2016-1-13

GIS 图形算法

程序设计报告

何天乐 **1004135121**

程序运行环境

本程序基于.NET FRAMEWORK 4.5 框架，

1 直线生成算法

1.1 逐点比较法

逐点比较法绘制直线时，每绘制一个点就比较该点与原点连成的直线和原直线的偏差，根据偏差结果决定下一个点的绘制位置。

本程序中通过鼠标单击事件 mouse down 事件获取线段起点，mouse up 事件获取线段终点。算法输入包括起点、终点、位图，运行算法可以将直线绘制到位图上，然后通过 picturebox 控件将位图呈现在屏幕窗口上。下面为逐点比较算法程序示例：

```
public static void potLine(Point startPoint, Point endPoint, Color myColor,Bitmap mainBitmap)
        {
            int x0 = startPoint.X, y0 = startPoint.Y, x1 = endPoint.X, y1 = endPoint.Y;
            int x = x0, y = y0;
            int f = 0;     // 判别式变量
            int num = Math.Abs((x1 - x0)) + Math.Abs((y1 - y0));
            mainBitmap.SetPixel(x0, y0, myColor);
            if (x0 <= x1 && y0 <= y1)
            {
                while (num > 0)
                {
                    if (f >= 0)
                    {
                        x += 1;
                        f = f - Math.Abs(y1 - y0); //判别式递推公式
                    }
```

```
            else
            {
                y += 1;
                f = f + Math.Abs(x1 - x0);//判别式递推公式
            }
            mainBitmap.SetPixel(x, y, myColor);
            num--;
        }
    }
    else if (x0 <= x1 && y0 > y1)
    {
        while (num > 0)
        {
            if (f >= 0)
            {
                y -= 1;
                f = f - Math.Abs(x1 - x0);//判别式递推公式
            }
            else
            {
                x += 1;
                f = f + Math.Abs(y1 - y0);//判别式递推公式
            }
            mainBitmap.SetPixel(x, y, myColor);
            num--;
        }
    }
    else if (x0 > x1 && y0 > y1)//third
```

```
{
    while (num > 0)
    {
        if (f >= 0)
        {
            x -= 1;
            f = f - Math.Abs(y1 - y0);//判别式递推公式
        }
        else
        {
            y -= 1;
            f = f + Math.Abs(x1 - x0);//判别式递推公式
        }
        mainBitmap.SetPixel(x, y, myColor);
        num--;
    }
}
else if (x0 >= x1 && y0 < y1)
{
    while (num > 0)
    {
        if (f >= 0)
        {
            y += 1;
            f = f - Math.Abs(x1 - x0);//判别式递推公式
        }
        else
        {
```

```
                    x -= 1;
                    f = f + Math.Abs(y1 - y0);//判别式递推公式
                }
                mainBitmap.SetPixel(x, y, myColor);
                num--;
            }
        }
        while (num > 0)
        {
            if (f >= 0)
            {
                x += 1;
                f = f - Math.Abs(y1 - y0);//判别式递推公式
            }
            else
            {
                y -= 1;
                f = f + Math.Abs(x1 - x0);//判别式递推公式
            }
            mainBitmap.SetPixel(x, y, myColor);
            num--;
        }
    }
```

程序运行结果展示

单机“potline”按钮，然后鼠标在绘图窗口中操作触发 mouse down 和 mouse up 事件即可实现直线绘制。

1.2 DDA 数字微分仪算法

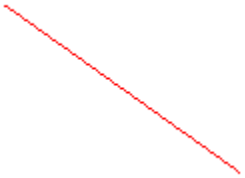
该算法基于微分方程求解，X 或 Y 方向的最大增量值为绘制点的步长，每一步各个方向的增量等于该方向的最大增量值除以步长，去一个计数器等于总共的步长，每绘制一步计数器减一，直到计数器为零停止绘制。此算法实现简单，但每一步都涉及复杂的浮点数运算，算法效率较低。

代码示例如下：

```
public static void ddaLine(Point startPoint, Point endPoint,Color myColor,Bitmap mainBitmap)
        {
            int x0 = startPoint.X, y0 = startPoint.Y, x1 = endPoint.X, y1 = endPoint.Y;
            mainBitmap.SetPixel(x0, y0, myColor);
            int dx = x1 - x0, dy = y1 - y0, steps, num;
            double xIncrement, yIncrement, x = x0, y = y0;
            if (Math.Abs(dx) > Math.Abs(dy))//计算步长
                steps = Math.Abs(dx);
            else
                steps = Math.Abs(dy);
            xIncrement = (double)dx / (double)steps;//X 方向每一步增量
            yIncrement = (double)dy / (double)steps;//Y 方向每一步增量
            for (num = 0; num < steps; num++)
            {
                x += xIncrement;
                y += yIncrement;
                mainBitmap.SetPixel((int)x, (int)y, myColor);
```

```
        }
    }
```

