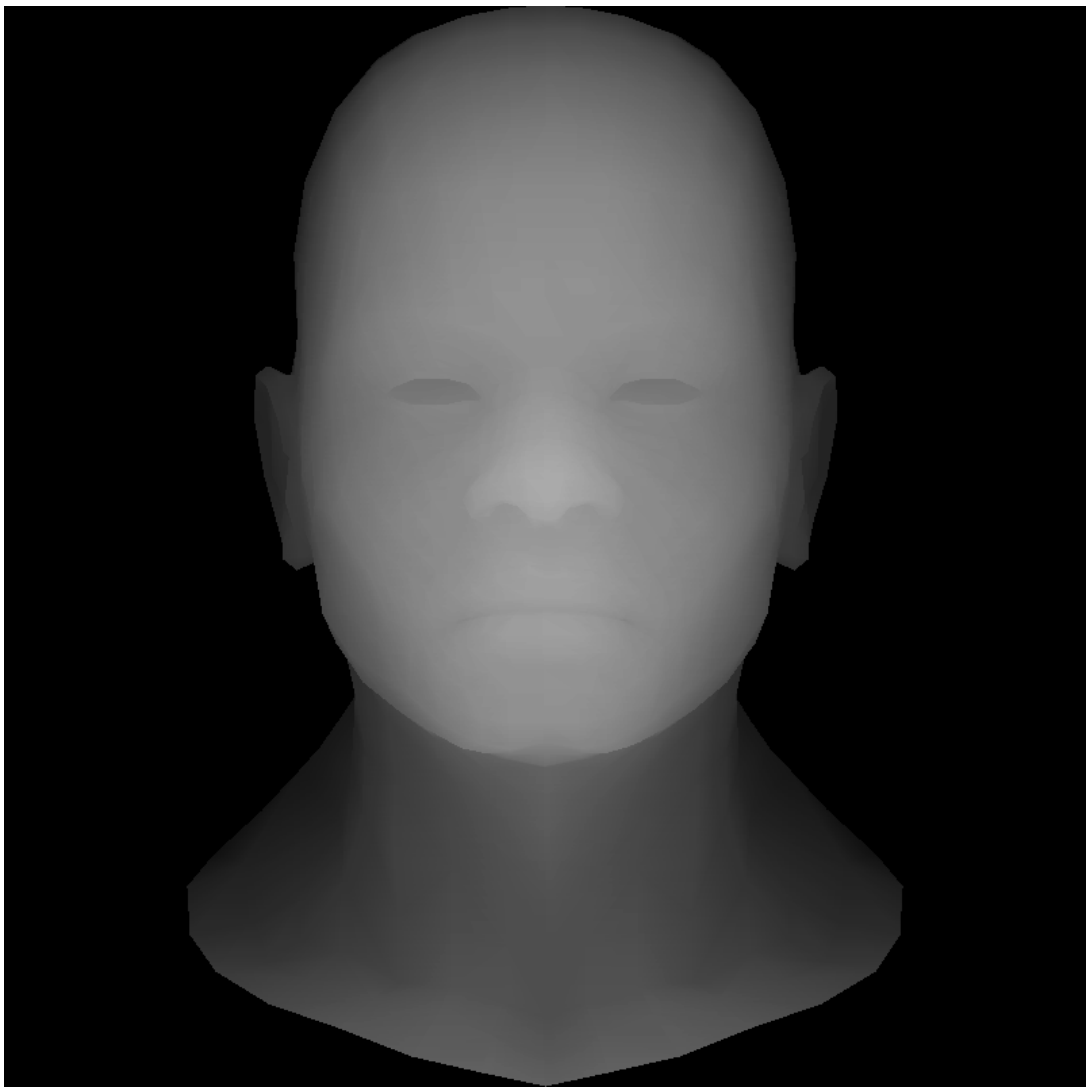
课3

课三：消除隐藏面（Z buffer）

介绍

