实验2：直线的生成

姓 名： 曹辰鹏

学 号： 2019218117

班 级： 计科19-4

实验地点： 第三机房

实验时间： 2021.4.6

1. 实验目的和要求

理解直线生成的原理；掌握典型直线生成算法；掌握步处理、分析实验数据的能力；

编程实现DDA算法、Bresenham中点算法；对于给定起点和终点的直线，分别调用DDA算法和Bresenham中点算法进行批量绘制，并记录两种算法的绘制时间；利用excel等数据分析软件，将试验结果编制成表格，并绘制折线图比较两种算法的性能。

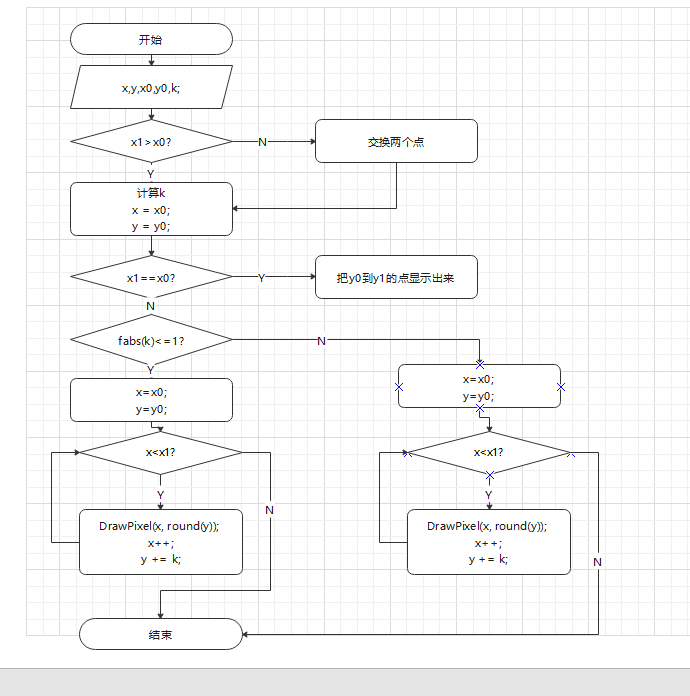
1. 实验环境和工具

本试验提供自带实验平台

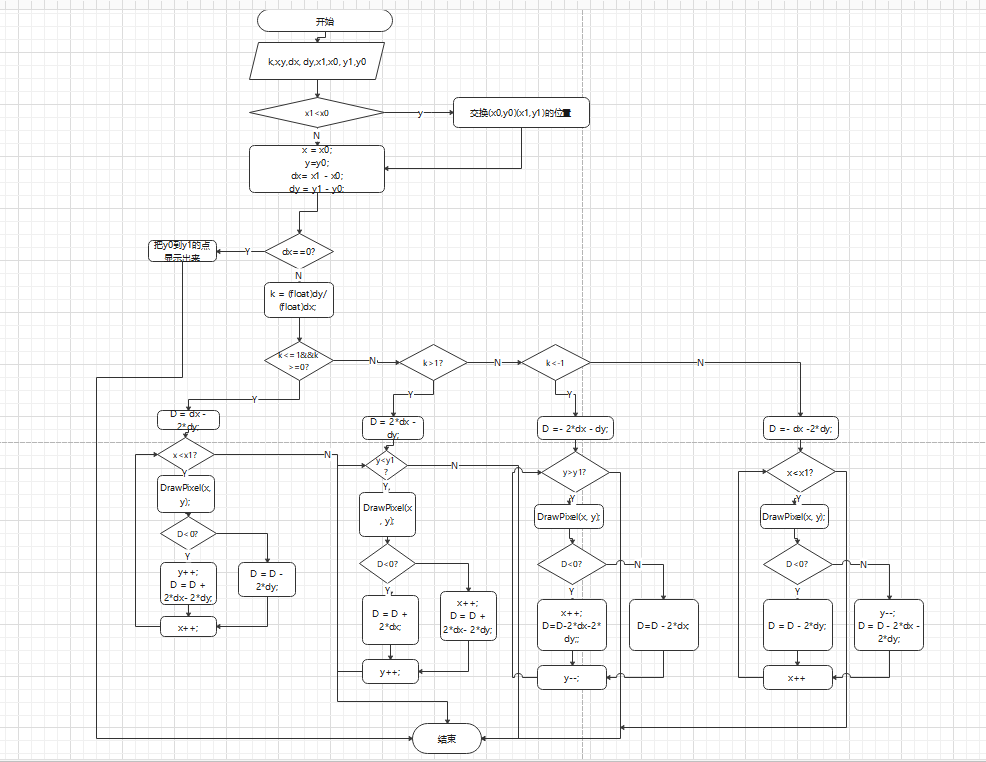
* 开发环境：Visual C++ 6.0
* 实验平台：Illumination（自制平台）