程序操作方式与前一算法相同，运行结果如下图所示：



1.3 Bresenham 算法

该算法也需要逐点绘制，绘制点时，增量最大方向上每次均走一步，另一方向上是否也增加取决于要绘制的点与元直线上的点的误差，选择使误差最小的点绘制。

```
public static void bresenhamLine(Point startPoint, Point endPoint,Color myColor,Bitmap mainBitmap)
    {
        try
        {
            int x0 = startPoint.X, y0 = startPoint.Y, x1 = endPoint.X, y1 = endPoint.Y;
            int dx = Math.Abs(x1 - x0), dy = Math.Abs(y1 - y0);
            double slope = (double)(y1 - y0) / (double)(x1 - x0);
            int p = 2 * dy - dx, doubleDy = 2 * dy, doubleDyMinusDx = 2 * (dy - dx), x, y;
            int doubleDx = 2 * dx, doubleDxMinusDy = 2 * (dx - dy);
            if (slope >= 0 && slope < 1)
            {
                if (x0 > x1)
                {
                    x = x1;
                    y = y1;
```

```
            x1 = x0;
        }
        else
        {
            x = x0;
            y = y0;
        }
        mainBitmap.SetPixel(x, y, myColor);

        while (x < x1)
        {
            x++;
            if (p < 0)
            {
                p += doubleDy;
            }
            else
            {
                y++;
                p += doubleDyMinusDx;
            }
            mainBitmap.SetPixel(x, y, myColor);
        }
    }
    if (slope >= -1 && slope < 0)
    {
        if (x0 > x1)
        {
```

```
            x = x1;
            y = y1;
            x1 = x0;
        }
        else
        {
            x = x0;
            y = y0;
        }
        mainBitmap.SetPixel(x, y, myColor);
        while (x < x1)
        {
            x++;
            if (p < 0)
            {
                p += doubleDy;

            }
            else
            {
                y--;
                p += doubleDyMinusDx;
            }
            mainBitmap.SetPixel(x, y, myColor);
        }
    }
    if (slope <= -1)
    {
```

```
if (y0 < y1)
{
	x = x1;
	y = y1;
	y1 = y0;
}
else
{
	x = x0;
	y = y0;
}
mainBitmap.SetPixel(x, y, myColor);

while (y > y1)
{
	y--;
	if (p < 0)
	{
		p += doubleDx;
	}
	else
	{
		x++;
		p += doubleDxMinusDy;
	}
	mainBitmap.SetPixel(x, y, myColor);
}
```

```
}
if (slope > 1)
{

    if (y0 > y1)
    {
        x = x1;
        y = y1;
        y1 = y0;
    }
    else
    {
        x = x0;
        y = y0;
    }
    mainBitmap.SetPixel(x, y, myColor);
    while (y < y1)
    {
        y++;
        if (p < 0)
        {
            p += doubleDx;
        }
        else
        {
            x++;
            p += doubleDxMinusDy;
        }
```

```
                    mainBitmap.SetPixel(x, y, myColor);
                }
            }
        }
        catch
        {
        }
    }
```

