计算机图形学-基础算法

SA17006030 方远强

2017-11-12

目录

1. 实验要求

1.1. 图元的生成

直线、圆、椭圆、区域填充。

1.2. 样条曲线的生成

Bezier 曲线、B-样条曲线的生成。

1.3. 分形图形的生成

Koch 曲线、Mandelbrot 集、Julia 集、蕨类植物。

- 算法任选。不可调用已有函数（画点函数除外）。
- 最终提交源程序、可执行文件、实验报告电子版。
- 实验报告包括：基本原理、实现算法、实验结果。

2. 实验环境

本实验使用 Qt 5.4.2 和 Visual Studio 2013 编译完成。使用 Qt 完成交互式界面的编写，Visual Studio 2013 编译和运行程序。

实验中仅使用 Qt 定义好的画点函数 void QPainter::drawPoint (int x, int y)；实验中需要划线的时候，一律调用自己编写好的 BresenhamLine 函数。

3. 实验内容

3.1. 图形用户界面

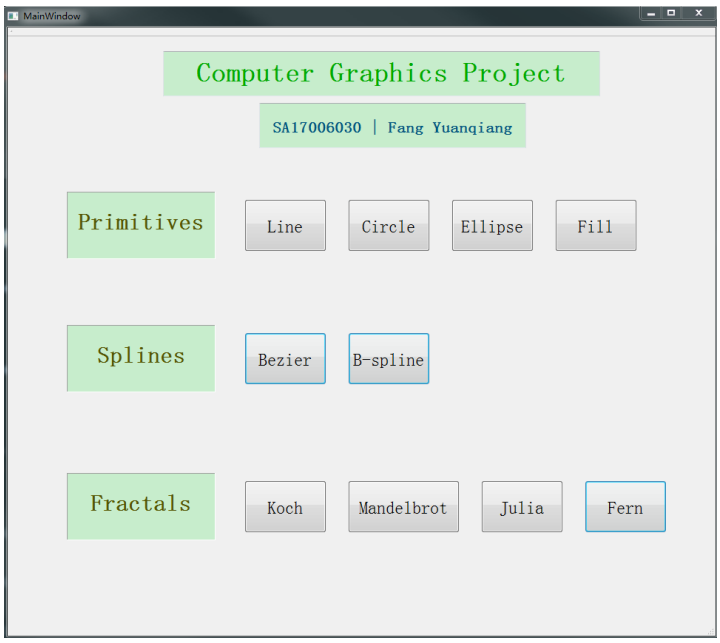


图 1. 图形用户界面

实验中的图形用户界面如图 1。主要分为 3 块：图元、样条曲线、分形图形。点击对应的按钮进入相应的画图窗口。

3.2. 图元生成

3.2.1. 直线

1) 原理

在图形显示设备上显示直线，要有计算机按一定规律，计算出直线上要显示的各点坐标并逐个显示这些点。由于设备坐标系是整数坐标系，所以计算直线上的点，其坐标有时可能正好位于设备坐标系的整数位置上，有时可能正好位于设备坐标系的非整数位置，这就需要选择距离该直线最近的整数点以逼近该直线，可见计算机绘图有两个特点：第一，线段是离散的，即有锯齿阶段效应；第二，线段有误差。

本次实验中采用的直线生成算法为 Bresenham 算法。该算法的基本原理是：过各行各列像素中心构造一组虚拟网格线；按直线从起点到终点的顺序计算直线与各垂直网格线的交点，然后确定该列像素中于此交点最近的像素；该算法采用增量计算，使得对于每一列，只要检查一个误差项的符号，就可以确定该列的所求像素，避免了乘除法和

浮点运算。

设直线从起点（x_1, y_1）到终点（x_2, y_2）。直线可表示为方程$y = mx + b$。其中

$$b = y_1 - mx_1$$

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{dy}{dx}$$

我们的讨论先将直线方向限于 1a 象限在这种情况下，当直线光栅化时，x每次都增加 1 个单元，即

$$x_{i+1} = x_i + 1$$

而y的相应增加应当小于 1。为了光栅化，y_{i+1}只可能选择如下两种位置之一（如图 2）：$y_{i+1} = y_i + 1$或$y_{i+1} = y_i$。

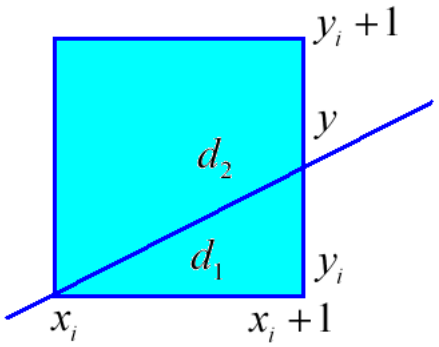


图 2. 直线的整数坐标逼近

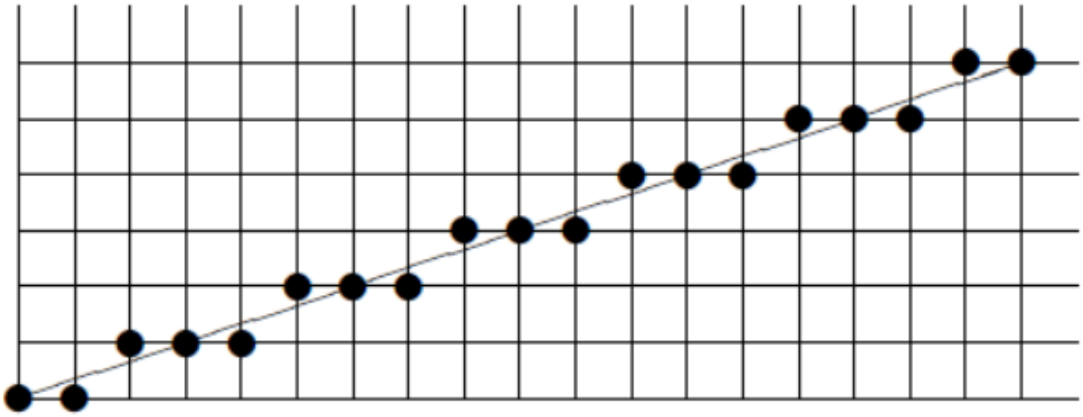


图 3. 直线及其近似点

2) 算法流程

根据图 2，选择的原则是看精确值y与y_i及$y_i + 1$的距离d_1及d_2的大小而定。计算式为：

$$y = m(x_i + 1) + b$$

$$d_1 = y - y_i$$

$$d_2 = y_i + 1 - y.$$

如果$d_1 - d_2 > 0$，则$y_{i+1} = y_i + 1$，否则$y_{i+1} = y_i$。因此算法的关键在于简便地求出$d_1 - d_2$的符号。将上公式带入得