1. 实验结果
   1. 程序流程图

DDA:



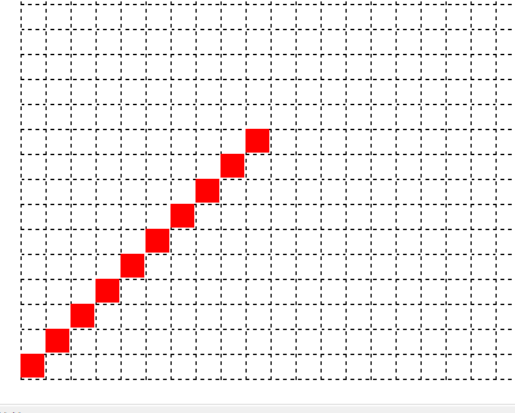
Bresenham中点算法：



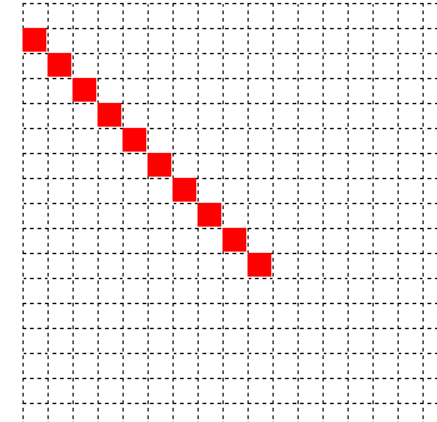
* 1. 程序代码
  2. 运行结果

DDA:

（0,0）（0,10）

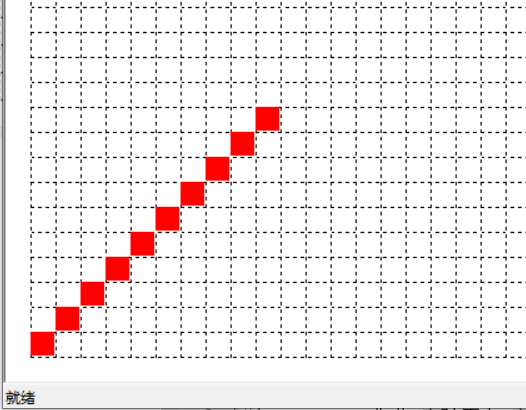


（0,20）（0,10）

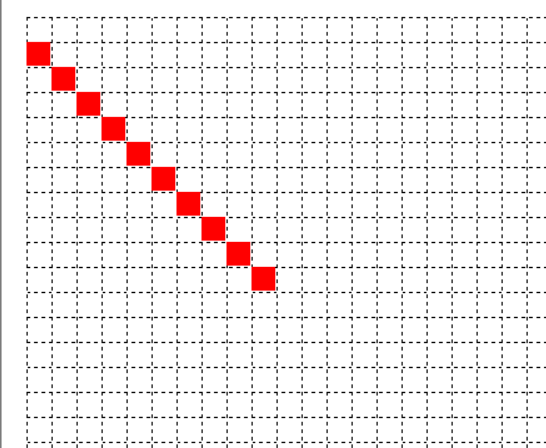


Bresenham中点算法：

（0,0）（0,10）



（0,20）（0,10）



* 1. 运行结果分析

我们其实发现这两种算法的计算结果是一样的。但是我们可以发现由于DDA算法中涉及到了浮点数的运算，时间上是稍微慢一点的。DDA算法的算法更加直观，很容易实现，需要考虑的时间也有点短。而Bresenham中点算法是DDA算法基于统一思想的改进版，这个也是利用直线的关系来计算的，但是它将浮点运算转化成了整数运算，运算速度大幅度提高。