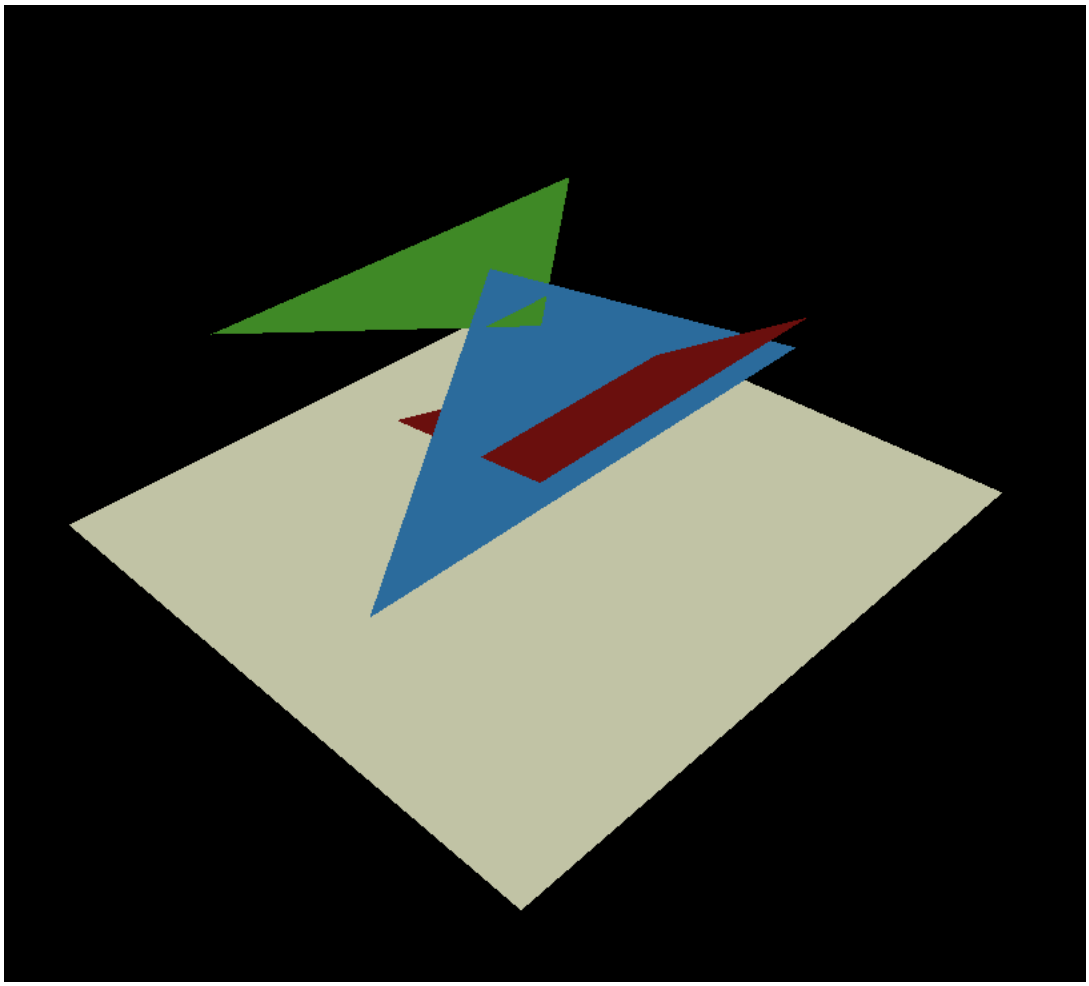
大家好，让我来给你们介绍我的Z-buffer小伙伴。他将帮我们去掉接下来我们在课程中会遇到的隐藏面的伪阴影。