$$d_1 - d_2 = 2y - 2y_i - 1 = 2(x_i + 1) - 2y_i + 2b - 1.$$

用dx乘等式两边，并以$p_i = dx(d_1 - d_2)$代入上述等式，得

$$p_i = 2x_i dy - 2y_i dx + 2dy + dx(2b - 1).$$

$d_1 - d_2$是我们用以判断符号的误差。由于在 1a 象限，dx总大于 0，所以p_i仍旧可以用作判断符号的误差。p_{i+1}为：

$$p_{i+1} = p_i + 2dy - 2dx(y_{i+1} - y_i).$$

误差的初值p_1，可将x_1，y_1，和b代入式中的x_i，y_i而得到：

$$p_1 = 2dy - dx.$$

算法流程图如下：

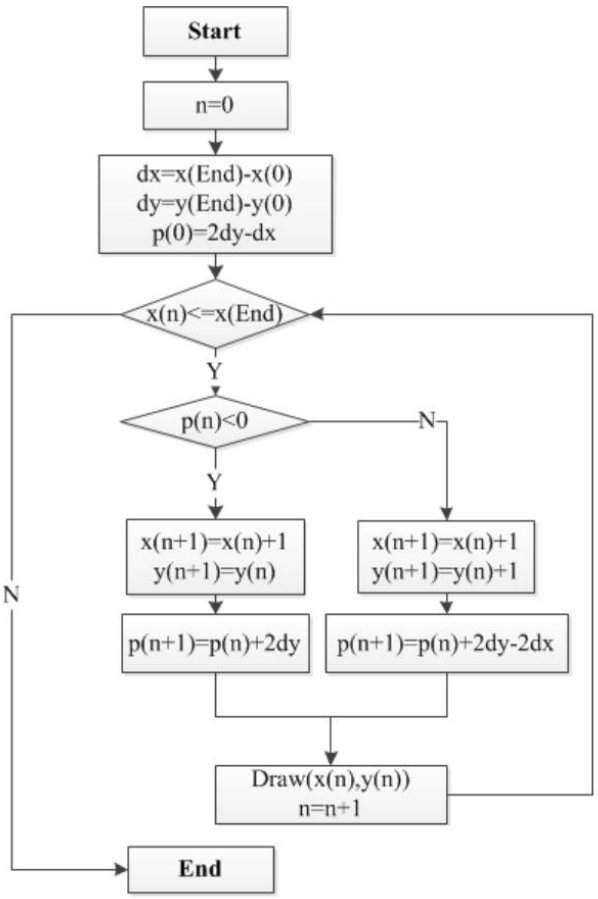


图 4. Bresenham 算法流程

3) 实验结果

实验结果如下图：

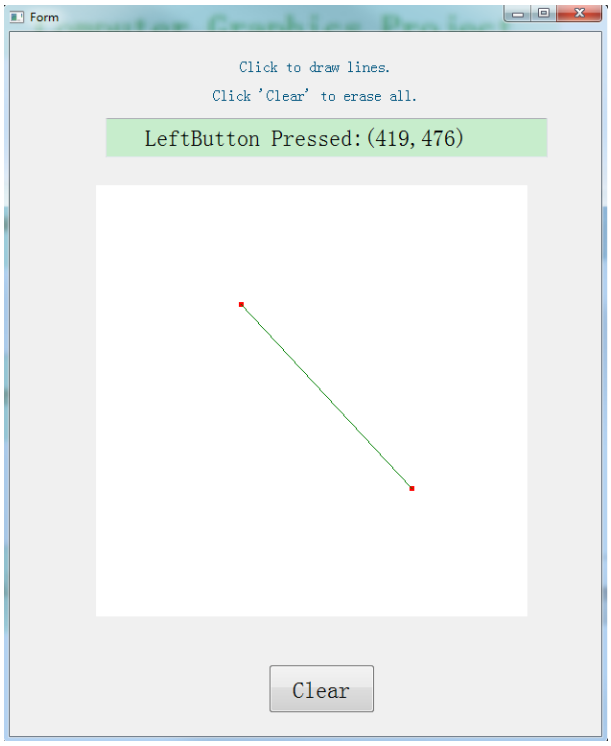


图 5. 直线生成效果图

实验中用鼠标在空白区域单击两点作为直线的起始点和终点（分别用加粗红点标出），可以瞬间得到连接两点的直线。

在该区域不同地方不断地点击，可以画出不同的直线。**不需要**画一条直线清除一次，很方便快捷。

单击 Clear 按钮会清除屏幕上的直线。

屏幕上方会实时显示窗口捕捉到的信息：鼠标按键信息和当前点的坐标。

3.2.2. 圆

1) 原理

不失一般性地，假设圆的圆心位于坐标原点（如果圆心不在原点，可以通过坐标平移使其与原点重合），半径为 R。以原点为圆心的圆 C 有四条对称轴：$x = 0, y = 0, x = y$和$x = -y$。若已知圆弧上一点$P_1 = C(x, y)$，利用其对称性便可以得到关于四条对称轴的其它 7 个点，即：

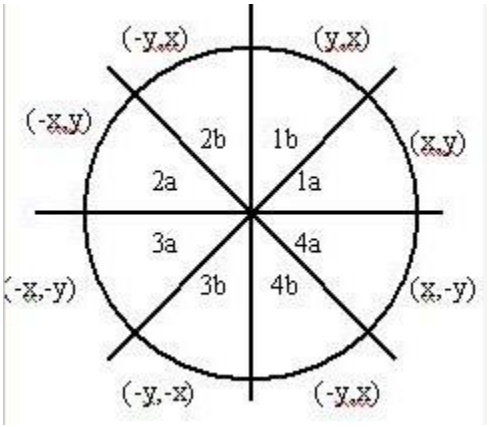


图 7. 八对称圆

P_2＝C（x,－y），

P_3＝C（－x,y），

P_4＝C（－x,－y），

P_5＝C（y，x），

P_6＝C（－y，x），

P_7＝C（y，－x），

P_8＝C（－y，－x）。

这种性质称为八对称性。

因此，只要扫描转换八分之一圆弧，就可以通过圆弧的八对称性得到整个圆。

简单图形的扫描转换常用算法是 Bresenham 算法。它的思想在于用误差量来衡量点选取的逼近程度。其过程如下：以平面二维图形的扫描转换为例，设要画的图形方程为$F(x,y)=0$，要画的区域为$[x_0,x]$（不妨设x方向是最大位移方向，即$\Delta x > \Delta y$），则$F(x,y)$也是一个误差度量函数，我们拿离散的点值代入如果大于 0 则正向偏离，否则负向偏离，等于 0 的情况比较少，它表示的是不偏离即恰好与真实点重合。既然x是最大位移方向，那每次对x自增 1，相应的y可以选择不增或增 1(或-1，具体问题具体分析)，选择的方法就是$d = F(x+1, y \pm 0.5)$的正负情况进行判断从而选择y的值。

实际情况中还要考虑到浮点数的计算问题，因为基本的图形扫描转换算法最好能够硬件实现，所以摆脱浮点数是最好的，常用的方法是对d进行递推，而不是直接由$F(x,y)$给出(直接给出速度会慢)。

(a).圆弧曲线　　(b). 圆弧的离散表示

图 8. 圆的离散表示

2) 算法流程

给出圆心的坐标(0, 0)和半径 R，求圆图像的最佳逼近点。

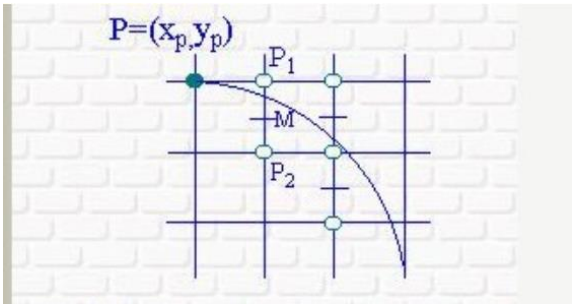


图 9. 候选点的选取

圆是中心对称的特殊图形，所以可以将圆八等分，则只须对八分之一圆孤求解，其它圆孤可以由对称变换得到，我们求的八分之一圆孤为$(0,R)-(R\sqrt{2},R\sqrt{2})$，可知最大位移方向是$x$方向，$x_0=0,\ y_0=R$。每次对$x$自增，然后判断$y$是否减 1，直到$x\geq y$为止(从点$(0,R)$到圆的八分之一处就有这种情况)。误差量由

$$F(x,y)=x\text{^}2+y\text{^}2-R\text{^}2$$

给出。

先找递推关系，若当前$d=F(x+1,y-0.5)>0$，则y须减 1，则下一d值为

$$\begin{aligned} d &= F(x+2,y-1.5)\\ &= (x+2)^2+(y-1.5)^2-R^2\\ &= (x+1)^2+(x-0.5)^2-R^2+2x+3-2y+2\\ &= d+2x-2y+5. \end{aligned}$$