1. 思考题（可选）

我们可以在程序接开头导入time.h文件，在那个计算直线之前，写上代码 clock\_t start,finish; double totaltime; start=clock();。 在算法结束之前，再写上finish=clock();  
   totaltime=(double)(finish-start)/CLOCKS\_PER\_SEC;。

可以将最后的结果写入到一个文件中。我们只要在程序结束之后，看那个文件输出的结果就行。

1. 实验心得

在这个试验中我学会了DDA算法，这个算法计算的y值每次都是不断的更新（我们以k>0且k<=1举例，下同）。更新的时候，每次我们的y都要加上k，每次还要根据更新的y值不断地判断是否原来的y加1。算法直观简单，容易实现。但是这个算法是用浮点数进行加减的，每次还要对y进行舍入求整，比例与硬件的实现。

中点Bresenham算法是在原来DDA的算法基础上发展来的。两个的基础思想是一样的，都是找到一个判定条件来判定直线周围的点的变更。每个点的变更都是根据周围的点来决定的。不过这个算法不用计算浮点数的加减乘除运算，所以有一定的进步，对于计算速度有了提高。判断的条件也发生了变化，是根据直线的性质变化的，根据不同的情况（直线的斜率和0的比较和1的比较）计算出D0。再计算出Di来判断Di+1该是什么样子，根据Di的结果判断y值是否加一。这个算法的比DDA算法精巧之处就在于这个增量的计算。对于每一列，只要检查一个误差项的符号就可以确定该列涂色像素所在的位置。

这个实验让我对这两个算法有了更深刻的认识。特别是，老师上课只是讲了对于上边所述的那一种情况。但是在做实验的时候我们需要考虑k不同的情况，还需要考虑x1<x0的情况。对于后者，我们可以直接将两个点的变量进行一个互换，达到最后我们想要的x1>x0。但是对于不同的k，DDA只用分k>1和k<1就行了，对于k<0的情况我发现是可以归结到k>0的情况中的。但是对于中点Bresenham算法我却是对k和领的比较与k和1的比较进行了划分，划分出来四种情况，最后计算出我们的结果。

实验5：光照模型

姓 名： 曹辰鹏

学 号： 2019218117

班 级： 计科19-4

实验地点： 第三机房

实验时间： 2021.4.6

1. 实验目的和要求

理解和掌握简单光照模型的基本原理和方法；并编程实现两种常用的明暗处理方法：

* Gouraud明暗处理方法
* Phong明暗处理方法

说明：本平台目前仅考虑环境光(Ambient Light)、漫反射光(Diffuse light)，暂不考虑镜面反射光(Specular Light ),而且不考虑光强衰减；

