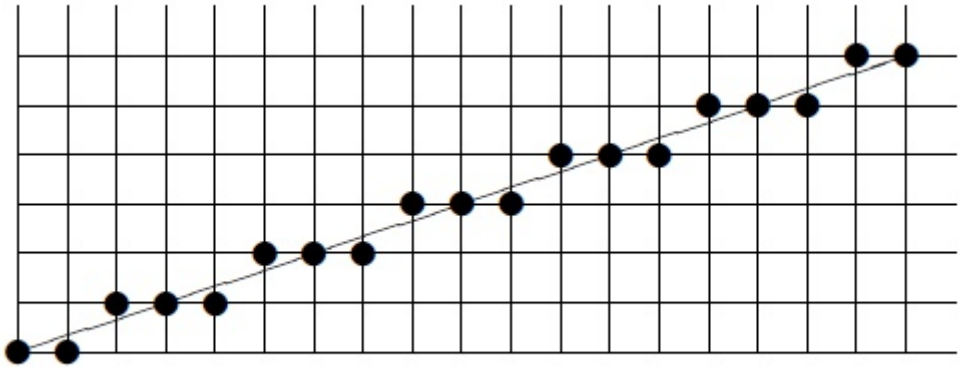
Bresenham直线算法

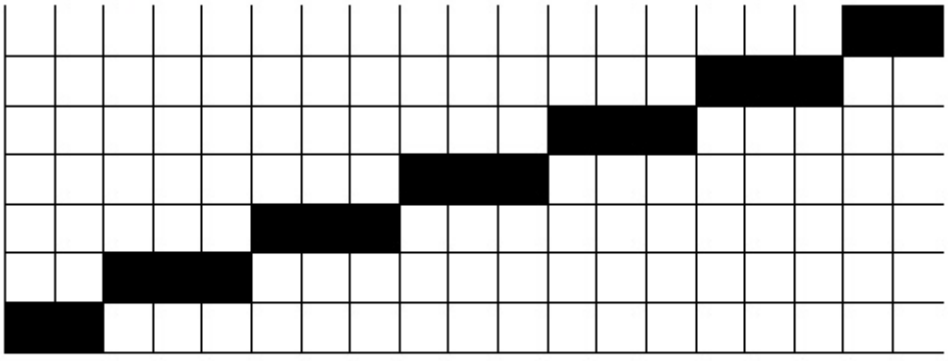
卖坷垃金...
骚话连篇连篇骚话

Bresenham直线算法是图形学中的经典画直线的算法。

真实的直线是连续的，但是计算机显示的精度有限，不可能真正显示连续的直线，于是在计算机中我们用一系列离散化后的点（像素）来近似表现这条直线，如下图所示。



(a).实际要求的直线及其近似点



(b).离散化后用像素点表示的的直线

选自《计算机图形学的概念与方法》柳朝阳，郑州大学数学系，侵删

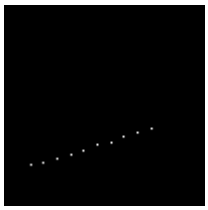
在本文中我们实现一个简单的直线算法，并逐步优化，最终引出经典的Bresenham直线算法，并详细讲解Bresenham直线算法的步骤和原理。

好的， 我们可以开始实现第一个最简单版本的直线算法。为方便讲解，首先我们规定我们的坐标系**以左下为原点，横轴为x, 纵轴为y。**

首先第一个版本我们将直线分成10份, 每一份画一个像素:

```
#include "tgaimage.h"

const TGAColor white = TGAColor(255, 255, 255, 255);
const TGAColor red   = TGAColor(255, 0,   0,   255);

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);
    }
}

int main(){
    TGAImage image(100, 100, TGAImage::RGB);
    line(13, 20, 80, 40, image, white);
    image.flip_vertically();
    image.write_tga_file("output.tga");
}
```

自然, 这个效果肯定好不到哪去:



我们发现这条线中间有很多的空洞，他并不是连续的。

在这份代码中, 至于TGAImage、TGAColor及相关方法是如何实现的, 可以在文末参考资料或者我自己的实现的代码github链接中找到, 相关部分大家完全可以当作伪代码来阅读, 不影响理解。

很简单的一个改进的思路就是，我们可以按x轴逐像素进行绘制。

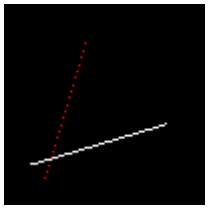
我们知道，两点确定一条直线，所以这个函数需要接受两个点(x0, y0), (x1, y1) 。我们沿着x轴，逐一填充每一个直线上的点。而点的y值可以根据x计算而来。根据等比原则我们可以得到:

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

```
#include "tgaimage.h"

const TGAColor white = TGAColor(255, 255, 255, 255);
```

```
const TGAColor red   = TGAColor(255, 0,   0,   255);

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);
    }
}

int main(){
    TGAImage image(100, 100, TGAImage::RGB);
    line(13, 20, 80, 40, image, white);
    line(20, 13, 40, 80, image, red);
    line(80, 40, 13, 20, image, red);
    image.flip_vertically();
    image.write_tga_file("output.tga");
}
```

结果如下：



我们发现line(80, 40, 13, 20, image, red); 对应的直线消失了， 红色的线也出现了空洞。仔细分析代码我们发现，**消失的线是因为在for循环中x的取值是从x0到x1， 如果x0大于x1的话，for循环没有执行。红色的线出现空洞是因为直线上的点在y轴上的增长速率比x轴上的大**，理想情况下，直线上存在多个x取值相同的点。而在x轴上逐像素画点的方法使得直线上所有像素点的x取值都不同。