若当前$d=F(x+1,y-0.5)<0$，则y不变，只有x增 1，则下一d值为

$$d=F(x+2,y-0.5)=d+2x+3。$$

d 的初值:

$$d_0=F(1,R-0.5)=1.25-R,$$

则可以对$d-0.25$进行判断，因为递推关系中只有整数运算，所以$d-0.25$相当于$d>0$。所以d取初值$1-R$。

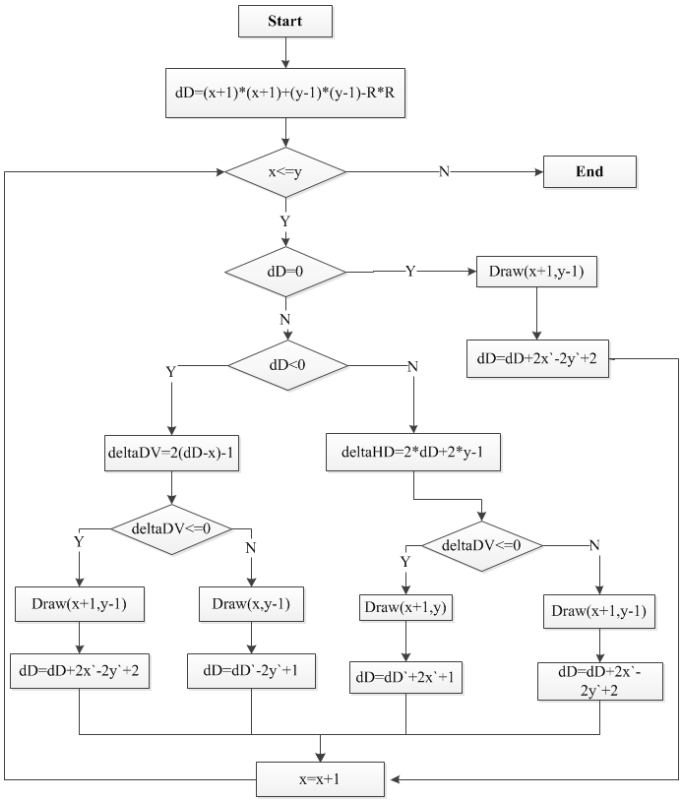


图 10. Bresenham 圆算法

3) 实验结果

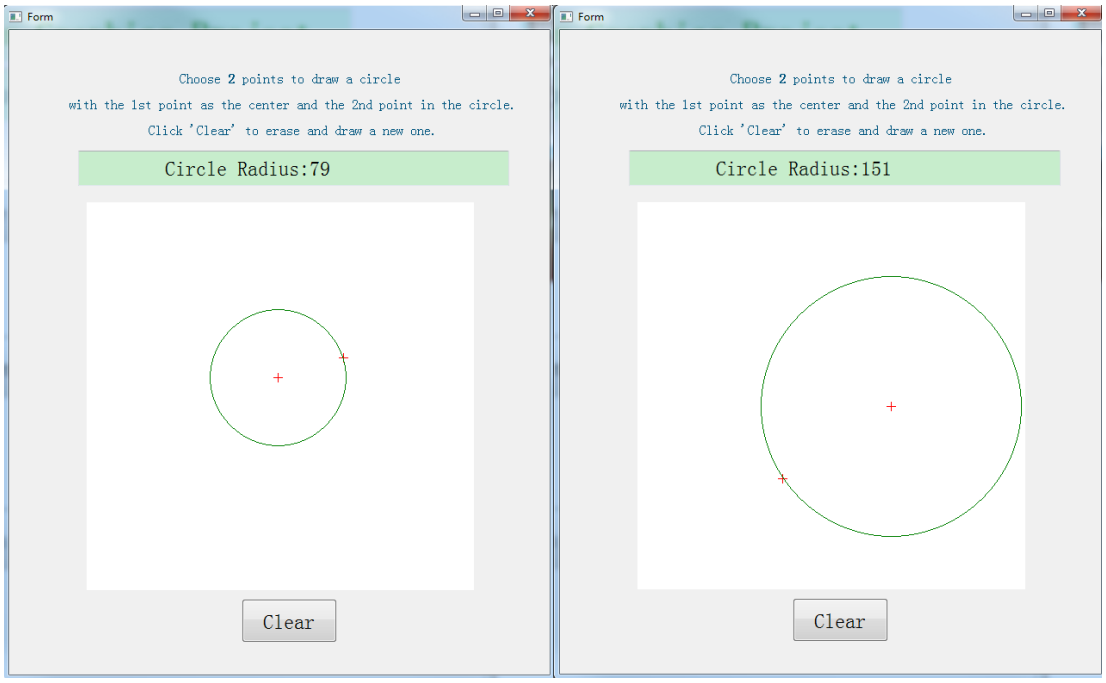


图 11. Bresenham 圆生成

实验结果如上图。

先选择 1 点作为圆心，再选择 1 点确定圆半径（四舍五入），可以得到一个完整的圆。
单击 Clear 可以清楚屏幕的圆进行重画。
窗口上方会动态显示选择的点的坐标以及圆半径。

3.2.3.椭圆

1) 原理

本次实验选用中点椭圆生成算法。将中点圆算法应用到椭圆上就是中点椭圆算法，则中点椭圆算法的基本思想是利用中点在椭圆内部还是外部来判定下一个点的位置。根据决策图 2.4 可判定

（a）如果$f(M)<0$，说明点 M 在椭圆内，取点 P1 画点。

（b）如果$f(M)\geq 0$，说明点 M 在椭圆外，取点 P2 画点。

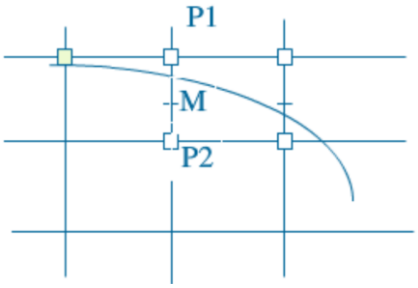


图 12. 中点椭圆算法画点决策图

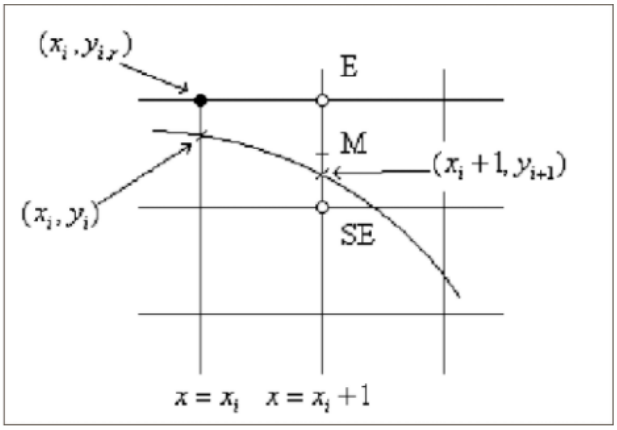


图 13. 椭圆上段圆弧

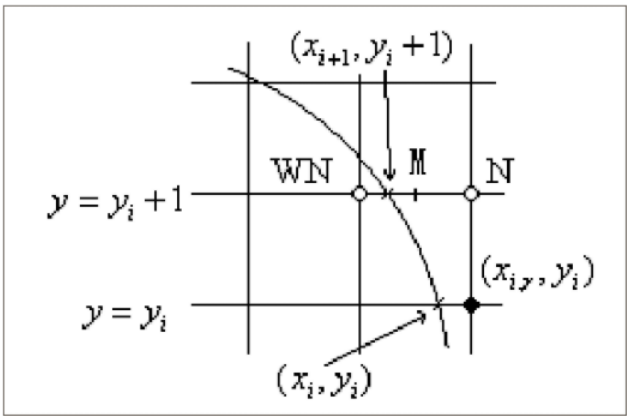


图 14. 椭圆下段圆弧

2) 算法流程

算法步骤:

a) 输入椭圆的长半轴a和短半轴b。

b) 计算初始值$d = b*b + a*a*(-b + 0.25)$， $x = 0, y = b$。

c) 绘制点 (x, y)及其在四分象限上的另外 3 个对称点。

d) 判断d的符号。若$d <= 0$，则先将 d 更新为$d + b*b*(2*x + 3)$，再将 (x, y)更新为$(x+1, y)$；否则先将d更新为

$$d + b*b*(2*x + 3) + a*a\,(-2*y + 2),$$

再将d 更新为$(x+1, y-1)$。

e) 当$b*b*(x+1) < a*a*(y - 0.5)$时，重复步骤 c)和 d)，否则转到步骤 f)。

f) 用上半部分计算的最后点(x, y)来计算下半部分中d的初值:

