课2

课2：三角形光栅化和背面剔除

填充三角形

Hi，你们好，这是我。

更确切的说，这是在接下来的一到两个小时的课程中我们会渲染出来的我的脸部模型。在上一堂课的最后我们画出来了一个三维的网格模型。而在本堂课上，我们将填充多边形而不仅仅是三角形。事实上，OpenGL总是把多边形拆成各式各样的三角形，所以，它也就不必去考虑复杂的情形。

让我提醒你一下，本系列课程在设计上是为了让你自己去实现程序。我上面说的，要用一到两个小时让你能实现如上图所示的图片，不是让你去阅读我的代码。而是让你自己从头开始实现自己的代码。我的代码是为了提供给你作为一个实现效果的对比。我的程序很糟糕，而你的可以做到更好。不要简单的复制粘贴，任何的讨论和问题都是欢迎的。

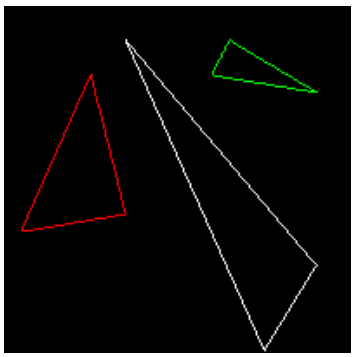
老办法：扫线

由于任务是画二维的三角形。对于积极的学生来说这通常要花费好几个小时，即使他们是垃圾程序员。上一堂课中我们我们见识了Bresenham画线算法。今天这堂课的内容是画一个填充满的三角形。很有趣，但是这任务不简单。我不确定原因，但是我知道我说的是对的。我大多数学生都不能完成这个简单任务。一般来说最初的形态会像是这样：

```
void triangle(Vec2i t0, Vec2i t1, Vec2i t2, TGAImage &image, TGAColor
color) {
    line(t0, t1, image, color);
    line(t1, t2, image, color);
    line(t2, t0, image, color);
}

// ...

Vec2i t0[3] = {Vec2i(10, 70),   Vec2i(50, 160),  Vec2i(70, 80)};
Vec2i t1[3] = {Vec2i(180, 50),  Vec2i(150, 1),   Vec2i(70, 180)};
Vec2i t2[3] = {Vec2i(180, 150), Vec2i(120, 160), Vec2i(130, 180)};
triangle(t0[0], t0[1], t0[2], image, red);
triangle(t1[0], t1[1], t1[2], image, white);
triangle(t2[0], t2[1], t2[2], image, green);
```



和以往一样，上述的代码是能在GitHub的这里找到的。代码很简单：我给你提供了三个三角形作为代码的初期调试对比。如果我们在三角函数里面调用Line()，我们就能得到三角形的轮廓，那么如何去画一个填充满的三角形呢?

一个画三角形的好方法必须满足如下的需求：

它应该快，而且简单

它应该具有对称性，图形不应该会因为传入函数的定点顺序问题导致图像的区别

如果有两个三角形具有相同的顶点数据，在光栅化计算后，这两个三角形中间不应该出现空洞

我们能添加更多要求，但在在此之前，让我们先完成上述的需求，通常来说，使用扫线应该像这样：

1.让三角形顶点按照它们的Y轴进行排序

2.同时对三角形的左右边进行光栅化处理

3.在三角形的左右边界点上画一条水平线段

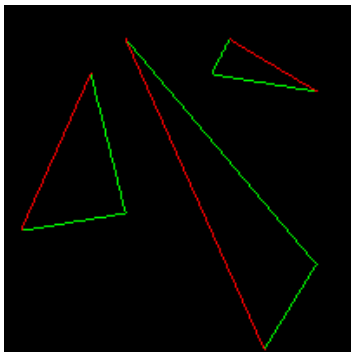
在这一点上我的学生开始疑惑，哪一条线段是左边，哪一条是右边？毕竟，一个三角形有三条线段，通常来说我会留给我的学生大约一个小时的时间去思考：另外，单纯阅读我的代码要比将我的代码与你的进行对比的作用要小得多。

一个小时之后

我是如何画一个三角形的？如果你有更好的想法，我会非常开心去接受它。让我们假定我们有t0,t1,t2三个点，它们按照y轴进行排序。这样一来，边界A就在t0和t2之间，边界B就在t0和t1之间。另一条边就在t1和t2之间。