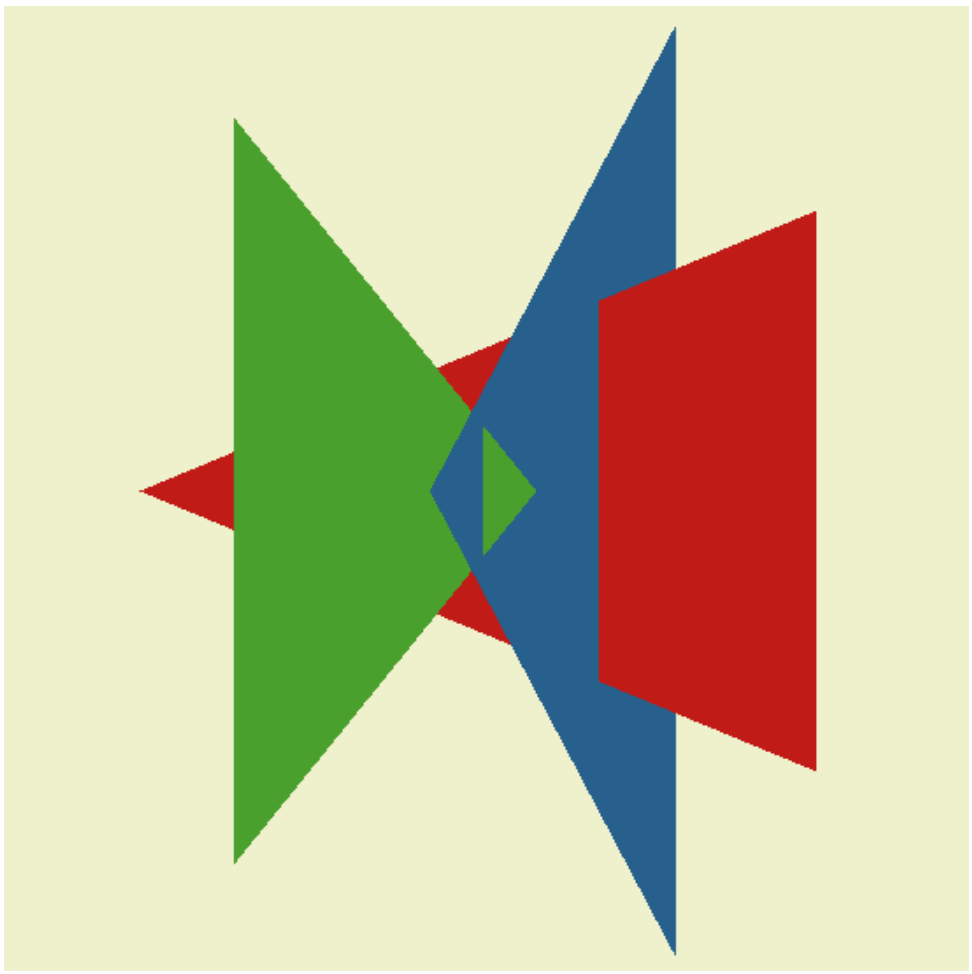
于此同时，我想提一下我们在这门课中大量使用的这个模型的制作者是Vidar Rapp.他好心的将这个模型给予我用于授课使用，但我却将它弄坏了，但是我承诺我会给这个模型恢复它的眼睛。好的，我们回到标题，理论上我们可以绘制所有的三角形而不丢弃它们。如果我们从后面渲染到前面，那么前面的面将会遮住后面的面。它称为画家算法。不幸的是它带来了很大的计算消耗：对每一次摄像机移动我们都要重新对场景的物体进行排序。一旦它们是动态场景的话。。。这甚至不是主要问题。主要问题是并不是每次计算都是正确的。