$$d = b*b*(x + 0.5)^2 + a*a*(y - 1)^2 - a*a*b*b。$$

g) 绘制点(x, y)及其在四分象限上的另外 3 个对称点。

h) 判断d的符号。若d <= 0，则先将d更新为

$$d + b*b*(2*xi + 2) + a*a*(-2*yi + 3),$$

再将d更新为$(x+1, y-1)$；否则先将d更新为$d + a*a*(-2*yi + 3)$，再将d更新为$(x, y-1)$。

i) 当$y >= 0$，重复步骤 g)和 h)，否则结束。

算法流程图如图 15。

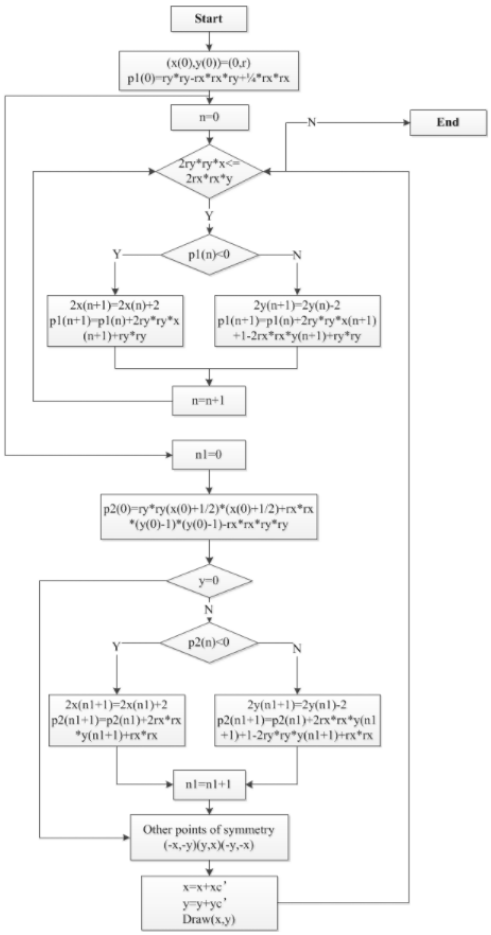


图 15. Bresenham 椭圆算法

3) 实验结果

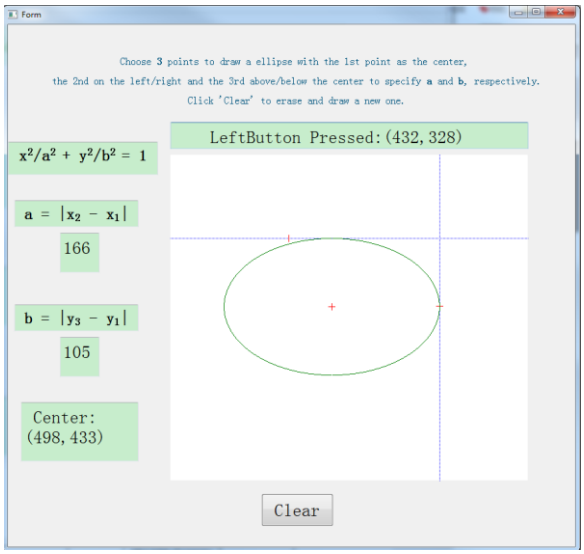


图 16. Bresenham 椭圆生成（1）

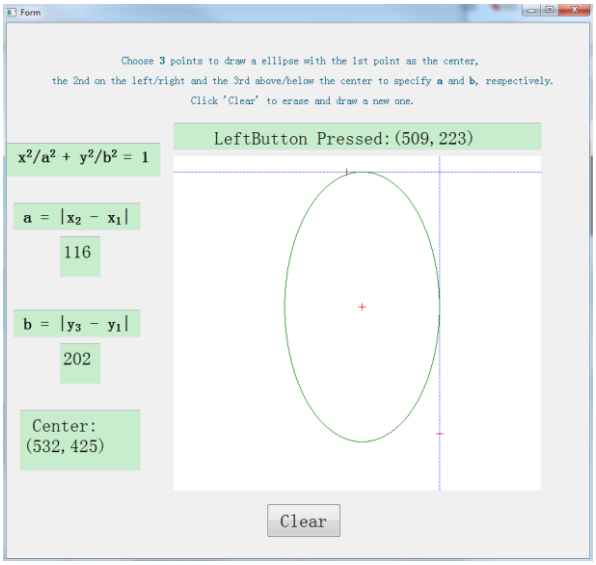


图 17. Bresenham 椭圆生成（2）

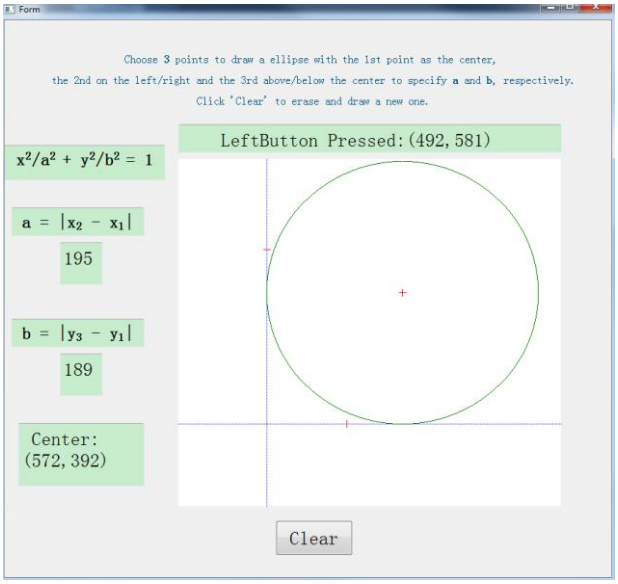


图 18. Bresenham 椭圆生成（3）

实验结果如上三图。

需要用户点击 3 个点来生成一个椭圆。第一个点是椭圆圆心，第二个点确定椭圆的长边 a，第 3 个点确定椭圆的短边 b。选定之后，会生成两条边界虚线以及椭圆。

窗口上方实时显示鼠标点击信息及点坐标，左边显示椭圆的参数信息。

单击 Clear 擦除窗口中的椭圆与直线进行重画。

3.2.4. 区域填充

1) 原理

填充所有在边界内的相连通的像素，主要分下面几个步骤：

1. 从区域内部一个像素点开始
2. 判断这个像素是否是一个边界像素点或者已经被填充了
3. 如果都不是，就把它填充，然后开始设置邻居像素点。

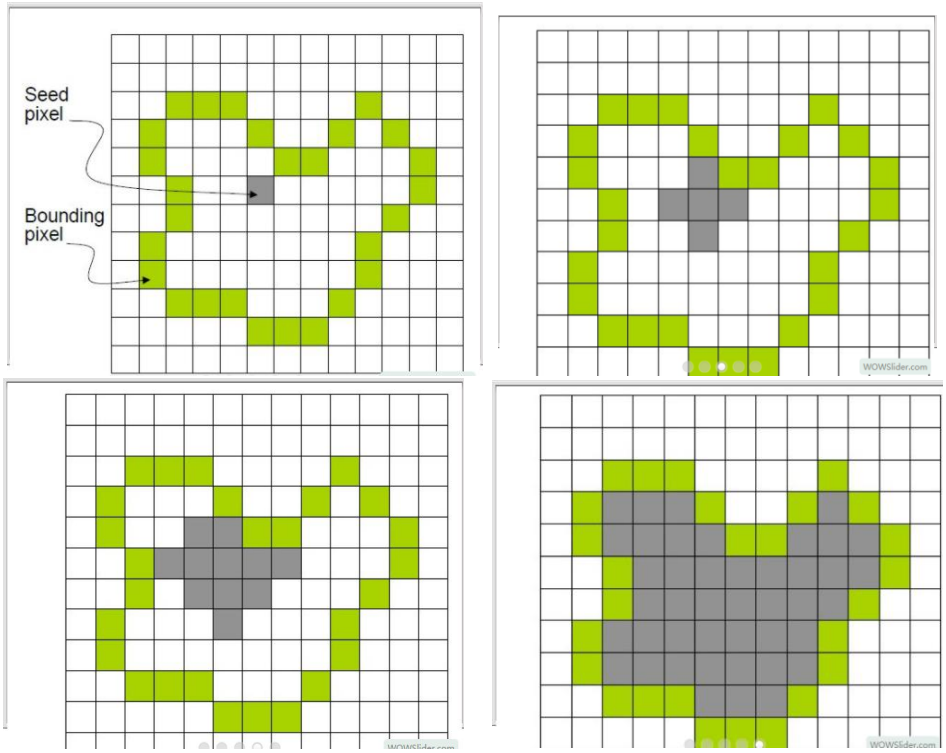


图 19. 区域填充流程

如果两个像素连通，则它们之间有一条“相邻（adjacent）”像素组成的连续路径，所以连通的概念就依赖“相邻”的定义。在图形学中，相邻通常有两种定义：

- 相邻（4-Adjacent）:两个像素是四相邻的，则它们在彼此水平或者垂直相邻的位置上。

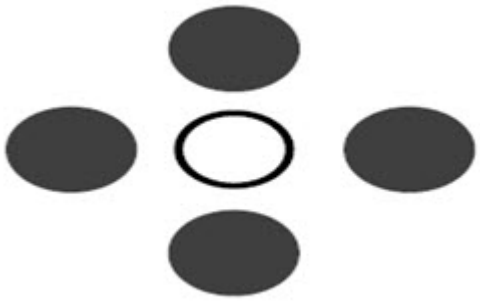


图 20. 四相邻

- 八相邻（8-Adjacent）:两个像素是八相邻的，则它们在彼此水平、垂直或者是斜方向上相邻的位置

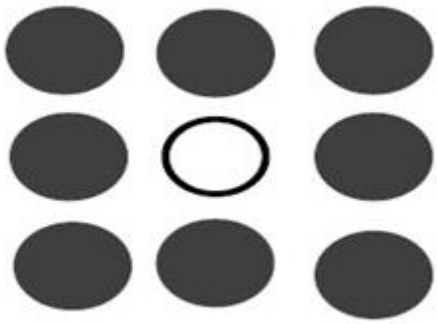


图 21. 八相邻

2) 算法流程

本实验采用边界填充算法。

边界填充算法采用四连通区域，从内部一个像素点开始，并用新的颜色替换它填充四连通，直到所有的内部点被替换了。算法描述如下：

```
void Filling(x, y)
{
    if(点(x, y)不是边界点)
    {
        drawPoint(x, y);
        Filling(x-1, y);
        Filling(x+1, y);
        Filling(x, y-1);
        Filling(x, y+1);
    }
}
```

缺点是：1）大量的嵌套调用；2）很多像素点可能会被测试多次；3）难以清楚的掌控由于嵌套调用所占的内存大小；4）如果算法多次测试一个像素，会导致占用的内存扩大。

改进的边界填充算法如下：