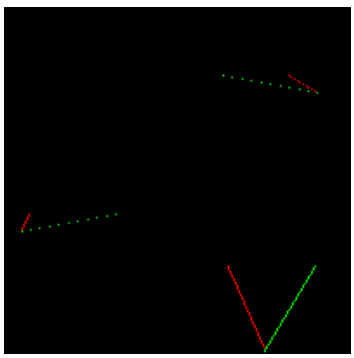
```
void triangle(Vec2i t0, Vec2i t1, Vec2i t2, TGAImage &image, TGAColor color) {
    // sort the vertices, t0, t1, t2 lower-to-upper (bubblesort yay!)
    if (t0.y>t1.y) std::swap(t0, t1);
    if (t0.y>t2.y) std::swap(t0, t2);
    if (t1.y>t2.y) std::swap(t1, t2);
    line(t0, t1, image, green);
    line(t1, t2, image, green);
    line(t2, t0, image, red);
}
```

如下所示，边界A是红色，边界B是绿色



不幸的是，边界B做了两部分，让我们切割水平线来画出三角形下面一半：

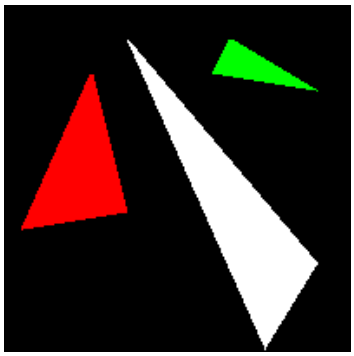
```
void triangle(Vec2i t0, Vec2i t1, Vec2i t2, TGAImage &image, TGAColor color) {
    // sort the vertices, t0, t1, t2 lower-to-upper (bubblesort yay!)
    if (t0.y>t1.y) std::swap(t0, t1);
    if (t0.y>t2.y) std::swap(t0, t2);
    if (t1.y>t2.y) std::swap(t1, t2);
    int total_height = t2.y-t0.y;
    for (int y=t0.y; y<=t1.y; y++) {
        int segment_height = t1.y-t0.y+1;
        float alpha = (float)(y-t0.y)/total_height;
        float beta  = (float)(y-t0.y)/segment_height; // be careful with divisions by zero
        Vec2i A = t0 + (t2-t0)*alpha;
        Vec2i B = t0 + (t1-t0)*beta;
        image.set(A.x, y, red);
        image.set(B.x, y, green);
    }
}
```



可见这些线段不都是连续的，上一堂课的时候我们很难去得到连续的线段并且我在这里没有旋转图像，（还记得xy交换的步骤吗？）为什么呢？我们之后要填充三角形，这就是原因。如果我们用水平线连接对应的点，图片会如下所示：

现在，让我们把剩下的一半三角形画出来，我们可以用第二遍循环去做：

```
void triangle(Vec2i t0, Vec2i t1, Vec2i t2, TGAImage &image, TGAColor color) {
    // sort the vertices, t0, t1, t2 lower-to-upper (bubblesort yay!)
    if (t0.y>t1.y) std::swap(t0, t1);
    if (t0.y>t2.y) std::swap(t0, t2);
    if (t1.y>t2.y) std::swap(t1, t2);
    int total_height = t2.y-t0.y;
    for (int y=t0.y; y<=t1.y; y++) {
        int segment_height = t1.y-t0.y+1;
        float alpha = (float)(y-t0.y)/total_height;
        float beta  = (float)(y-t0.y)/segment_height; // be careful with divisions by zero
        Vec2i A = t0 + (t2-t0)*alpha;
        Vec2i B = t0 + (t1-t0)*beta;
        if (A.x>B.x) std::swap(A, B);
        for (int j=A.x; j<=B.x; j++) {
            image.set(j, y, color); // attention, due to int casts t0.y+i != A.y
        }
    }
    for (int y=t1.y; y<=t2.y; y++) {
        int segment_height =  t2.y-t1.y+1;
        float alpha = (float)(y-t0.y)/total_height;
        float beta  = (float)(y-t1.y)/segment_height; // be careful with divisions by zero
        Vec2i A = t0 + (t2-t0)*alpha;
        Vec2i B = t1 + (t2-t1)*beta;
        if (A.x>B.x) std::swap(A, B);
        for (int j=A.x; j<=B.x; j++) {
            image.set(j, y, color); // attention, due to int casts t0.y+i != A.y
        }
    }
}
```



