课1

课1：Bresenham线条画法

第一步

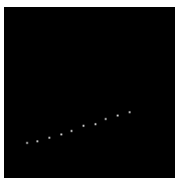
第一课的目标是渲染出一个线网模型。为了画出这个模型我们得学会如何去画一条线段。我们需要去简单了解Bresenham线条原理，以便于我们能够编写对应代码。画一条从（x0，y0）到（x1，y1)的线段的最简单代码应该长啥样？如下所示：

```
void line(int x0, int y0, int x1, int y1, TGAImage &image, TGAColor color)
{
    for (float t=0.; t<1.; t+=.01) {
        int x = x0 + (x1-x0)*t;
        int y = y0 + (y1-y0)*t;
        image.set(x, y, color);
    }
}
```

你能在这里找到对应的源代码

第二步

这份代码的问题（除了效率低之外）在于结束条件的选择，在上面我用的是0.01，如果我们将它改成0.1，我们画出的线条就会像这样：

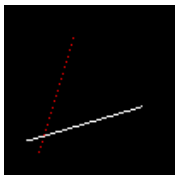


我们可以很容易找到关键的步骤，那就是我们需要画多少像素。最简单的代码如下所示（但它是错误的）

```
void line(int x0, int y0, int x1, int y1, TGAImage &image, TGAColor color)
{
    for (int x=x0; x<=x1; x++) {
        float t = (x-x0)/(float)(x1-x0);
        int y = y0*(1.-t) + y1*t;
        image.set(x, y, color);
    }
}
```

注意！在这个源代码中的第一个错误就是整数的除法，就像（x – x0) / (y – y0)。如果我们尝试画如下代码所示的线段：

```
line(13, 20, 80, 40, image, white);
line(20, 13, 40, 80, image, red);
line(80, 40, 13, 20, image, red);
```



如上图所示，第一条线段是完好的，第二条线段变成了断续的，第三条线段没有了。造成这样的结果的原因是在这份代码中，我们用了不同的颜色和不同的方向但其他条件一样的线段来表示第一和第三条线段。我们能看见完好的白色那条线段。我也希望能够将线段颜色改为红色，但做不到。这是一个对称性的测试：画一条线段的结果和它填充像素点的方向无关，即（a，b）方向的线段和（b，a）方向的线段表现效果一致。

第三步

我们将在代码中使得X0永远小于X1来修复丢失的红色线段。
之所以会有这些空隙在在其中一个线段上是因为它的高度要大于宽度，以下是我建议的解决方法：

```
if (dx>dy) {for (int x)} else {for (int y)}
```

我们会得到如下结果：

```
void line(int x0, int y0, int x1, int y1, TGAImage &image, TGAColor color)
{
    bool steep = false;
    if (std::abs(x0-x1)<std::abs(y0-y1)) { // if the line is steep, we
transpose the image
        std::swap(x0, y0);
        std::swap(x1, y1);
        steep = true;
    }
    if (x0>x1) { // make it left-to-right
        std::swap(x0, x1);
        std::swap(y0, y1);
    }
    for (int x=x0; x<=x1; x++) {
        float t = (x-x0)/(float)(x1-x0);
        int y = y0*(1.-t) + y1*t;
        if (steep) {
            image.set(y, x, color); // if transposed, de-transpose
        } else {
            image.set(x, y, color);
        }
    }
}
```

Timings：第四步

Warning：编译器（g++ -O3）比我们更懂得怎么让编译速度加快，这是历史和文化的原因。这段代码运行的很好。而它的复杂度也是我想在最终版本或者我们的渲染器中看见的。它显然是低效的（类似于它其中的除法运算等），但是它足够短且容易阅读。值得注意的是它没有对边界进行检查也不便于维护这一点是不好的。在接下来的文章中我尽量不去重载这些需要被大量重复引用的代码。与此同时，我会隔三岔五的去提醒你检查的必要性。
所以，即便先前的代码运行的挺好，但我们还是得去尽自己的可能去优化它。优化是一件危险的事情。我们必须清楚的知道代码的每一步运行。为了渲染显卡还是只是为了CPU去进行代码优化是一件截然不同的事情。在开始或者在进行优化的时候。我们需要去分析代码。试图猜测哪种操作可能是导致渲染瓶颈的最大原因。
为了测试，我把我们之前的三条线段画了1000000次。我的CPU是Intel® Core(TM) i5-3450 CPU @ 3.10GHz. 对于每一个像素，上述代码都要调用一次TGAColor的拷贝构造函数。即1000000*3条线段*一条线段大概50个像素。如此之多的调用，不是么？我们该如何开始优化呢？分析工具会告诉我们。
我用g++ -ggdb -g -pg -O0 keys去编译代码。并用gprof进行分析：

```
%   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 69.16      2.95     2.95  3000000     0.00     0.00  line(int, int, int, int, TGAImage&, TGAColor)
 19.46      3.78     0.83 204000000     0.00     0.00  TGAImage::set(int, int, TGAColor)
  8.91      4.16     0.38 207000000     0.00     0.00  TGAColor::TGAColor(TGAColor const&)
  1.64      4.23     0.07        2    35.04    35.04  TGAColor::TGAColor(unsigned char, unsigned char, unsigned char, unsigned char)
  0.94      4.27     0.04                             TGAImage::get(int, int)
```