每次填充在同一条扫描线上相邻的一排像素，同时把与它相邻的未填充的种子像素放在堆栈中。伪码如下所示：

```
Stack.push(种子像素);
while(栈非空)
{
    栈首元素出栈，得到下一个种子像素 seed;
    Filling(seed);
    Filling(seed 上面的行);
    Stack.push(该行最右的像素);
    Filling(seed 下面的行);
    Stack.push(该行最右的像素);
}
```

3) 实验结果

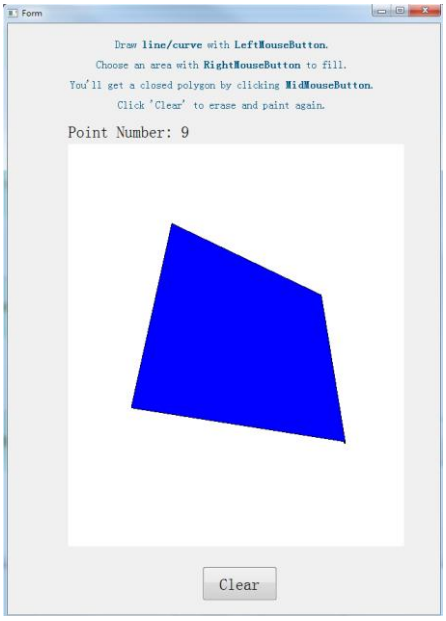


图 22. 规则多边形的填充

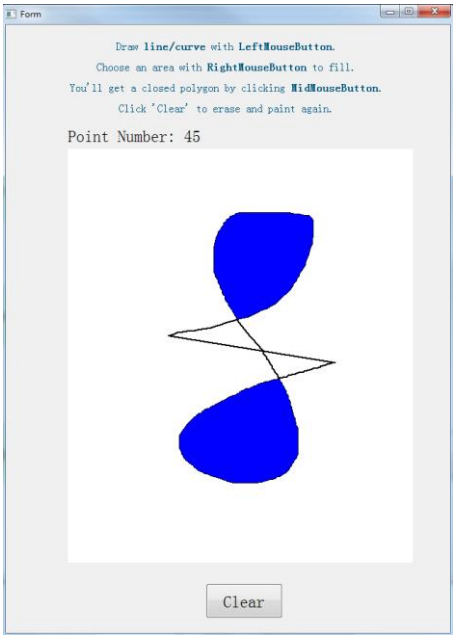


图 23. 不规则图形的内填充

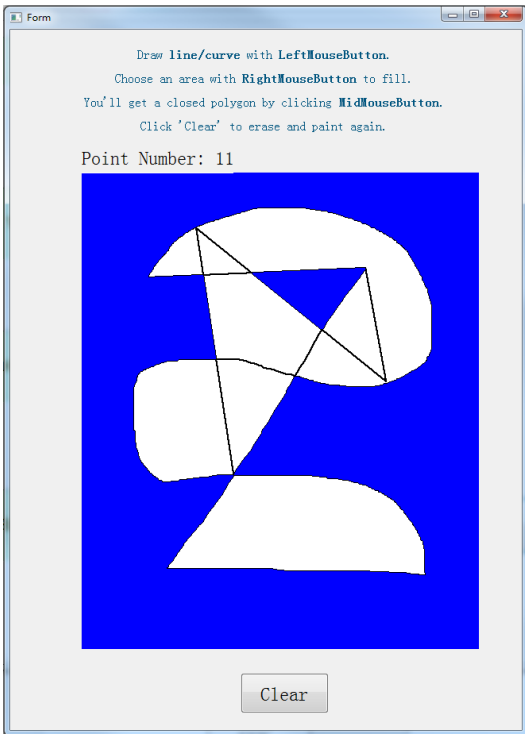


图 23. 不规则图形的外填充

实验结果如上三图所示。

鼠标左击画点，长按鼠标左键可持续画出曲线。

由于用于一般难以用鼠标连接起始点和终点，因此，该窗口提供了**封闭图形**的功能：鼠标中键点击任何一个地方，会自动连接曲线起点和终点，得到一个封闭图形。当然，用户可以选择不进行封闭。

鼠标右击闭合区域可以进行快速填充。连续点击多个区域可以进行多区域填充。而且，填充之后用户可以选择继续画线，并进行再次填充。

窗口上方实时显示用户点击/画点数量。

点击 Clear 对窗口进行清屏处理。

需要说明的是，用户用鼠标画线时，系统调用的是最开始定义的 Line 类（生成直线定义的类）的划线函数 BresenhamLine，而避免了在 Fill 类（填充类）重复定义。

3.3. 样条曲线

3.3.1. Bezier 曲线

1) 原理

Bezier 曲线由法国工程师皮埃尔·贝塞尔（Pierre Bézier）提出，并成功运用于汽车的外形设计中。

Bezier 曲线由 n+1 个控制点$\boldsymbol{P_i}$来定义，其向量形式为

$$\boldsymbol{P}(u)=\sum_{i=0}^{n}\boldsymbol{P}_i\cdot B_{i,n}(u).$$

其中参数u的取值范围是$u\in[0,1]$;

$$B_{i,n}(u)=C(n,i)\cdot u^i\cdot(1-u)^{n-i},$$

$$C(n,i)=\frac{n!}{i!\cdot(n-i)!}.$$

标量形式为：

$$\begin{cases}x(u)=\sum_{i=0}^{n}x_i\cdot B_{i,n}(u)\\ y(u)=\sum_{i=0}^{n}y_i\cdot B_{i,n}(u)\end{cases}$$

Bezier 曲线的性质：

- Bezier 曲线在起点和终点的切线方向与其特征多边形的第一条边和最后一条边的走向一致。
- 对称性：保持 Bezier 曲线特征多边形的各顶点不变，而将其次序颠倒，新的 Bezier 曲线与原曲线形状相同，走向相反。
- 凸包性：当 Bezier 曲线特征多边形是凸包状时，Bezier 曲线曲线上点都落在凸包中。
- 几何不变性：Bezier 曲线的形状仅与特征多边形的顶点有关，与具体坐标选择无关。
- 全局控制性：改变特征多边形任意一个顶点，会对整个曲线产生影响。
- 变差缩减性：对于平面图形，一条直线与 Bezier 曲线的交点个数不多于该直线与该特征多边形的交点个数。
- 曲线的可分割性：可以利用 Bezier 曲线的递归分割多边形来逼近 Bezier 曲线本身。