这样图像就完整了，但是我不想看见相同的代码两次。这就是为什么我们让它可读性变差，但是更容易维护的原因。

```
void triangle(Vec2i t0, Vec2i t1, Vec2i t2, TGAImage &image, TGAColor color) {
    if (t0.y==t1.y && t0.y==t2.y) return; // I dont care about degenerate triangles
    // sort the vertices, t0, t1, t2 lower-to-upper (bubblesort yay!)
    if (t0.y>t1.y) std::swap(t0, t1);
    if (t0.y>t2.y) std::swap(t0, t2);
    if (t1.y>t2.y) std::swap(t1, t2);
    int total_height = t2.y-t0.y;
    for (int i=0; i<total_height; i++) {
        bool second_half = i>t1.y-t0.y || t1.y==t0.y;
        int segment_height = second_half ? t2.y-t1.y : t1.y-t0.y;
        float alpha = (float)i/total_height;
        float beta  = (float)(i-(second_half ? t1.y-t0.y : 0))/segment_height; // be careful: with above conditions no division by zero here
        Vec2i A =               t0 + (t2-t0)*alpha;
        Vec2i B = second_half ? t1 + (t2-t1)*beta : t0 + (t1-t0)*beta;
        if (A.x>B.x) std::swap(A, B);
        for (int j=A.x; j<=B.x; j++) {
            image.set(j, t0.y+i, color); // attention, due to int casts t0.y+i != A.y
        }
    }
}
```

这里是画2D三角形的代码

我自己采用的方法：

尽管代码不是很复杂，但扫线的源码也有一点混乱。除此之外，它确实是为了单核CPU所设计的老办法。让我们看看如下所示的伪代码:

```
triangle(vec2 points[3]) {
    vec2 bbox[2] = find_bounding_box(points);
    for (each pixel in the bounding box) {
        if (inside(points, pixel)) {
            put_pixel(pixel);
        }
    }
```

```
}
```

你喜欢它吗？我很喜欢，它可以很简单的找到一个包围盒。他不存在需要去检查一个点是否在2D三角形内部（或者任意一个多边形）的问题。

题外话：假若我需要去写代码判断一个点是否在多边形内，并且代码判断的情形在一个平面上。那么我将永远找不到这个平面。事实证明。为了尽可能解决这个问题是非常困难的。但是我们仅仅是为了填充像素，这就容易很多。

我之所以喜欢这个伪代码的另一个原因是：新手会很高兴去接受它，而更有资格的程序员则会说：”是哪个白痴写出来的这种东西？“而计算机图形学专家则会耸耸他的肩说到：”好吧，真实生活中它就是这样运行的“成千上万线程的大规模运算方式改变了思考的方式（我这里讨论的只是消费级的电脑）。

好，让我们开始吧：首先我们需要知道重心坐标的概念，给定一个2D三角形ABC和一个点P，都在笛卡尔坐标系上（xy），我们的目的是找到在点P相对于三角形ABC的重心坐标。这意味着我们需要三个数（1–u–v，u，v）因此我们能找到P点如下：

$$P = (1 - u - v)A + uB + vC$$

尽管第一眼看上去它有点吓人，但它真的很简单：假设我们给顶点A、B和C分别放置三个权重（1–u–v，u，v）。而系统的重心恰好在P点。换句话说：点P的坐标（u，v）在坐标(A,$\overrightarrow{AB}$,$\overrightarrow{AC}$)上：

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

如此一来，我们就有了向量$\overrightarrow{AB}$，$\overrightarrow{AC}$ 和$\overrightarrow{AP}$，我们需要去找到两个确切是数字u和v来表示如下的方程：

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

这是一个简单的向量方程式，或者说是一个两个方程和两个变量的直线方程组

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

我比较懒，不太想用书本上的方法去解这个直线方程组，让我们将它写成矩阵的形式：

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

这意味着我们能看见一个向量（u，v，1）与（AB.x，AC.x，PA.x）和(AB.y, AC.y, PA.y)同时正交。我希望你能知道我在说什么。小小的提示一下：在一个平面内找到两条直线的交点（这就是我一直在做的事情），只需要去计算一个叉乘。与此同时，问一问你自己：我们是怎样找到通过两个点的直线方程的?

既然如此，让我们编写新的光栅化程序：我们遍历给定三角形包围盒的所有像素点。对每一个像素点我们都计算它的重心坐标。如果存在一个负向量，那么像素点就不在这个三角形内。通过程序去看可能会更清楚：

```
#include <vector>
#include <iostream>
#include "geometry.h"
#include "tgaimage.h"

const int width  = 200;
const int height = 200;