让我们尝试去渲染一个简单场景

想象一个由三个三角形组成的场景：摄像机由上往下看，我们将带颜色的三角形放在白色的屏幕上：



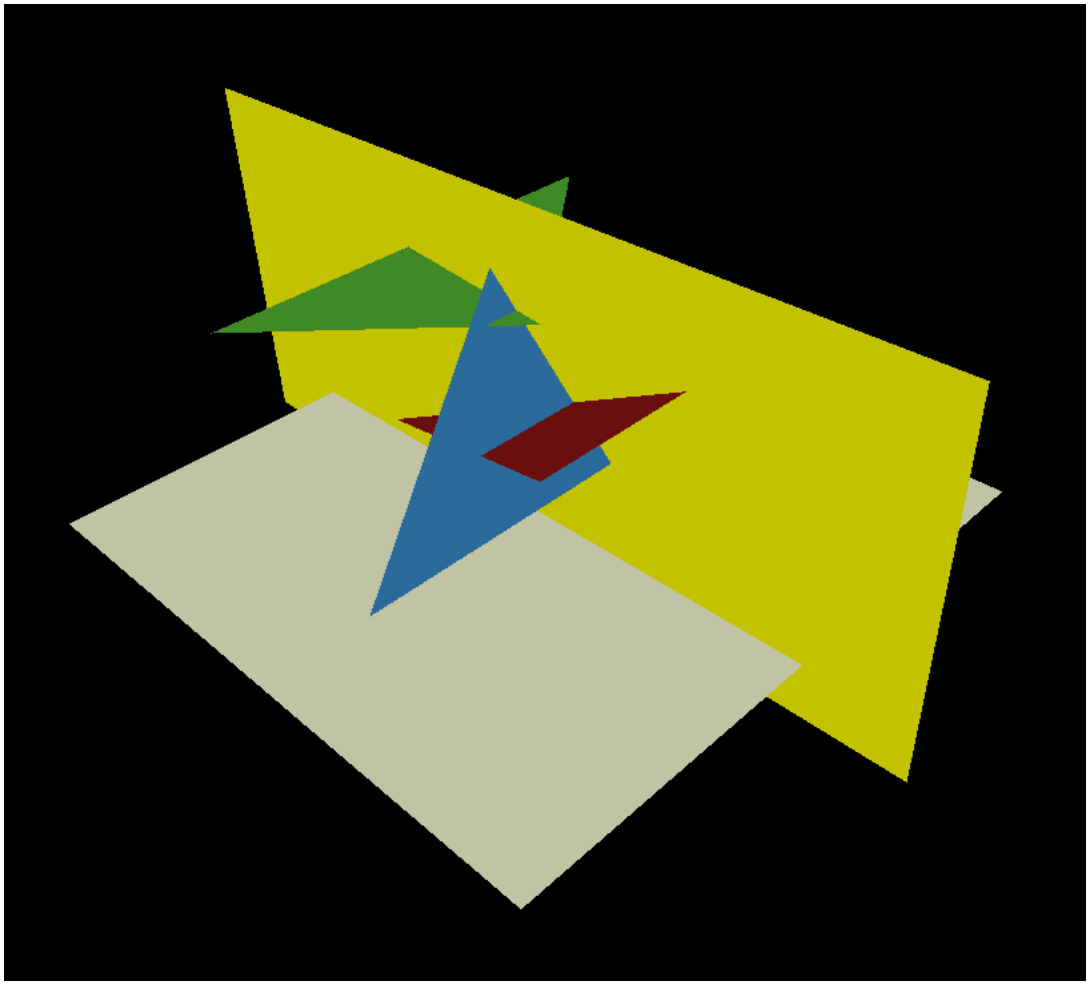
渲染后应该看起来像这样：



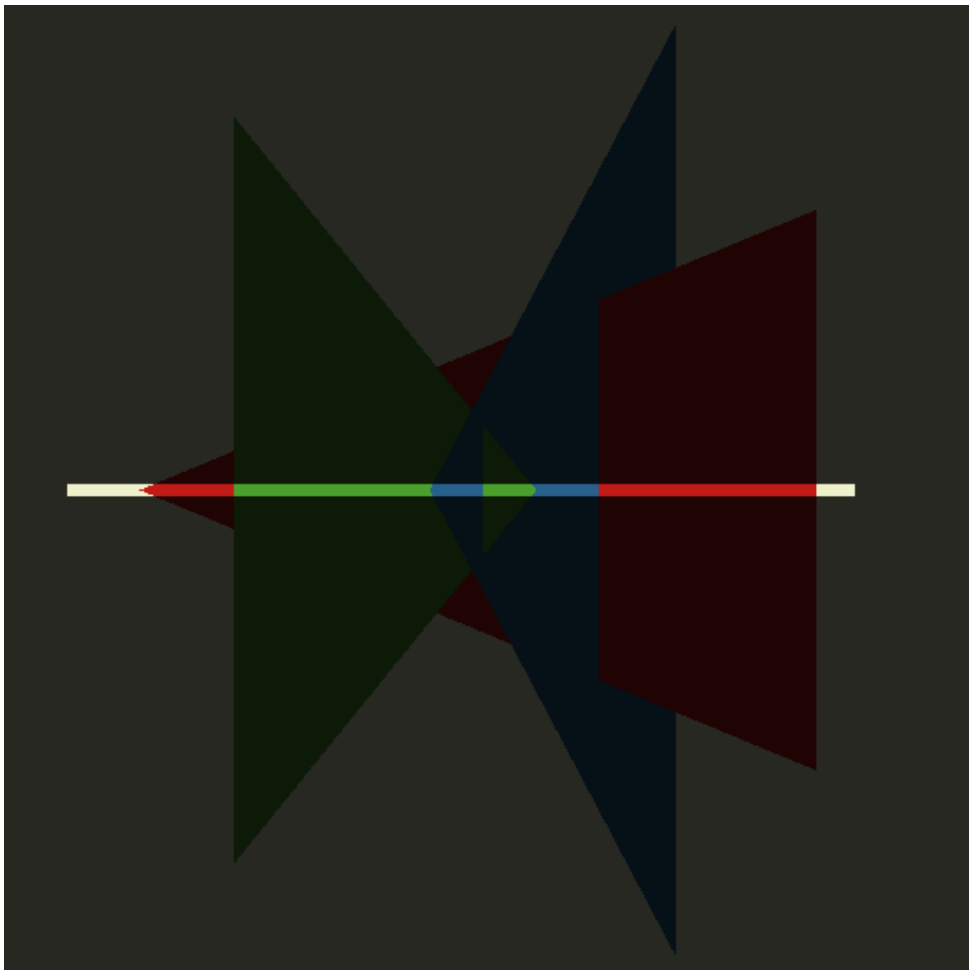
蓝色面–它是在红色的前面还是后面？画家算法在这里就不起作用了。它看起来就像把蓝色的面分成了两块（一块在红面的前面一块在后面）。而且在红色前面这一块也好像被分割成了两个–一个在绿色三角形前面一个在后面。。。我认为你现在会有一个问题了：在一个有上百万个三角形的场景里这真的很难去计算，这真的太混乱了，生活如此之短以至于不能让它变得如此混乱。

更简单一点：让我们降低一个维度。Y-buffer!