程序运行结果如下图所示：

2 圆生成算法（中点画圆法）

对于圆心在坐标原点的圆，可以只考虑画八分之一圆弧然后通过圆的对称性绘制出整个圆周，然后通过平移将圆的位置从坐标原点平移到实际圆心所在位置。

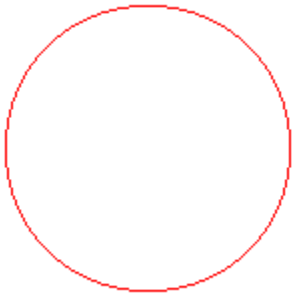
此圆生成算法中，对于每个要绘制的点，通过计算判断两个带选点的中点与圆周的位置决定将要绘制的点。

```
public static void drawCircle(Point startPoint,Point endPoint,Color myColor,Bitmap mainBitmap)
    {
        int radius = (int)Math.Sqrt((endPoint.X - startPoint.X) * (endPoint.X - startPoint.X) + (endPoint.Y - startPoint.Y) * (endPoint.Y - startPoint.Y));//计算半径
        int x = 0;
        int y = radius;
        double xMid;
        double yMid;
```

```
int xMax = (int)(((double)radius) / Math.Sqrt(2.0));  //八分之一圆弧 X 轴方向的步长

while (x <= xMax)
{
    try
    {
        mainBitmap.SetPixel(startPoint.X + x, startPoint.Y + y, myColor);
        mainBitmap.SetPixel(startPoint.X + x, startPoint.Y - y, myColor);
        mainBitmap.SetPixel(startPoint.X + y, startPoint.Y - x, myColor);
        mainBitmap.SetPixel(startPoint.X + y, startPoint.Y + x, myColor);
        mainBitmap.SetPixel(startPoint.X - x, startPoint.Y + y, myColor);
        mainBitmap.SetPixel(startPoint.X - x, startPoint.Y - y, myColor);
        mainBitmap.SetPixel(startPoint.X - y, startPoint.Y + x, myColor);
        mainBitmap.SetPixel(startPoint.X - y, startPoint.Y - x, myColor);
    }
    catch(Exception e)
    {
    }
    xMid = x;
    yMid = ((double)((2 * y - 1))) / 2.0;
    if (Math.Sqrt((xMid) * (xMid) + (yMid) * (yMid)) < radius)//判别式，判断中点与圆的位置关系
    {
    }
    else
    {
        y--;
    }
```

```
                x++;
            }
        }
```

程序输入为两个点，起点作为圆心，终点与起点之间的距离作为圆的半径，单机“画圆”按钮后，操作鼠标触发 mouse down 和 mouse up 事件选取起点和终点，程序运行结果如下图所示：



3 椭圆生成算法（中点画椭圆）

类似于圆生成算法，绘制椭圆时需要绘制四分之一椭圆周长。在椭圆斜率为 1 时将四分之一椭圆分成两个部分，分别绘制上下两个部分，每一步判断需要绘制的点的位置方法和绘制圆时相同。

（1）上部分：待选择的点（xi,yi）是已经绘制点的正右方点或者右下方点。用这两点的中点(xi+1,yi-0.5)和椭圆参数函数计算出来这一点和椭圆的关系。因此有判别式：

选择的标准和递推公式：

di<0,选择正右方点，且 di+1=di+b2(2xi+3)

di>=0，选择右下方点，此时，di+1=di+b2(2xi+3)+a2(-2yi+2)

初始时，x0=0，y0=b，代入则可得到

d0=F(1，（b-0.5）)=b2+a2(-b+0.25)

结束点(xi,yi)满足条件 2b2xi>=2a2yi

（2）下半部分，即 2b2xi >= 2a2yi

已选点(xi,yi)，选择下一点是在右下方或者正下方点中选择。

$$d_i = F(x_i + 0.5, y_i - 1) = b^2(x_i+0.5)^2 + a^2(y_i - 1)^2 - a^2b^2$$

则选择标准：

di<0,选择右下方点，且 di+1=di+b2(2xi+2)+a2(-2yi+3)

di>=0，选择正下方点，此时，di+1=di+a2(-2yi+3)

初始点处 d0=F(xi+0.5，yi-1)=b2(xi+0.5)2+a2(yi-1)2-a2b2

```
public static void drawEllipse(Point startPoint, Point endPoint, Color myColor, Bitmap mainBitmap)
        {
            try
            {
                int a = Math.Abs(endPoint.X - startPoint.X), b = Math.Abs(endPoint.Y - startPoint.Y);  //分别计算椭圆的长半轴和短半轴
                int x = 0;
                int y = b;
                double midPointX = x + 1;
                double midPointY = y - 0.5;
                int tempPoint = (int)(((double)a * (double)a) / (Math.Sqrt((double)a * (double)a + (double)b * (double)b)));        //计算椭圆斜率为一时的点
                while (x <= tempPoint)
                {
                    mainBitmap.SetPixel(startPoint.X + x, startPoint.Y + y, myColor);
                    mainBitmap.SetPixel(startPoint.X - x, startPoint.Y + y, myColor);
                    mainBitmap.SetPixel(startPoint.X + x, startPoint.Y - y, myColor);
                    mainBitmap.SetPixel(startPoint.X - x, startPoint.Y - y, myColor);
                    if ((b * b * midPointX * midPointX + a * a * midPointY * midPointY - a * a * b * b) >= 0)
                    {
                        y--;
```

```
                    }
                    x++;
                    midPointX = x + 1;
                    midPointY = y - 0.5;
                }
                midPointX = x + 0.5;
                midPointY = y - 1;
                while (y >= 0)
                {
                    mainBitmap.SetPixel(startPoint.X + x, startPoint.Y + y, myColor);
                    mainBitmap.SetPixel(startPoint.X - x, startPoint.Y + y, myColor);
                    mainBitmap.SetPixel(startPoint.X + x, startPoint.Y - y, myColor);
                    mainBitmap.SetPixel(startPoint.X - x, startPoint.Y - y, myColor);
                    if ((b * b * midPointX * midPointX + a * a * midPointY * midPointY -
a * a * b * b) < 0)
                    {
                        x++;
                    }
                    y--;
                    midPointX = x + 0.5;
                    midPointY = y - 1;
                }
            }
            catch (Exception e)
            { }
        }
```