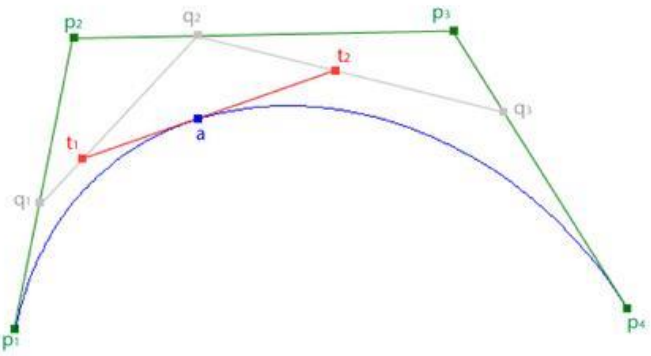


图 24. Bezier 曲线示意图

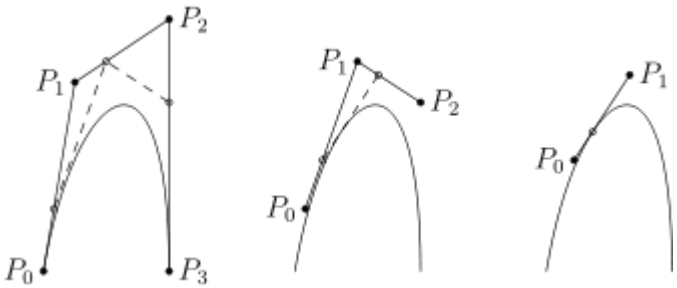
2) 算法

- 基本算法：将$u\in[0,1]$均匀分成 1000 段，得到控制点坐标之后，计算 Bezier 曲线上对应每个u的点的坐标。

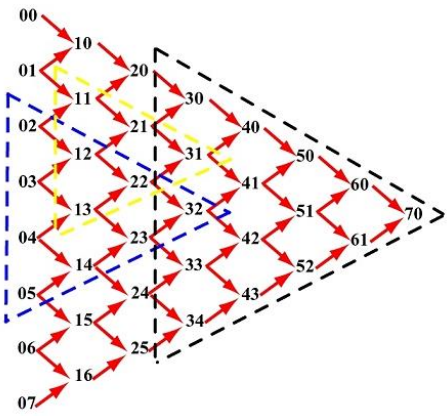
```
const doublue du = 0.001;
void Bezier::BezierCurve(QVector<QPoint> P, QPainter &painter)
{
    int n = P.size() - 1;
    for (double u = 0; u <= 1; u += du)
    {
        double xu = 0.0;
        double yu = 0.0;
        for (int i = 0; i <= n; ++i)
        {
            double C = Factorial(n)/(Factorial(i)*Factorial(n - i));
            double B = C * pow(u, i) * pow(1 - u, n - i);
            xu += B * P[i].x();
            yu += B * P[i].y();
        }
        painter.drawPoint(QPoint(xu, yu));
    }
}
```

- 利用 Bezier 曲线的可分割性，采用 Casteljau 算法递推公式

$$P_i^r(u) = (1-u)P_i^{r-1}(u) + \mathrm{u}P_{i+1}^{r-1}(u),$$
$$r = 1,2,\dots,n; i = 0,1,\dots,n-r;$$



a



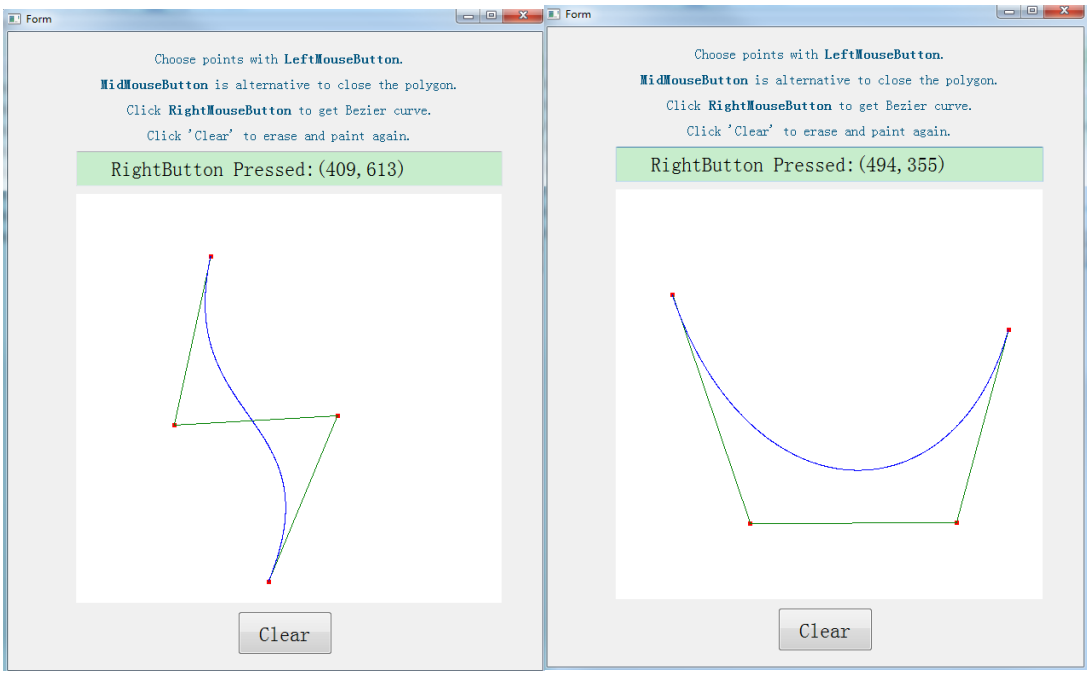
b

图 25. Casteljau 算法递推公式

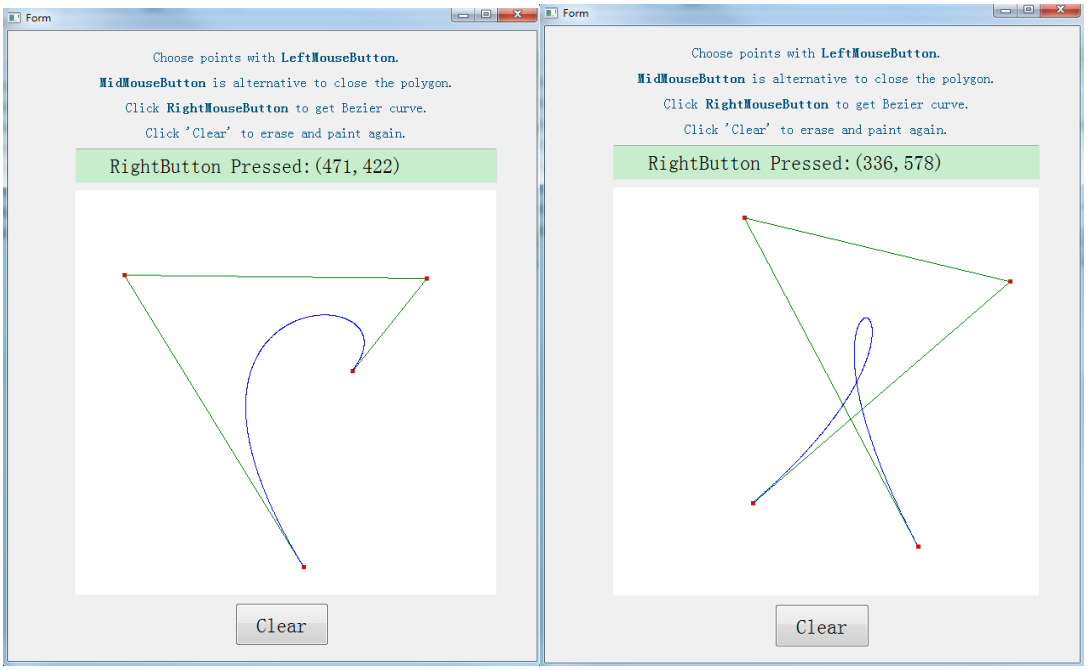
部分代码如下：

```
for (int r = 1; r <= n; ++r)
{
    int i = 0;
    while (i <= n - r)
    {
        px[i] = (1 - u) * px[i] + u * px[i + 1];
        py[i] = (1 - u) * py[i] + u * py[i + 1];
        ++i;
    }
}
```