```

```
Vec3f barycentric(Vec2i *pts, Vec2i P) {
    Vec3f u = cross(Vec3f(pts[2][0]-pts[0][0], pts[1][0]-pts[0][0], pts[0][0]-P[0]), Vec3f(pts[2][1]-pts[0][1], pts[1][1]-pts[0][1], pts[0][1]-P[1]));
    /* `pts` and `P` has integer value as coordinates
       so `abs(u[2])` < 1 means `u[2]` is 0, that means
       triangle is degenerate, in this case return something with negative coordinates */
    if (std::abs(u[2])<1) return Vec3f(-1,1,1);
    return Vec3f(1.f-(u.x+u.y)/u.z, u.y/u.z, u.x/u.z);
}

void triangle(Vec2i *pts, TGAImage &image, TGAColor color) {
    Vec2i bboxmin(image.get_width()-1,  image.get_height()-1);
    Vec2i bboxmax(0, 0);
    Vec2i clamp(image.get_width()-1, image.get_height()-1);
    for (int i=0; i<3; i++) {
        for (int j=0; j<2; j++) {
            bboxmin[j] = std::max(0,        std::min(bboxmin[j], pts[i][j]));
            bboxmax[j] = std::min(clamp[j], std::max(bboxmax[j], pts[i][j]));
        }
    }
    Vec2i P;
    for (P.x=bboxmin.x; P.x<=bboxmax.x; P.x++) {
        for (P.y=bboxmin.y; P.y<=bboxmax.y; P.y++) {
            Vec3f bc_screen  = barycentric(pts, P);
            if (bc_screen.x<0 || bc_screen.y<0 || bc_screen.z<0) continue;
            image.set(P.x, P.y, color);
        }
    }
}

int main(int argc, char** argv) {
    TGAImage frame(200, 200, TGAImage::RGB);
    Vec2i pts[3] = {Vec2i(10,10), Vec2i(100, 30), Vec2i(190, 160)};
    triangle(pts, frame, TGAColor(255, 0, 0));
    frame.flip_vertically(); // to place the origin in the bottom left corner of the image
    frame.write_tga_file("framebuffer.tga");
    return 0;
}
```

barycentric()函数展现了计算在给定三角形中的点P坐标的细节。现在让我们观察triangle()函数是怎么运作的。首先，他计算了一个通过两个点描述的包围盒这两个点分别在左下和右上。为了找到这两个点的位置我们遍历了给定三角形的所有像素点找到最小/最大的坐标。我也为包围盒添加了一个屏幕内的三角形裁剪方法，以便节省CPU去计算屏幕外的三角形的时间。恭喜，你知道如何去画一个三角形了！

平滑阴影渲染

我们已经会用空三角形去画模型。让我们用不同的颜色去填充它们。这将帮助我们了解我们是怎么填充三角形的。代码如下:

```
for (int i=0; i<model->nfaces(); i++) {
    std::vector<int> face = model->face(i);
    Vec2i screen_coords[3];
    for (int j=0; j<3; j++) {
        Vec3f world_coords = model->vert(face[j]);
        screen_coords[j] = Vec2i((world_coords.x+1.)*width/2., (world_coords.y+1.)*height/2.);
    }
    triangle(screen_coords[0], screen_coords[1], screen_coords[2], image, TGAColor(rand()%255, rand()%255, rand()%255, 255));
}
```