程序输入同样通过捕捉鼠标 mouse down 和 mouse up 事件获取两个点，起点为椭圆中心点，起点和终点构成的长方形的宽为椭圆的短半轴，长方形的长为椭圆的长半轴。程序运行结果如下图所示：

4 多边形生成算法

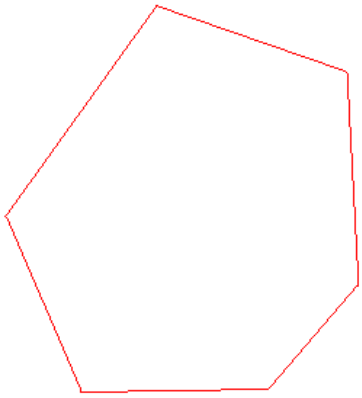
本程序中绘制多边形根据输入的多边形的顶点，依次将各个顶点连线即可绘制出需要的多边形。关键程序如下所示，鼠标单击事件取到的点全部存入一个活性链表中，然后从链表中依次取 2 个点绘制直线。

```
private void pictureBox1_MouseClick(object sender, MouseEventArgs e)
        {
            if (penType == "drawPolygon")
            {
                Geometry.polygonPointSets.Add(new Point(e.X, e.Y));
                if(Geometry.polygonPointSets.Count>=2)
                {
                    Geometry.drawLine(previewPoint, new Point(e.X, e.Y),myColor,mainBitmap);
                    pictureBox1.Refresh();
                }
                if (e.Button == MouseButtons.Right)
                {
                    Geometry.drawLine(Geometry.polygonPointSets[0], new Point(e.X, e.Y), myColor, mainBitmap);
                    pictureBox1.Refresh();
                    penType = "";
                }
```

```
                previewPoint.X = e.X;
                previewPoint.Y = e.Y;
            }
        }
```

程序操作：

单机“多边形”，鼠标左键依次单机画布取点，右键单机结束绘制。程序运行结果如下图所示：



5 多边形填充算法

本程序中多边形填充采用扫描先填充算法，对于多边形中的每一条变构造一个变表，对于每一个边表分别记录了该条边的 X、Y 坐标的最大和最小值，以及当 Y 最小时的 X 坐标和直线的斜率。

```
public class EdgeTable
    {
        public int Ymax;
        public int Ymin;
        public int Xmax;
        public int Xmin;
        public int X_ymin;//
```

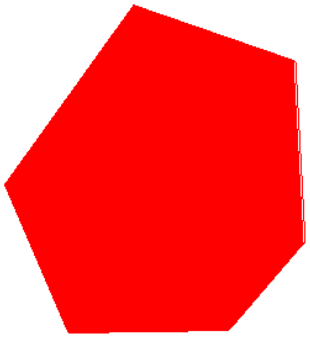
```
        public double slope;
        public EdgeTable(Point p1,Point p2)
        {
            if(p1.X>=p2.X)
            {
                Xmax = p1.X;
                Xmin = p2.X;
            }
            else
            {
                Xmax = p2.X;
                Xmin = p1.X;
            }
            if(p1.Y>=p2.Y)
            {
                Ymax = p1.Y;
                Ymin = p2.Y;
                X_ymin = p2.X;
            }
            else
            {
                Ymax = p2.Y;
                Ymin = p1.Y;
                X_ymin = p1.X;
            }
            slope = ((double)(p2.Y - p1.Y)) / ((double)(p2.X - p1.X));
        }
    }
```