大约10%花费在复制颜色，但是有70%的时间花费在调用line()。这就是我们需要优化的地方。

第四步续接

我们应该记得每一个除法的除数都是一样的。让我们把它拿出循环。误差变量告诉我们从当前（x，y）像素到最佳直线的距离，每次误差大于一个像素时，我们将y值增大或者减小一个像素。对应的误差也随即减小一个像素。
代码你可以在这里获得：

```
void line(int x0, int y0, int x1, int y1, TGAImage &image, TGAColor color)
{
    bool steep = false;
    if (std::abs(x0-x1)<std::abs(y0-y1)) {
        std::swap(x0, y0);
        std::swap(x1, y1);
        steep = true;
    }
    if (x0>x1) {
        std::swap(x0, x1);
        std::swap(y0, y1);
    }
    int dx = x1-x0;
```

```
    int dy = y1-y0;
    float derror = std::abs(dy/float(dx));
    float error = 0;
    int y = y0;
    for (int x=x0; x<=x1; x++) {
        if (steep) {
            image.set(y, x, color);
        } else {
            image.set(x, y, color);
        }
        error += derror;
        if (error>.5) {
            y += (y1>y0?1:-1);
            error -= 1.;
        }
    }
}
```

运行上述代码，分析器的输出如下：

```
%   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 38.79      0.93     0.93  3000000     0.00     0.00  line(int, int, int, int, TGAImage&, TGAColor)
 37.54      1.83     0.90 204000000     0.00     0.00  TGAImage::set(int, int, TGAColor)
 19.60      2.30     0.47 204000000     0.00     0.00  TGAColor::TGAColor(int, int)
  2.09      2.35     0.05        2    25.03    25.03  TGAColor::TGAColor(unsigned char, unsigned char, unsigned char, unsigned char)
  1.25      2.38     0.03                             TGAImage::get(int, int)
```

Timings：第五也是最后一步

为什么我们需要浮点类型的点？唯一的原因就是为了使1/dx和在循环体里面比较0.5变得有意义。我们可以使用另一个误差变量来替换当前的误差变量以便抛弃掉浮点类型的数据。让我们叫它Error2并使它等于error*dx*2。这里是完整的代码

```
void line(int x0, int y0, int x1, int y1, TGAImage &image, TGAColor color)
{
    bool steep = false;
    if (std::abs(x0-x1)<std::abs(y0-y1)) {
        std::swap(x0, y0);
        std::swap(x1, y1);
        steep = true;
    }
    if (x0>x1) {
        std::swap(x0, x1);
        std::swap(y0, y1);
```

```
    }
    int dx = x1-x0;
    int dy = y1-y0;
    int derror2 = std::abs(dy)*2;
    int error2 = 0;
    int y = y0;
    for (int x=x0; x<=x1; x++) {
        if (steep) {
            image.set(y, x, color);
        } else {
            image.set(x, y, color);
        }
        error2 += derror2;
        if (error2 > dx) {
            y += (y1>y0?1:-1);
            error2 -= dx*2;
        }
    }
}
```

```
%   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 42.77      0.91     0.91 204000000     0.00     0.00  TGAImage::set(int, int, TGAColor)
 30.08      1.55     0.64  3000000     0.00     0.00  line(int, int, int, int, TGAImage&, TGAColor)
 21.62      2.01     0.46 204000000     0.00     0.00  TGAColor::TGAColor(int, int)
  1.88      2.05     0.04        2    20.02    20.02  TGAColor::TGAColor(unsigned char, unsigned char, unsigned char, unsigned char)
```

现在，已经移除了足够多的拷贝复制操作在通过引用传递颜色的函数调用中（或者仅仅是使能编译flag -O3），现在在代码已经没有无意义的乘法或者除法了。运行时间也从2.95秒降到了0.64秒。
我也欢迎你到这里去讨论这些问题，优化是复杂的。

线框渲染

我们现在已经可以去渲染一个网格了。你可以在这里找到源代码和测试模型。我用wavefront obj文件格式来存储这个模型。我们需要做的事情就是从文件中读取顶点数组，就像如下所示的一样：

```
v 0.608654 -0.568839 -0.416318
```