很简单：就像之前的做法一样，我们遍历所有的三角形，将世界坐标转换为屏幕坐标并画对应的三角形。我将在接下来的文章里详细的描述各个坐标系统。现在图片会渲染成这样：

让我们去掉这些五花八门的艳色并放一个点光源在这。显而易见：”在相同的光源强度下，多边形最亮的地方在它与光源正交的点上“

让我们比较一下：

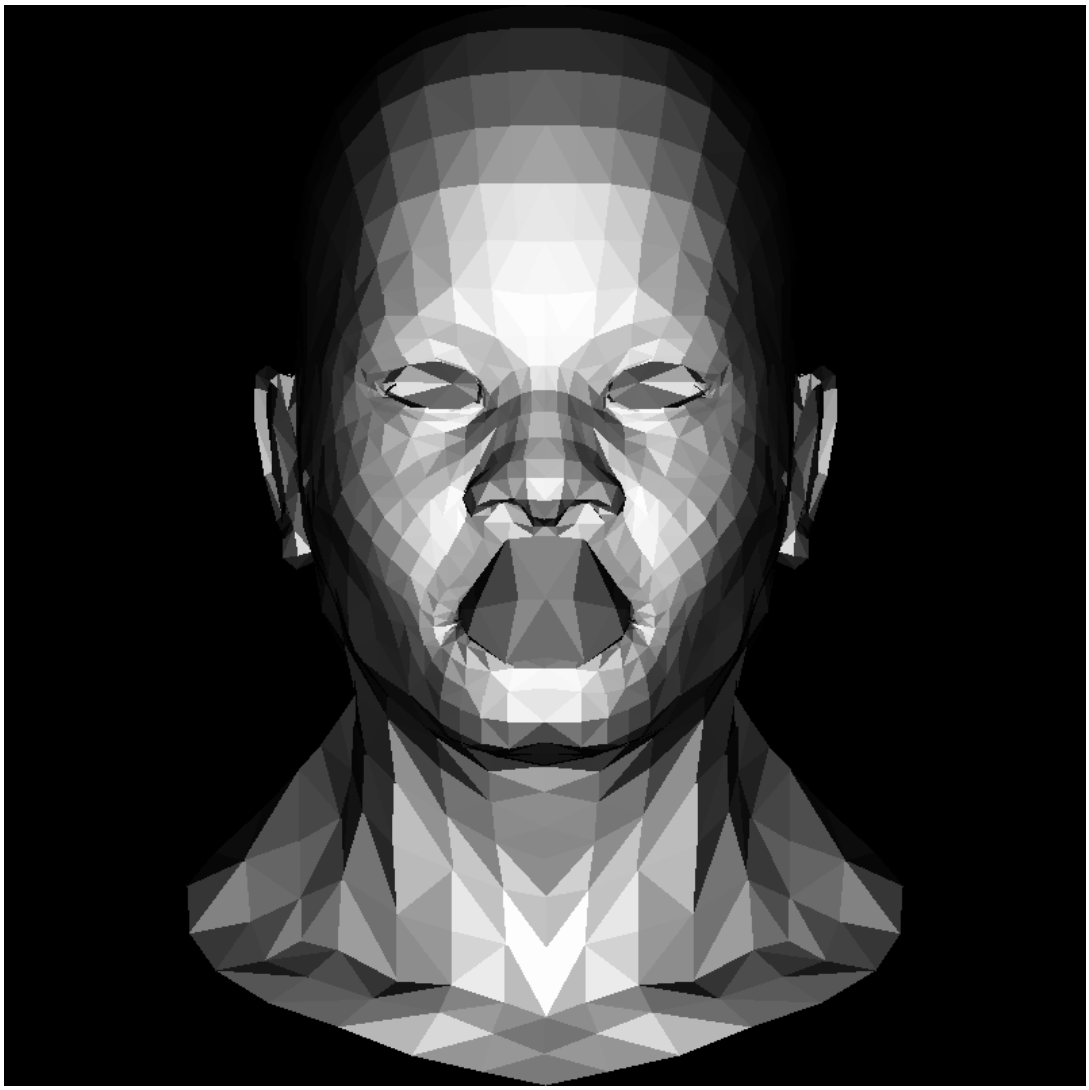
当多边形和点光源方向向量平行的时候，光照强度为0。释义：光照强度等于点光源和三角形的法线向量的点乘积。求三角形的法线向量可以简单的对三角形的任意两边进行叉乘。

最后提一点：在门课上我们将会对颜色进行线性计算。然而（128，128，128）色的亮度不是（255，255，255）的一半。我们将忽略忽略伽马校正及忽略不正确的色彩表现。

```
for (int i=0; i<model->nfaces(); i++) {
    std::vector<int> face = model->face(i);
```

```
    Vec2i screen_coords[3];
    Vec3f world_coords[3];
    for (int j=0; j<3; j++) {
        Vec3f v = model->vert(face[j]);
        screen_coords[j] = Vec2i((v.x+1.)*width/2., (v.y+1.)*height/2.);
        world_coords[j]  = v;
    }
    Vec3f n = (world_coords[2]-world_coords[0])^(world_coords[1]-world_coords[0]);
    n.normalize();
    float intensity = n*light_dir;
    if (intensity>0) {
        triangle(screen_coords[0], screen_coords[1], screen_coords[2], image, TGAColor(intensity*255, intensity*255, intensity*255, 255));
    }
}
```

但是点乘能够为负数。这说明了什么？这意味着光源来自于多边形内部。如果环境建模比较好（通常情况下）我们可以简单忽略这个三角形。这能帮我们比较快的提出部分看不见的三角形，它叫做影面剔除。



注意，再上图中，口腔被画在了嘴唇上方。这是因为我们对看不见的三角形进行了不透明裁剪。它只适用于凸面。我们将在下一堂课中使用Z-buffer解决这个问题。

这里是到现在为止的渲染代码。你发现了吗?我的面部细节更多了。好吧，我有点偷跑：这里有25万计的面.而构造人头的大概有一千个。但是我的脸确实是用上面的代码画出来的。我向你保

证我们会在接下来的文章中给这张图片添加个更多的的细节。