填充算法程序输入为记录需要填充多边形的顶点的动态数组，然后依次去数组中的两个点构造变表，并存入一个变表动态数组中，之后按照变表的最小 Y 值对变表进行排序。最后扫描先，从 Y 的最小值开始依次扫描，如果发现与变表有交点则计算出交点并绘制该直线，直到扫描到 Y 的最大值，程序结束。

```
public static void fillPolygon(List<Point> pointSets,Color myColor,Bitmap mainBitmap)
        {
            List<EdgeTable> edgeTableSets = new List<EdgeTable>();
            if(pointSets.Count>=2)
            {
                for(int i=0;i<pointSets.Count;i++)
                {
                    try
                    {
                        edgeTableSets.Add(new  EdgeTable(pointSets[i],  pointSets[i  +
1]));
                    }
                    catch(Exception e)
                    {
                        edgeTableSets.Add(new EdgeTable(pointSets[i], pointSets[0]));
                    }
                }
            }
            edgeTableSets = getOrdered(edgeTableSets);
            int Ytop = edgeTableSets[0].Ymax;
            int Ybottom = edgeTableSets[edgeTableSets.Count - 1].Ymin;
            foreach (EdgeTable eg in edgeTableSets)
            {
                if(eg.Ymin<Ybottom)
```

```
                {
                    Ybottom = eg.Ymin;
                }
            }

            int tempY = Ybottom;
            while(tempY<Ytop)
            {
                List<Point> pSets = new List<Point>();
                foreach(EdgeTable eg in edgeTableSets)
                {
                    if((tempY>=eg.Ymin)&&(tempY<eg.Ymax))
                    {
                        pSets.Add(new Point((int)((tempY - eg.Ymin) / eg.slope) +
eg.X_ymin, tempY));
                    }
                }
                if(pSets.Count>=2)
                    Geometry.drawLine(pSets[0], pSets[1], myColor, mainBitmap);
                tempY++;
            }
        }
```

程序操作：在上一步绘制成功多边形后直接单机“填充多边形”按钮即可实现多边形填充，如下图所示。



6 线段裁剪算法

本程序实现了 CohenSutherland 线段裁剪算法，裁剪框为事先定义好的矩形。程序输入为需要裁剪直线的起点和终点，输出为重新绘制裁剪后的直线。

首先需要执行如下函数，对裁剪直线的起点和终点进行编码。

```
public static int CodeLeft = 1;

public static int CodeRight = 2;

public static int CodeButtom = 4;

public static int CodeTop = 8;
```

```
public static int Code(int x, int y)
        {
            int c = 0;
            if (x < left)
            {
                c = c | CodeLeft;
            }
            if (x > right)
```

```
        {
            c = c | CodeRight;
        }
        if (y < top)
        {
            c = c | CodeTop;
        }
        if (y > buttom)
        {
            c = c | CodeButtom;
        }
        return c;
    }
```

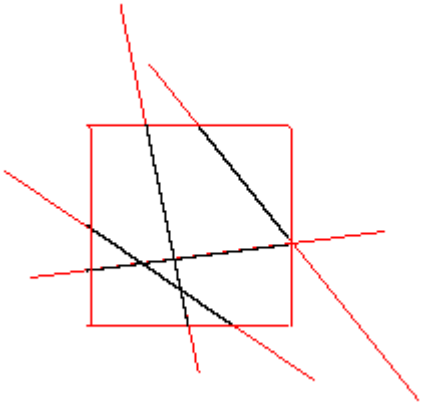
然后程序需要判断裁剪直线起点和终点分别与裁剪框之间的关系，然后确定需要重新绘制的直线部分，实现线段裁剪。

```
public static void CohenSutherland(int P1x, int P1y, int P2x, int P2y,Color myColor,Bitmap mainBitmap)
    {
        int C1 = Code(P1x, P1y), C2 = Code(P2x, P2y);
        int C;
        int Px = 0, Py = 0;//记录交点
        while (C1 != 0 || C2 != 0)//两个点（P1x,P1y）,（P2x,P2y）不都在矩形框内；都在内部就画出线段
        {
            if ((C1 & C2) != 0)     //两个点在矩形框的同一外侧 → 不可见
            {
                P1x = 0;
                P1y = 0;
```

```
        P2x = 0;
        P2y = 0;
        break;
    }
    C = C1;
    if (C1 == 0)// 判断 P1 P2 谁在矩形框内（可能是 P1，也可能是 P2）
    {
        C = C2;
    }

    if ((C & CodeLeft) != 0)//用与判断的点在左侧
    {
        Px = left;
        Py = P1y + (int)(Convert.ToDouble(P2y - P1y) / (P2x - P1x) * (left - P1x));
    }
    else if ((C & CodeRight) != 0)//用与判断的点在右侧
    {
        Px = right;
        Py = P1y + (int)(Convert.ToDouble(P2y - P1y) / (P2x - P1x) * (right - P1x));
    }
    else if ((C & CodeTop) != 0)//用与判断的点在上方
    {
        Py = top;
        Px = P1x + (int)(Convert.ToDouble(P2x - P1x) / (P2y - P1y) * (top - P1y));
    }
    else if ((C & CodeButtom) != 0)//用与判断的点在下方
    {
        Py = buttom;
```

```
                Px = P1x + (int)(Convert.ToDouble(P2x - P1x) / (P2y - P1y) * (buttom -
P1y));
            }

            if (C == C1) //上面判断使用的是哪个端点就替换该端点为新值
            {
                P1x = Px;
                P1y = Py;
                C1 = Code(P1x, P1y);
            }
            else
            {
                P2x = Px;
                P2y = Py;
                C2 = Code(P2x, P2y);
            }
        }
        Geometry.potLine(new Point(P1x, P1y), new Point(P2x, P2y), Color.Black,
mainBitmap);
    }
```