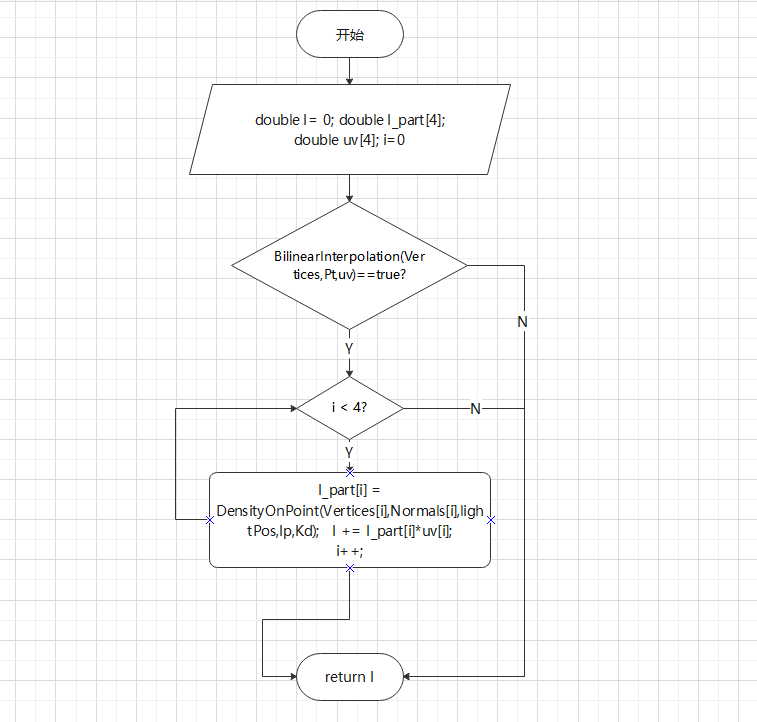
1. 实验环境和工具

本试验提供自带实验平台

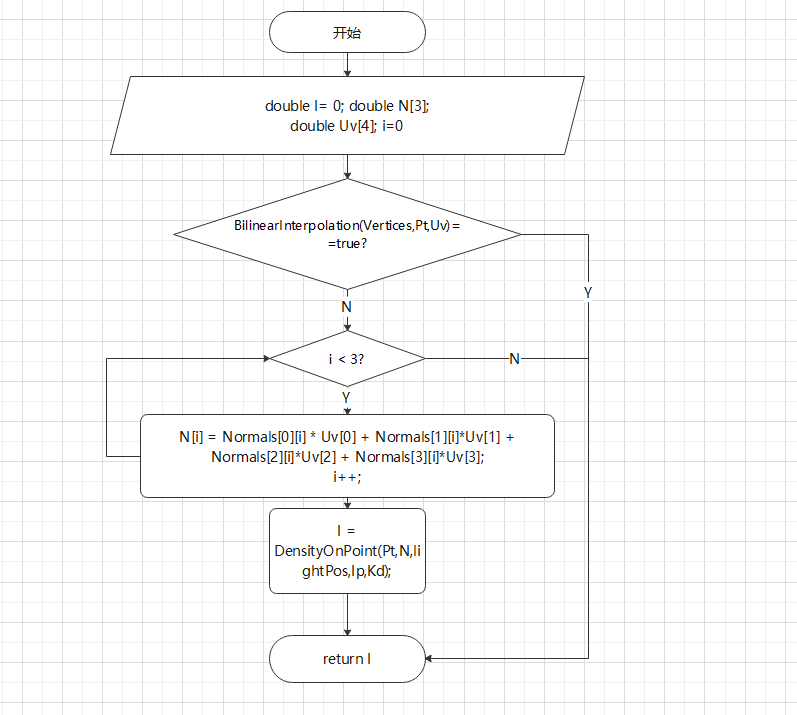
* 开发环境：Visual C++ 6.0
* 实验平台：Illumination（自制平台）