但是呢，这种方法在直线斜率取值为0-1之间的时候效果是完美的，我们可以考虑把其他情况转换为直线斜率的绝对值取值在0-1之间的情况：

（1） 直线斜率 $|dy/dx| \in (1, +\infty)$

我们可以x，y轴互换，使其变成斜率在0-1之间的情况：

```
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 i
        std::swap(x0, y0);
```

```
        std::swap(x1, y1);
        steep = true;
    }
    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);
        }
    }
}
```

目前斜率>0的情况我们都可以处理了。

（2）直线终点x值小于起点

我们可以将起止点位置互换，则转换成了我们可以处理的情况了：

```
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 i
        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);
        }
    }
}
```

测试一下：

我们已经能够得到想要的结果了。但是我们回头看一下我们的代码，我们使用了多次浮点运算，TGAColor存在复制构造，这都使得我们代码的效率降低。

我们可以把"float t = (x-x0)/(float)(x1-x0); "的分母放到循环外面，以避免多次进行乘除法运算：

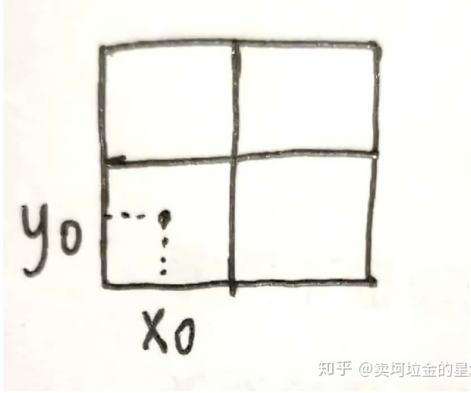
```
void line(int x0, int y0, int x1, int y1, TGAImage &image, TGAColor &color) {
    bool steep = false;
    if (std::abs(x0-x1)<std::abs(y0-y1)) {
        std::swap(x0, y0);
        std::swap(x1, y1);
        steep = true;
    }
    if (x0>x1) {
        std::swap(x0, x1);
        std::swap(y0, y1);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

这里我们详细解释一下：

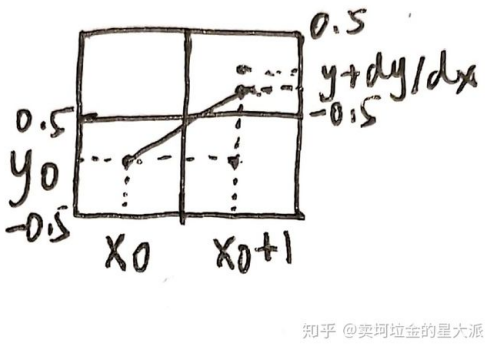
```
if (error>.5) {
    y += (y1>y0?1:-1);
    error -= 1.;
}
```

这一步是怎么来的。

我们在画图时，其实默认的像素点的准确位置是在网格的中点：



当x逐像素增长时，y的坐标值也随之增长，x轴每前进一个像素，y轴前进dy/dx像素。error可以认为是当y轴累计的前进值。当该值大于0.5时，我们可以认为当前像素点已经进入了屏幕像素矩阵的下一“行”，于是绘制的y值+=1。当网格中心点的y值定义为0时，y在一个网格的取值范围为[-0.5, 0.5)，在进入下一个网格后，y值更新为在当前网格的值，故需要使error-=1.



最后，dy/dx的分母dx是固定的，我们可以使用整形操作来代替浮点操作：

```
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;
        }
    }
}
```

最后再做一点点优化，比如添加边界控制，使用unsigned int替代int避免负数，用Eigen vector替代整形传值：

```
#include <eigen3/Eigen/Dense>
#include "tgaimage.h"

using uint = unsigned int;
using Point = Eigen::Matrix<uint, 2, 1>;

const TGAColor white = TGAColor(255, 255, 255, 255);
const TGAColor red   = TGAColor(255, 0,   0,   255);

bool is_in_bound(TGAImage &image, const Point &p){
    uint width = image.get_width();
    uint height = image.get_height();
    auto ans = p.x() < width && p.y() < height ? true : false;
    return ans;
}

void line(const Point &p_start, const Point &p_end, TGAImage &image, const TGAColor &c
    uint x0, x1, y0, y1;
    x0 = p_start.x();
```

```
    y0 = p_start.y();
    x1 = p_end.x();
    y1 = p_end.y();
    if(!is_in_bound(image, p_start) || !is_in_bound(image, p_end)){
        fprintf(stderr, "Point out of bound");
        std::exit(-1);
    }
    bool flip = false;
    if(std::abs((int)(y1-y0)) > std::abs((int)(x1-x0))){
        std::swap(x0, y0);
        std::swap(x1, y1);
        flip = true;
    }
    if(x0 > x1){
        std::swap(x0, x1);
        std::swap(y0, y1);
    }
    int dy = y1 - y0;
    int dx = x1 - x0;
    int y = y0;
    int error = 0;
    int derror = 2 * std::abs((int)dy);
    for (int x=x0; x<=x1; x++) {
        flip?image.set(y, x, color):image.set(x, y, color);
        error += derror;
        if(error > dx){
            error -= 2 * dx;
            y += (y1>y0?1:-1);
        }
    }
}
```

参考资料:

https://github.com/ssloy/tinyrenderer/wiki/Lesson-1-Bresenham%E2%80%99s-Line-Drawing-...
github.com/ssloy/tinyrenderer/wiki/Lesson-1-Bresenh...

编辑于 2021-11-19 23:47

计算机 算法 计算机图形学