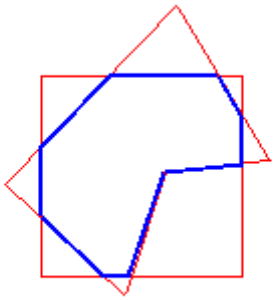
程序操作：首先单机“线段裁剪”按钮实现裁剪矩形框，然后任选一种画线算法绘制直线，然后再次单机“裁剪直线”按钮即可实现线段裁剪。程序运行结果如下图所示:



7.多边形裁剪

多边形裁剪采用 Sutherland_Hodgman 算法，鼠标单击“多边形裁剪”按钮屏幕出现裁剪框，然后单击“多边形”按钮绘制多边形，再次单击“多边形裁剪”按钮可裁剪出需要的多边形。

程序运行结果如下图所示：



8.自由曲线

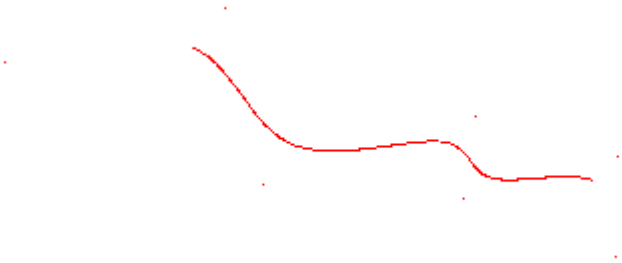
本程序中自由曲线的绘制采取三阶样条曲线算法，算法程序如下所示，输入为四个控制点，divisions 为需要插值点的个数。经过程序计算好的需要内插的点的坐标分别存储在 splinex 和 spliney 两个大数组中。

```
public static double[] splinex = new double[1001];

public static double[] spliney = new double[1001];

public static void bsp(point p1, point p2, point p3, point p4, int divisions)
```

```
{
    double[] a = new double[5];
    double[] b = new double[5];
    a[0] = (-p1.x + 3 * p2.x - 3 * p3.x + p4.x) / 6.0;
    a[1] = (3 * p1.x - 6 * p2.x + 3 * p3.x) / 6.0;
    a[2] = (-3 * p1.x + 3 * p3.x) / 6.0;
    a[3] = (p1.x + 4 * p2.x + p3.x) / 6.0;
    b[0] = (-p1.y + 3 * p2.y - 3 * p3.y + p4.y) / 6.0;
    b[1] = (3 * p1.y - 6 * p2.y + 3 * p3.y) / 6.0;
    b[2] = (-3 * p1.y + 3 * p3.y) / 6.0;
    b[3] = (p1.y + 4 * p2.y + p3.y) / 6.0;
    splinex[0] = a[3];
    spliney[0] = b[3];
    int i;
    for (i = 1; i <= divisions - 1; i++)
    {
        float t = System.Convert.ToSingle(i) / System.Convert.ToSingle(divisions);
        splinex[i] = (a[2] + t * (a[1] + t * a[0])) * t + a[3];
        spliney[i] = (b[2] + t * (b[1] + t * b[0])) * t + b[3];
    }
}
```