让我们暂时降低一个维度并且用一个黄色的平面裁剪上述的场景。



我的意思是我们的场景由三条线段组成（连接黄色平面和每一个三角形的交点），最终渲染结果是一个正常宽但只有一像素高的图：



老规矩，我们能在这里得到代码。我们的屏幕由两个维度组成，所以能很简单的用我们第一堂课学到的line()方法画出来这种场景。

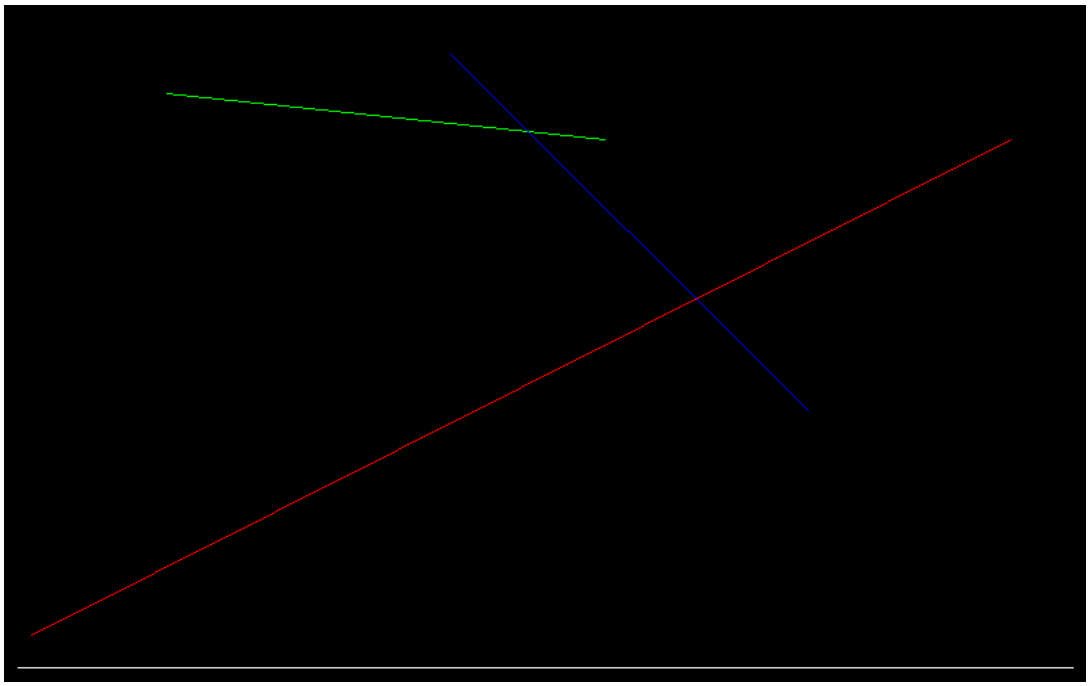
```
    { // just dumping the 2d scene (yay we have enough dimensions!)
        TGAImage scene(width, height, TGAImage::RGB);

        // scene "2d mesh"
        line(Vec2i(20, 34),   Vec2i(744, 400), scene, red);
        line(Vec2i(120, 434), Vec2i(444, 400), scene, green);
        line(Vec2i(330, 463), Vec2i(594, 200), scene, blue);

        // screen line
        line(Vec2i(10, 10), Vec2i(790, 10), scene, white);

        scene.flip_vertically(); // i want to have the origin at the left bottom corner of the image
        scene.write_tga_file("scene.tga");
    }
```

如下就是我们从侧面看这个2D场景的样子：



让我们渲染它。回想一下渲染高度是一个像素。在我的代码中我创建了一个十六像素的图以便于我们是在屏幕上有更好的阅读性。rasterize()方法仅用于写入图片渲染的第一条线。

```
TGAImage render(width, 16, TGAImage::RGB);
        int ybuffer[width];
        for (int i=0; i<width; i++) {
            ybuffer[i] = std::numeric_limits<int>::min();
        }
        rasterize(Vec2i(20, 34),   Vec2i(744, 400), render, red,
ybuffer);
        rasterize(Vec2i(120, 434), Vec2i(444, 400), render, green,
ybuffer);
        rasterize(Vec2i(330, 463), Vec2i(594, 200), render, blue,
ybuffer);
```