1. 实验结果
   1. 程序流程图

Gouraud算法：

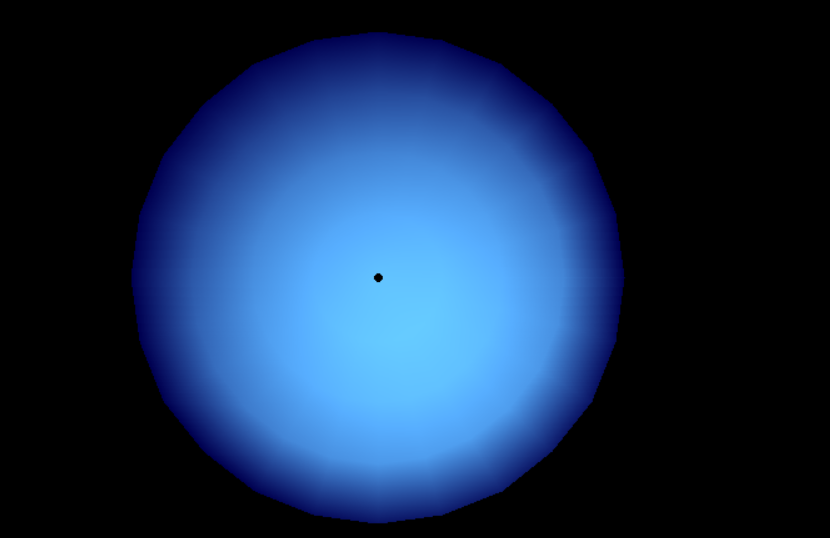


Phong算法：

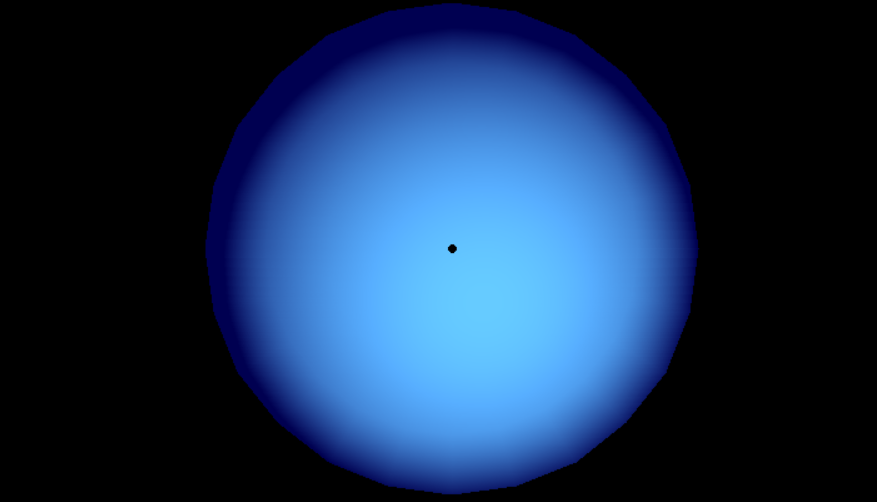


* 1. 程序代码
  2. 运行结果

Gouraud算法：



Phong算法



* 1. 运行结果分析

根据运行结果，我们可以看出Gourand算法算出来的结果在真实感上较FlatShading有了很大的提高。但马赫带效应依然存在。

Phong算法算出来的结果很清楚地发现这整个过度很自然，能很好地模拟高光，但是计算量也是一个问题。

1. 思考题（可选）

* 如何实现光强衰减效果

答：光照的衰减效果是和距离有关的。关照强度按照因子1/d^2的关系排列。但是这个当d较小时，会发生较大的变化。所以这个系数的变化可以用函数f(x) = min(1,1/(C0 + C1\*d + C2\*d^2))来表示。

* 如何实现镜面反射光效果

答：镜面反射的光强公式为I = Ip\*Ks\*(cos(a))^n，Ip为光源的强度。Ks表示为镜面反射系数，a表示的是试点方向V和镜面反射角的R之间的夹角，n表示的是与物体表面光滑度有关的一个系数。表面越光滑n越大。

1. 实验心得

在这个实验中，我学会了光强的计算方法，两个计算光强的方式都有他们的优点和缺点，但是我们都可以发现线性插值法贯穿其中。现在只是一个光源的，当我们有多个光源的时候，我们需要对除了环境光外的其他类型的光进行叠加。Ground算法计算简单快捷，计算出来的图形在真实感上相对于FlatShading有了很大的提高。但是我们看计算的结果依旧可以发现马赫带效应依然存在，也不能正确地模拟高光。这种算法也是只是考虑了漫反射，而对镜面反射效果不太理想，主要表现在高光域的形状不太规整。高光域只能在定点周围形成，不能在多边形域内形成。我们只能尽可能的细分多边形来尽量减少这个问题。

Phong算法利用线性插值得到多边形内各点的法向量，再根据这一点的法向量计算光强。法向量插值结果精确，真实感强。这样的算法的优点是过渡自然，能很好的模拟高光。但是计算量也随之上升。