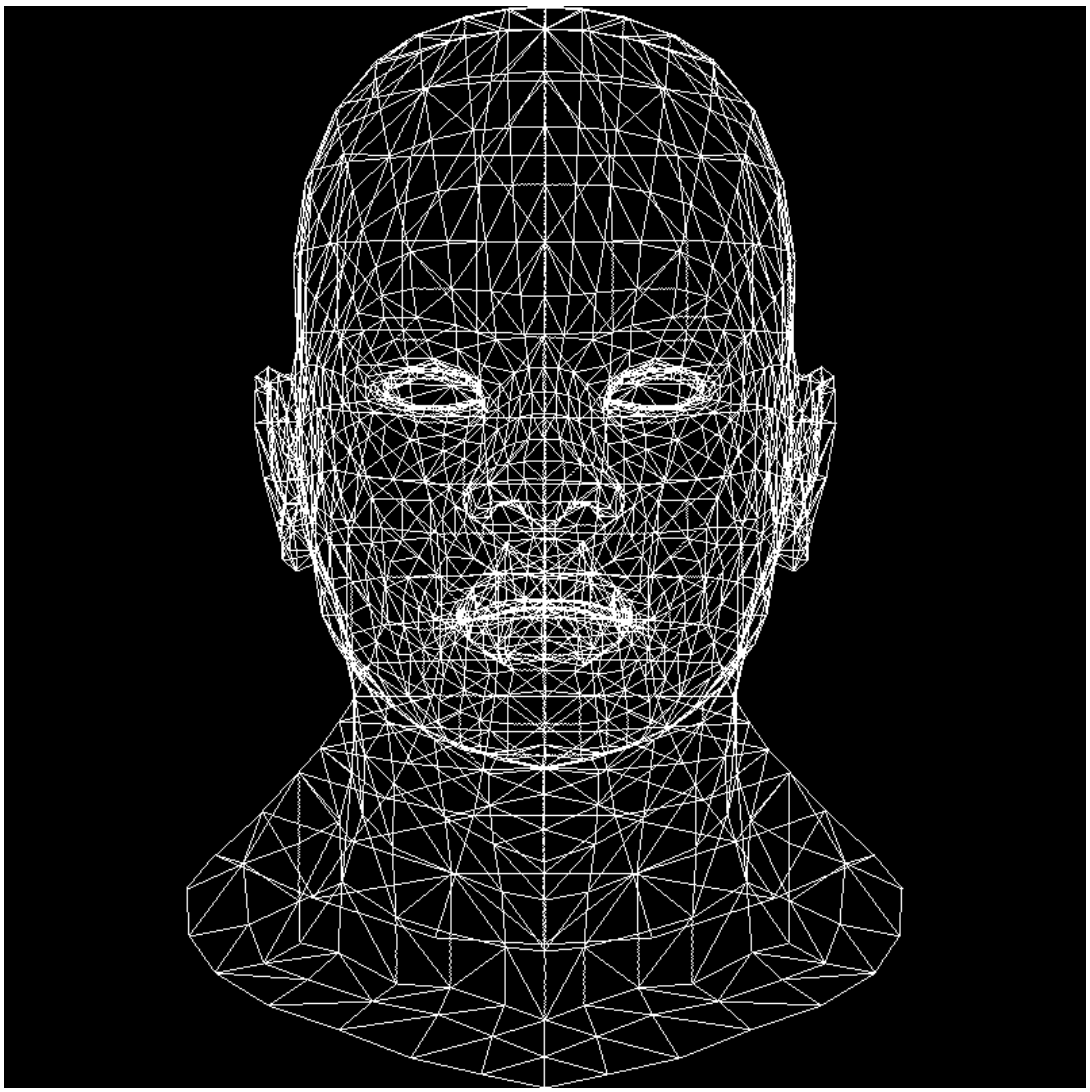
这是在每个文件的线面上一个顶点的x，y，z轴上的坐标

```
f 1193/1240/1193 1180/1227/1180 1179/1226/1179
```

我们应该关注每一个空格的第一个数字。这就是我们之前从文件里面读取的数组中顶点的数量。所以，这一行说的是一个三角形由1193，1180，1179个顶点组成。要记住的是，在一个

obj文件里索引的开始下标是1，这意味着你得分别寻找1192，1180，1179个顶点。模型的.cpp文件包括了一个简单的解析器。将下方的循环结构添入我们的main.cpp文件并编译，我们的网格渲染就准备好了。

```
for (int i=0; i<model->nfaces(); i++) {
    std::vector<int> face = model->face(i);
    for (int j=0; j<3; j++) {
        Vec3f v0 = model->vert(face[j]);
        Vec3f v1 = model->vert(face[(j+1)%3]);
        int x0 = (v0.x+1.)*width/2.;
        int y0 = (v0.y+1.)*height/2.;
        int x1 = (v1.x+1.)*width/2.;
        int y1 = (v1.y+1.)*height/2.;
        line(x0, y0, x1, y1, image, white);
    }
}
```



下一堂课我们将画2d界面上的三角形，并继续提高我们的渲染效果。