程序操作：鼠标单击“自由曲线”按钮，鼠标连续取直到第 4 个点时开始绘制曲线，然后继续取点可实现连续绘制样条曲线。

程序运行结果如下图所示：



9.二维图形变换

本程序选取固定好的三角形，可通过键盘事件实现对图形的二维操作。其中“W”“S”“A”“D”四个按键分别控制图形上下左右的平移；“Z”“X”分别控制图形的顺时针和逆时针旋转；“V”“B”分别控制图形的放大和缩小。

```
switch (e.KeyCode)
 {
                                case Keys.S:
                                    List<Point> p = new List<Point>();
                                    for (int i = 0; i < tranglePoints.Count; i++)
                                    {
                                        p.Add(new  Point(tranglePoints[i].X,  tranglePoints[i].Y  +
5));
                                    }
                                    tranglePoints = p;
                                    button11_Click(sender, e);
                                    break;
                                case Keys.D:
                                    List<Point> pLeft = new List<Point>();
                                    for (int i = 0; i < tranglePoints.Count; i++)
                                    {
                                        pLeft.Add(new       Point(tranglePoints[i].X       +       5,
```

```
tranglePoints[i].Y));
                    }
                    tranglePoints = pLeft;
                    button11_Click(sender, e);
                    break;
                case Keys.W:
                    List<Point> pUp = new List<Point>();
                    for (int i = 0; i < tranglePoints.Count; i++)
                    {
                        pUp.Add(new Point(tranglePoints[i].X, tranglePoints[i].Y -
5));
                    }
                    tranglePoints = pUp;
                    button11_Click(sender, e);
                    break;
                case Keys.A:
                    List<Point> pRight = new List<Point>();
                    for (int i = 0; i < tranglePoints.Count; i++)
                    {
                        pRight.Add(new Point(tranglePoints[i].X - 5,
tranglePoints[i].Y));
                    }
                    tranglePoints = pRight;
                    button11_Click(sender, e);
                    break;
                case Keys.Z:
                    List<Point> pClockwise = new List<Point>();
                    for (int i = 0; i < tranglePoints.Count; i++)
                    {
```

```
                    pClockwise.Add(new Point((int)(tranglePoints[i].X * Math.Cos(Math.PI / 36) - tranglePoints[i].Y * Math.Sin(Math.PI / 36)), (int)(tranglePoints[i].X * Math.Sin(Math.PI / 36) + tranglePoints[i].Y * Math.Cos(Math.PI / 36))));
                }
                tranglePoints = pClockwise;
                button11_Click(sender, e);
                break;
            case Keys.X:
                List<Point> pCounterclockwise = new List<Point>();
                for (int i = 0; i < tranglePoints.Count; i++)
                {
                    pCounterclockwise.Add(new Point((int)(tranglePoints[i].X * Math.Cos(-Math.PI / 36) - tranglePoints[i].Y * Math.Sin(-Math.PI / 36)), (int)(tranglePoints[i].X * Math.Sin(-Math.PI / 36) + tranglePoints[i].Y * Math.Cos(-Math.PI / 36))));
                }
                tranglePoints = pCounterclockwise;
                button11_Click(sender, e);
                break;
            case Keys.V:
                List<Point> pZoomin = new List<Point>();
                for (int i = 0; i < tranglePoints.Count; i++)
                {
                    pZoomin.Add(new Point((int)(tranglePoints[i].X * 1.1), (int)(tranglePoints[i].Y * 1.1)));
                }
                tranglePoints = pZoomin;
                button11_Click(sender, e);
                break;
            case Keys.B:
                List<Point> pZoomout = new List<Point>();
```

```
                    for (int i = 0; i < tranglePoints.Count; i++)
                    {
                        pZoomout.Add(new Point((int)(tranglePoints[i].X * 0.9),
(int)(tranglePoints[i].Y * 0.9)));
                    }
                    tranglePoints = pZoomout;
                    button11_Click(sender, e);
                    break;
                default:
                    break;
            }
        }
```