所以，我声明了一个宽度为1的一维数组Ybuffer。这数组被初始化负无穷大。然后使用这个图像和这个数组作为rasterize()方法的参数。这方法看起来应该是什么样的?

```
void rasterize(Vec2i p0, Vec2i p1, TGAImage &image, TGAColor color, int
ybuffer[]) {
    if (p0.x>p1.x) {
        std::swap(p0, p1);
    }
    for (int x=p0.x; x<=p1.x; x++) {
        float t = (x-p0.x)/(float)(p1.x-p0.x);
        int y = p0.y*(1.-t) + p1.y*t;
        if (ybuffer[x]<y) {
            ybuffer[x] = y;
            image.set(x, 0, color);
        }
    }
}
```

这确实很简单：遍历p0.x到p1.x所有的点并计算Y轴上的线段点。然后就能根据现有的X轴获取Ybuffer数组。如果现在的Y轴上的值比Ybuffer上的值更接近于摄像机我就把它显示在屏幕上并

更新Ybuffer。
让我们一步步来观察它。在rasterize()方法调用后红色线段显示的就是我们现在的内存。
screen:

ybuffer:

洋红色表示的是负无穷大，这些表示我们无法从屏幕上触及的地方。剩下的所有都用灰色表示，里摄像机越近颜色越浅，里摄像机越远越深。
接下来画绿色的线段。
screen:

ybuffer:

最后画蓝色。
screen:

ybuffer:

恭喜，我们在1维的屏幕上画出了2D的场景！再看一次我们的渲染结果：

回到3D

因此，为了画在一个2D屏幕上Z-buffer必须是二维的：

```
int *zbuffer = new int[width*height];
```

我个人会更倾向于将二维的数组转成一维的，就像这样：

```
int idx = x + y*width;
```

以及后面的：

```
int x = idx % width;
int y = idx / width;
```

然后我简单遍历所有的三角形并且调用rasterizer()方法在当前三角形上并引入Z-buffer。
唯一的困难是我们怎么去计算要引入的Z值。回想一下我们是如何在Ybuffer上计算Y值的。

```
int y = p0.y*(1.-t) + p1.y*t;
```

> t变量的实质是什么？事实证明（1-t,t）是p0，到p1:(x,y) = p0*(1-t) + p1*t的重心坐标(x,y)。所以办法就是在三角形的光栅化后的版本中取得重心，对每一个我们想画的像素用它自身重心的坐标乘以三角形顶点光栅化后的Z值：