这两种方法各有千秋，因为插值方式不一样，一个是对光强插值，一个是对向量插值，表现出来的效果也不一样。这个结果令我再次感受到了两种算法的差异。这两种算法对于一般的显示图形形状来说已经够用了。但是如果我们要绘制更完美的真实感图形，这两个就有点不够用，我们需要更加精确的方法。

### 附录：

DDA算法：

void CExperiment\_Frame\_OneView::DDA(int x0, int y0, int x1, int y1)

{

float k, b;

//进行交换

if(x1<x0)

{

int t;

t = x0;

x0 = x1;

x1 = t;

t = y0;

y0 = y1;

y1 = t;

}

if(x1 == x0)

{

int y = y0;

while(y<=y1)

{

DrawPixel(x0, y);

y++;

}

return;

}

k=float(y1-y0)/float(x1-x0);

b=float(x1\*y0-x0\*y1)/float(x1-x0);

if(fabs(k)<=1)

{

int x;

float y;

x = x0;

y = y0;

while(x < x1)

{

DrawPixel(x, round(y)); //判断y和中间值的大小

x++;

y += k;

}

}

else

{

float x;

int y;

x = x0;

y = y0;

while(y < y1)

{

DrawPixel(round(x), y);

y++;

x += 1/k;

}

}

return;

}

中点Bresenham算法：

void CExperiment\_Frame\_OneView::Mid\_Bresenham(int x0, int y0, int x1, int y1)

{

int D;

int dx, dy;

int x, y;

float k;

if(x1<x0)

{

int t;

t = x0;

x0 = x1;

x1 = t;

t = y0;

y0 = y1;

y1 = t;

}

x = x0;

y = y0;

dx = x1 - x0;

dy = y1 - y0;

if(dx == 0)

{

while(y<=y1)

{

DrawPixel(x, y);

y++;

}

return;

}

k = (float)dy/(float)dx;

if(k<=1 && k>=0)

{

D = dx - 2\*dy;

while (x < x1)

{

DrawPixel(x, y);

if(D < 0)

{

y++;

D = D + 2\*dx - 2\*dy;

}

else

{

D = D - 2\*dy;

}

x++;

}

}

else if(k > 1)

{

D = 2\*dx - dy;

while (y < y1)

{

DrawPixel(x, y);

if(D > 0)

{

x++;

D = D + 2\*dx - 2\*dy;

}

else

{

D = D + 2\*dx;

}

y++;

}

}

else if( k < -1)

{

D = -2\*dx-dy;

while(y > y1)

{

DrawPixel(x, y);

if(D > 0)

{

D = D - 2\*dx;

}

else

{

x++;

D = D- 2\*dx - 2\*dy;

}

y--;

}

}

else

{

D = -dx - 2\*dy;

while(x < x1)

{

DrawPixel(x, y);

if (D < 0)

{

D = D - 2\*dy;

}

else

{

y--;

D = D - 2\*dx - 2\*dy;

}

x++;

}

}

return;

}

Gouraud算法

double CIlluminationView::Gouraud (int Pt[3],

int Vertices[4][3],

double Normals[4][3],

int lightPos[3],

double Ip,

double Kd)

{

double I = 0; //itensity;

double I\_part[4];

double uv[4];

if(!BilinearInterpolation(Vertices,Pt,uv))

{

return I;

}

for(int i = 0;i < 4;++i)

{

I\_part[i] = DensityOnPoint(Vertices[i],Normals[i],lightPos,Ip,Kd);

I += I\_part[i]\*uv[i];

}

return I;

}

Phong算法

double CIlluminationView::Phong(int Pt[3],

int Vertices[4][3],

double Normals[4][3],

int lightPos[3],

double Ip,

double Kd)

{

double I = 0; //itensity;

double Uv[4] = {0};

if(!BilinearInterpolation(Vertices,Pt,Uv))

{

return I;

}

double N[3];

for(int i = 0;i < 3;++i)

{

N[i] = Normals[0][i] \* Uv[0] + Normals[1][i]\*Uv[1] + Normals[2][i]\*Uv[2] + Normals[3][i]\*Uv[3];

}

I = DensityOnPoint(Pt,N,lightPos,Ip,Kd);

return I;

}