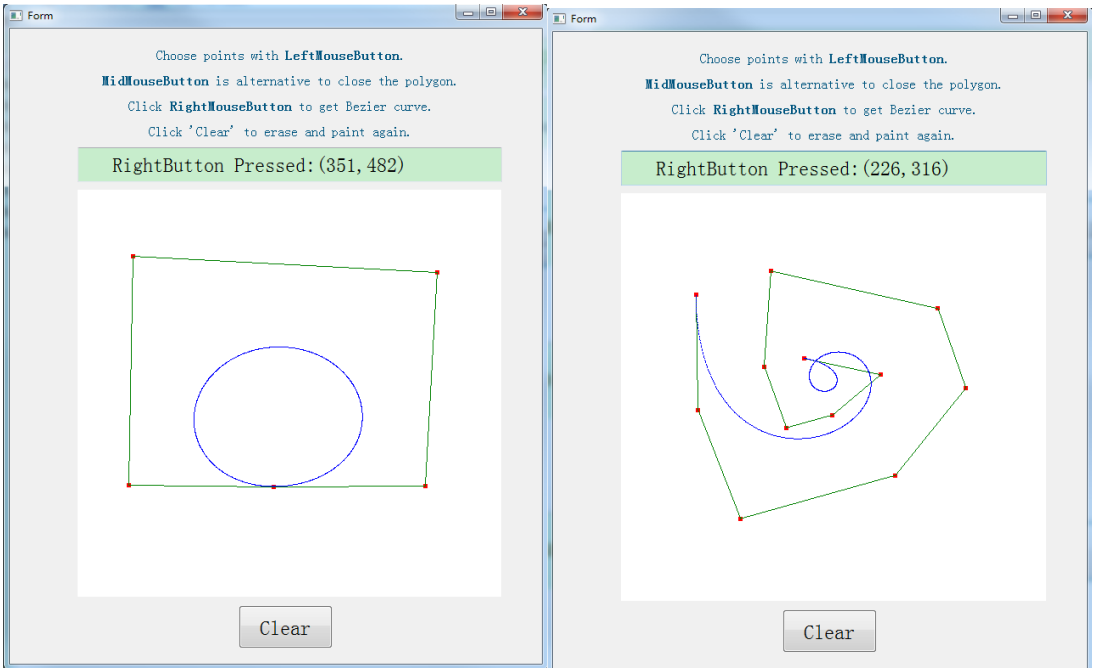
3) 实验结果



a b



c d



e f

图 26. Bezier 曲线生成。

实验结果如图 25 所示。

点击鼠标左键确定控制点。点击鼠标右键生成 Bezier 曲线。可以选择点击鼠标中键闭合该特征多边形。

点击 Clear 清除曲线进行重画。

窗口上方实时显示控制点的坐标以及鼠标点击信息。

3.3.2. B 样条曲线

1) 原理

与 Bezier 曲线相比，B 样条曲线对其特征多边形更为逼近，曲线更加光滑（很容易达到二阶导数连续），其多项式的次数可由人们根据需要决定，而 Bezier 曲线多项式的次数由控制点的个数确定。因此在曲线的形状设计中，B 样条曲线更具有优越性，得到越来越广泛的应用。

B 样条曲线也是用一组混合基函数来组合控制点，用以产生对应的 B 样条曲线，其向量表达式如下：

$$\boldsymbol{P}(u)=\sum_{i=0}^{n}\boldsymbol{P}_i\cdot N_{i,k}(u)\quad(u_1\le u\le u_2.)$$

式中，n为控制点的个数，这时 B 样条曲线共有n+1 个控制点；$\boldsymbol{P}_i$为控制点的坐标，依次连接各控制点所形成的折线为 B 样条曲线的特征多边形；$N_{i,k}(u)$为 B 样条曲线的基函数，它用递归的方式定义为

$$\begin{cases}N_{i,1}(u)=\begin{cases}1 & t_i\le u<t_{i+1}\\ 0 & \text{其他}\end{cases}\\ N_{i,k}(u)=\dfrac{u-t_i}{t_{i+k-1}-t_i}N_{i,k-1}(u)+\dfrac{t_{i-k}-u}{t_{i+k}-t_{i+1}}N_{i+1,k-1}(u)\end{cases}$$

由于此分母可能为 0， 故采用$\frac{0}{0}=0$这个约定，参数u的变化范围是$[u_1,\ u_2]=[t_{k-1},\ t_{n+1}]$.

在上式定义中，k控制曲线连续的阶数，由使用者根据实际需要指定。此时，B 样条曲线是参数u的$k-1$阶多项式且保持$k-2$阶导数连续。 t_i为结点向量，作用是联结 B 样条曲线的自变量参数u与各控制点，使得各控制点也能影响 B 样条曲线的形状。

2) 算法

De Boor 算法是快速而且数值上稳定的算法，用于计算 B 样条形式的样条曲线。这是用于 Bezier 曲线的 de Casteljau 算法的一个推广。

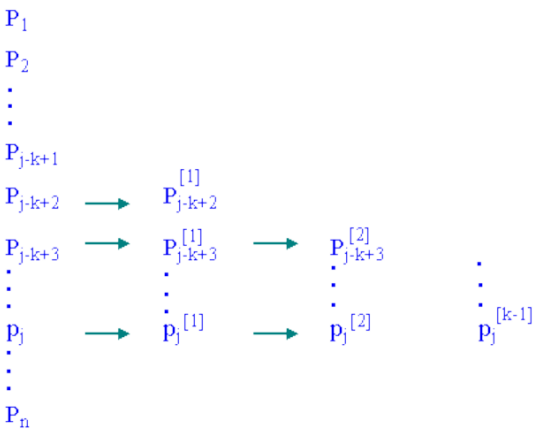


图 27. De Boor 算法系数递推关系图

3) 实验结果

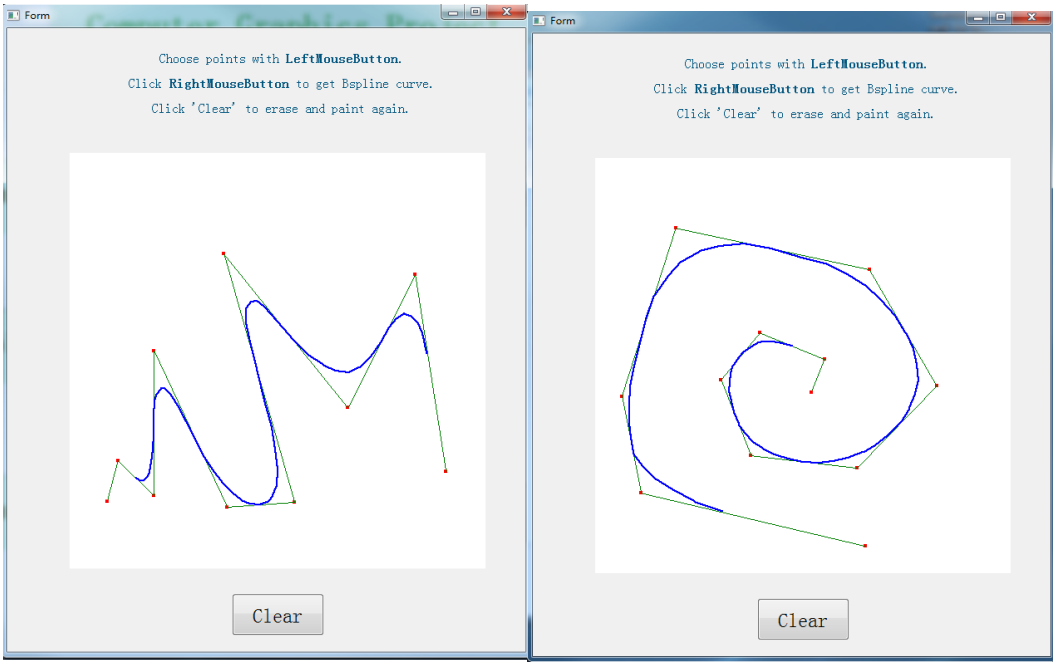


图 28. B 样条曲线生成

实验结果如图 25 所示。

点击鼠标左键确定控制点。点击鼠标右键生成三阶 B 样条曲线。

点击 Clear 清除曲线进行重画。

3.4. 分形图形

3.4.1. Koch 雪花分形

1) 原理

瑞典人 Koch 于 1904 年提出了著名的“雪花”曲线，这种曲线的作法是，从一个正三角形开始，把每条边分成三等份，然后以各边的中间长度为底边。分别向外作正三角形，再把“底边”线段抹掉，这样就得到一个六角形，它共有 12 条边。再把每条边三等份，以各中间部分的长度为底边，向外作正三角形后，抹掉底边线段。反复进行这一过程，就会得到一个“雪花”样子的曲线。这曲线叫做科赫曲线或雪花曲线。