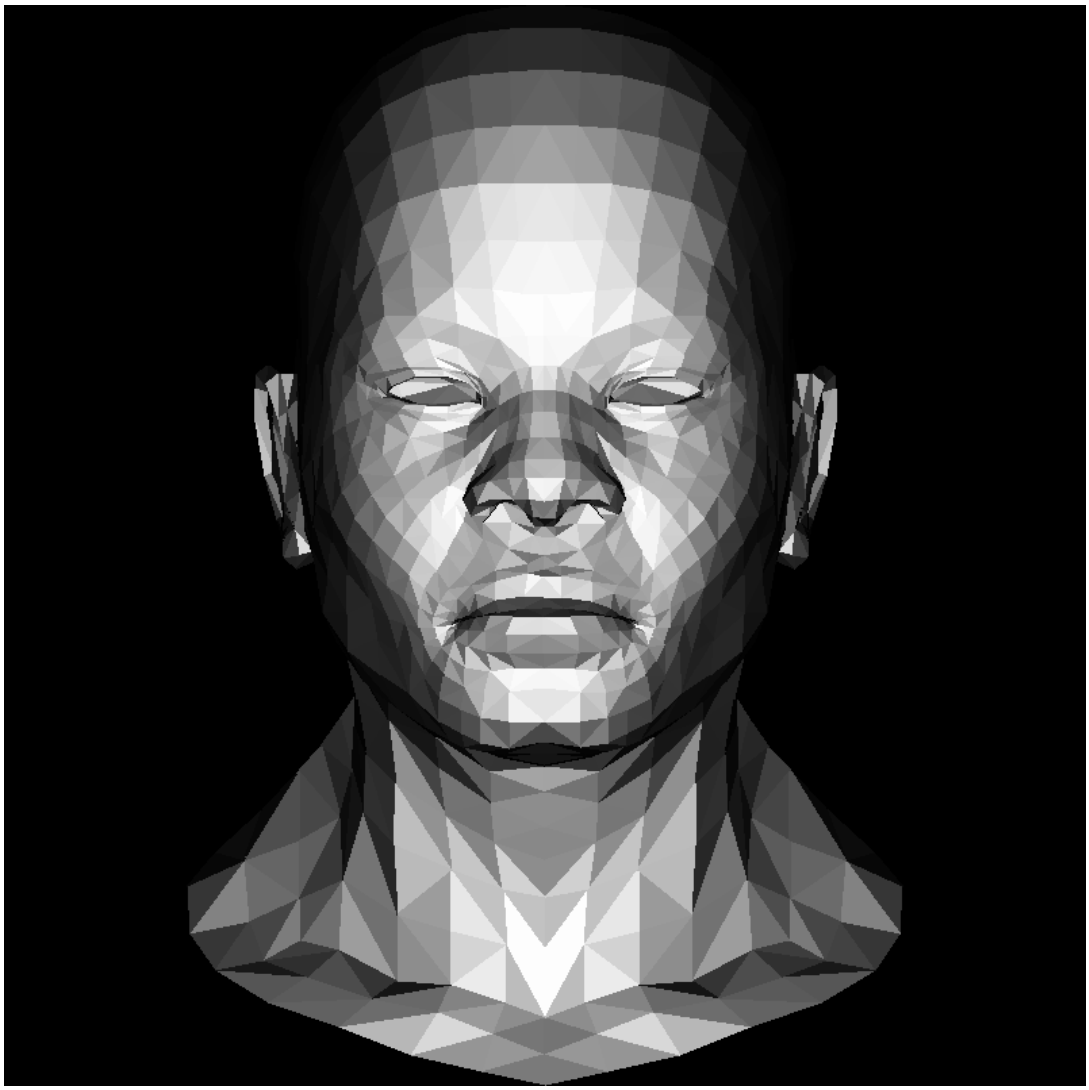
```
triangle(screen_coords, float *zbuffer, image, TGAColor(intensity*255, intensity*255, intensity*255, 255));

[...]

void triangle(Vec3f *pts, float *zbuffer, TGAImage &image, TGAColor color)
{
```

```
    Vec2f bboxmin( std::numeric_limits<float>::max(), std::numeric_limits<float>::max());
    Vec2f bboxmax(-std::numeric_limits<float>::max(), -std::numeric_limits<float>::max());
    Vec2f 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.f,      std::min(bboxmin[j], pts[i][j]));
            bboxmax[j] = std::min(clamp[j], std::max(bboxmax[j], pts[i][j]));
        }
    }
    Vec3f 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[0], pts[1], pts[2], P);
            if (bc_screen.x<0 || bc_screen.y<0 || bc_screen.z<0) continue;
            P.z = 0;
            for (int i=0; i<3; i++) P.z += pts[i][2]*bc_screen[i];
            if (zbuffer[int(P.x+P.y*width)]<P.z) {
                zbuffer[int(P.x+P.y*width)] = P.z;
                image.set(P.x, P.y, color);
            }
        }
    }
}
```

对上一堂课的内容我们仅仅只是对源码做了这么一点微小的改变就忽略了隐藏面！渲染结果如下：



代码在这里

好了，我们仅仅只是想加入一个Z值，还有什么是我们能干的?

纹理！这就是我们的家庭作业了。

在.obj文件里面我们有用“U V”字母开头的行数据，它们就是贴图数据组。在刻线面“f x/x/x x/x/x x/x/x”在斜线之间的数字就是贴图在三角形的顶点数据。把它放进三角形内，乘以纹理图像的宽高就能获得渲染出来的颜色。

纹理数据可以从这里得到。

如下就是一个我提供给你的例子：