f coords = al \* t.tex\_coords[0] + be\*t.tex\_coords[1] + ga\*t.tex\_coords[2];

Eigen::Vector3f point = al \* view\_pos[0] + be\* view\_pos[1] + ga\*view\_pos[2];

auto a = sqrt( (normal[0]\* normal[0]) + (normal[1]\*normal[1]) + (normal[2]\*normal[2]) );

normal[0] = normal[0] / a;

normal[1] = normal[1] / a;

normal[2] = normal[2] / a;

//fragment\_shader\_payload(const Eigen::Vector3f& col, const Eigen::Vector3f& nor,const Eigen::Vector2f& tc, Texture\* tex, Eigen::Vector3f& pos) :color(col), normal(nor), tex\_coords(tc), texture(tex), view\_pos(pos) {}

auto shader\_load = fragment\_shader\_payload(color, normal,coords, &\*texture );

//texture is std::optional!= Texture\*

shader\_load.view\_pos = point;

auto new\_color = fragment\_shader( shader\_load );

set\_pixel({x,y}, new\_color);

}

}

}

}

}

phong\_fragment\_shader() in main.cpp: 实现 Blinn-Phong 模型计算 Fragment Color。

实现结果：



{

Eigen::Vector3f light\_point3f = light.position - point;

float light\_point\_lenght = light\_point3f.x()\*light\_point3f.x() + light\_point3f.y()\* light\_point3f.y() + light\_point3f.z()\* light\_point3f.z() ;

Eigen::Vector3f eye\_point3f = eye\_pos - point ;

Eigen::Vector3f light\_eye\_lized = (eye\_point3f.normalized() + light\_point3f.normalized() ).normalized();

Eigen::Vector3f light\_int = light.intensity / light\_point\_lenght;

//diffuse ==

//point\_light\_sum= k\_loss \* light.intensity / light\_point\_lenght^2 \* max(0, dot( light\_point\_lenght.normalized,payload.normal) )

//eye\_light = point\_light\_sum / eye\_point\_lenght^2

Eigen::Vector3f point\_light\_diff = 1.0f \* light\_int \* std::max(0.0f , normal.dot( light\_point3f.normalized() ) );

//ambient == ka \*amb\_light\_intensity

//Specular ==

//point\_light\_toplight= k\_loss \* light.intensity / light\_point\_lenght^2

// \* ( max(0, dot( (eye\_point.normalized()+light\_point.normalized() ).normalized()) )^p )

Eigen::Vector3f point\_light\_toplight = 1.0f \* light\_int \* std::pow (std::max(0.0f , normal.dot( light\_eye\_lized ) ) , p);

result\_color += Eigen::Vector3f( ka.x()\*amb\_light\_intensity.x(), ka.y()\*amb\_light\_intensity.y(), ka.z()\*amb\_light\_intensity.z() )

+ Eigen::Vector3f(kd.x()\*point\_light\_diff.x() ,kd.y()\*point\_light\_diff.y(), kd.z()\*point\_light\_diff.z() )

+ Eigen::Vector3f(ks.x()\*point\_light\_toplight.x() ,ks.y()\*point\_light\_toplight.y(), ks.z()\*point\_light\_toplight.z() )

;

}