10.三维图形变换

本程序实现了对现有三维立方体图形的填充，且可以实现图形的三维自由旋转。

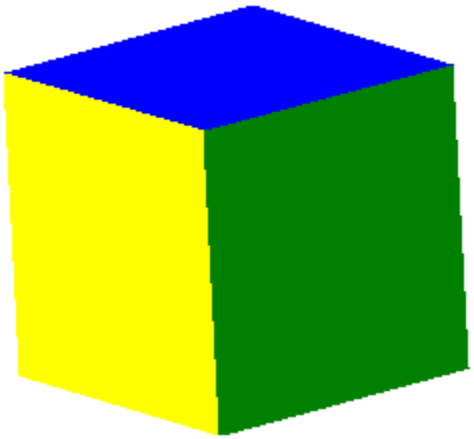
鼠标单击“3d变换”按钮屏幕自动绘制出一个立方体，且调用上文中使用到扫描线边表填充算法对立方体的各个面进行颜色填充，通过键盘事件”U”，”I“，“H”，“J”，“K”，“L”控制立方体的旋转。单击“自由变换”按钮立方体按照时间频率每隔50ms进行一次旋转变换。再次单击“自由变换”立方体停止转动。

立方体填充依据点光源投影方法，对各个表面的按照三维几何中心坐标的Z值大小进行排序，按照此顺序依次对各个表面进行填充。

下面为对立方体表面按照z值深度的排序算法：

```
public List<FillFace3D> getOrded(List<FillFace3D> fillFace)
{
List<FillFace3D> orderedFillingFace = new List<FillFace3D>();
if (fillFace.Count==6)
{
for (int j = 0; j < 6; j++)
{
FillFace3D f3 = fillFace[0];
for (int i = 0; i < fillFace.Count; i++)
{
if (fillFace[i].depth < f3.depth)
{
f3 = fillFace[i];
}
}
orderedFillingFace.Add(f3);
fillFace.Remove(f3);
}
}
return orderedFillingFace;
}
```

程序运行结果展示：



11.图像复制、剪贴、粘贴、功能

首先需要单机“编辑”--“选择”按钮，通过鼠标 mouse down 和 mouse up 事件拖动选取矩形框为要操作的区域，然后单机“复制”或者“剪贴”按钮

Clipboard.SetImage(mainBitmap.Clone(cloneZone, mainBitmap.PixelFormat));//复制

Clipboard.SetImage(mainBitmap.Clone(cloneZone, mainBitmap.PixelFormat));//剪贴

Graphics g = Graphics.FromImage(mainBitmap);

g.FillRectangle(Brushes.White, cloneZone);

pictureBox1.Refresh();

然后鼠标再在画布上单机选取一个点为要粘贴的位置，单机“粘贴”按钮即可将选取的图形部分粘贴到指定位置

相关代码如下所示：

Graphics g = Graphics.FromImage(mainBitmap);

g.DrawImage(Clipboard.GetImage(),startPoint);

pictureBox1.Refresh();

程序运行结果如下图所示：

11.图像保存功能的实现

本程序绘制的所有图像都是绘制在一个 Bitmap 对象之上，所以可以直接将 Bitmap 对象保存成 bmp 格式图片文件。相关程序如下所示。

private void sAVRToolStripMenuItem_Click(object sender, EventArgs e)

{

```
        if (besaved == false)
        {
            if (saveFileDialog1.ShowDialog(this) == DialogResult.OK)
            {
                string s = saveFileDialog1.FileName + ".bmp";
                mainBitmap.Save(s, System.Drawing.Imaging.ImageFormat.Bmp);
                besaved = true;
            }
        }
    }
```

附录：对本课程的建议

收先非常感谢王晓延老师一学期以来的付出，经过本学期图形学课程的学习本人掌握了一些有关基本图元的生成算法和相关图形学知识，非常感谢王老师。

我们专业开设 GIS 图形算法这门课旨在让大家掌握基本图形生成原理，能运用图形学知识处理相关问题。大多现有的经典的基本图元生成算法已经没有太多优化的余地，而且伴随着计算机性能的不断提高个人觉得已经没有必要花太多时间关注底层基本图元的生成过程，而应该更过的关注问题本身。怎样运用图形学的知识解决空间问题，或者三维可视化与建模、空间分析方面的应用等。希望老师可以教授一到两个案例，供学生们学习数学模型方法和相关图形生成技术，一便更为清楚的理解图形学知识在实际生产中的作用。

授课编程环境方面，基本图元生成算法练习仍可沿用.NET FRAMEWORK, 除此之外还希望老师能教授更多的一些图形函数库例如 GDI+或者 OpenGL 和 Direct3D 的使用方法，只需提供一到两个示例即可由学生自主查阅文档完成其他部分更深入的学习。

以上仅为个人见解供老师参考，尚有诸多不足之处。最后，衷心的希望 GIS 图形算法这门课程能够越办越好。