反复进行这一作图过程，得到的曲线越来越精细。

Koch 曲线有着极不寻常的特性，不但它的周长为无限大，而且曲线上任两点之间的距离也是无限大。该曲线长度无限，却包围着有限的面积。

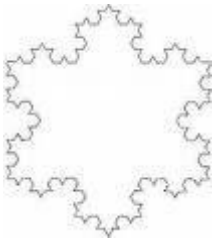


图 28. Koch 雪花分形

很神奇的一个曲线，他说明了一个悖论：“无限长度包围着有限面积。”

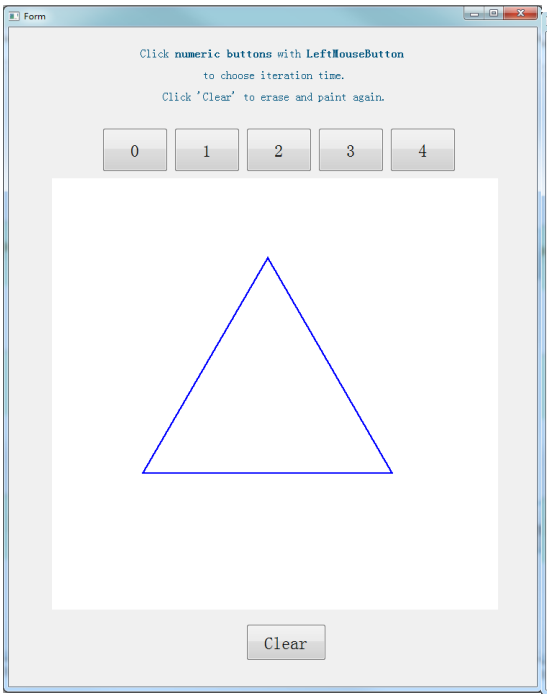
2) 算法

在一单位长度的线段上对其三等分，将中间段直线换成一个去掉底边的等边三角形，再在每条直线上重复以上操作，如此进行下去，就得到分形曲线 Koch 曲线。

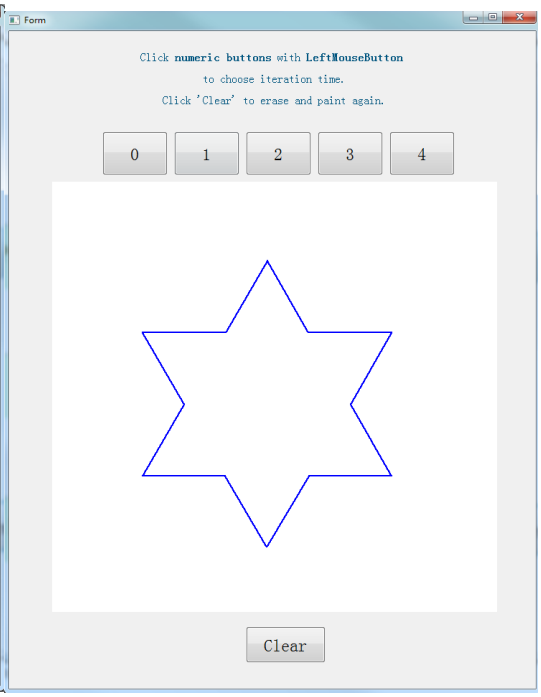
本实验中实现的 Koch 分形可以指定迭代的次数 level。具体算法如下。

```
初始化点集 points 为等边三角形的三个顶点；
Koch(points, level)
{
    if level==0, return
    复制点集 points 到临时点集 tpoints；
    for 每两个相邻的点 p1, p2 in tpoints
        compute(线段 p1, p2 直线需要插入的三个点的坐标)；
        points.Insert(三个点的坐标)；
    end for
    level = level - 1;
    Koch(points, level);
 }
connect(points);
```

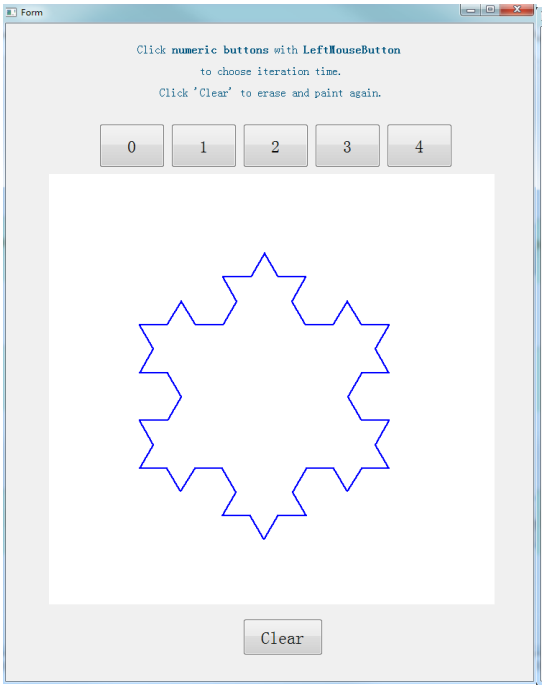
3) 实验结果



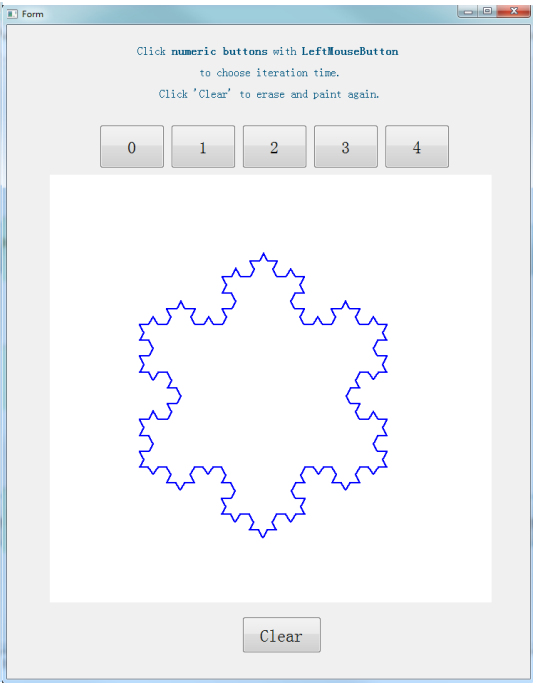
a. 迭代 0 次



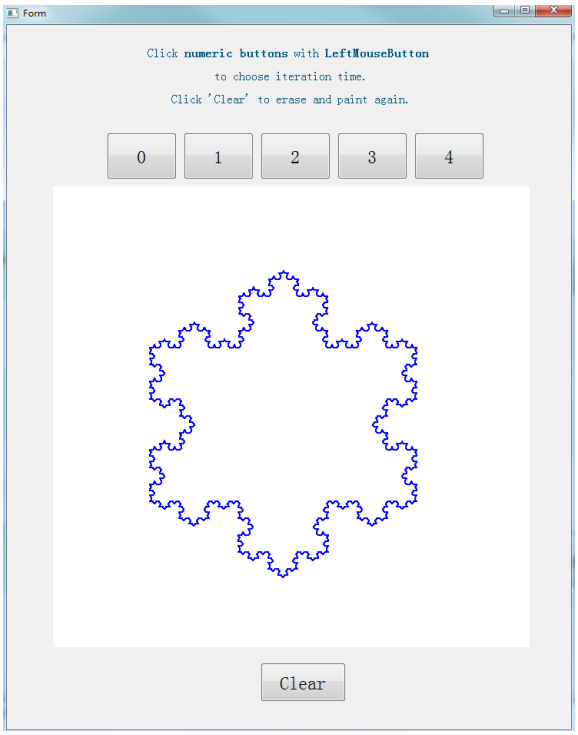
b. 迭代 1 次



c. 迭代 2 次



d. 迭代 3 次



e. 迭代 4 次

图 29. 各级 Koch 雪花分形

实验结果如图 29。

窗口中可以通过点击数字按钮选择迭代次数，0 表示生成一个等边三角形，1 表示在等边三角形的基础上迭代一次生成 Koch 分形，2、3、4 类推。

点击 Clear 进行清屏。

3.4.2. Mandelbrot 集

1) 原理

Mandelbrot 集合是在复平面上组成分形的点的集合，它正是以数学家 Mandelbrot 命名。

Mandelbrot 集合可以用复二次多项式

$$f_c(z) = z^2 + c.$$

来定义。其中c是一个复参数。对于每一个c，从开始对$f_c(z)$进行迭代。

序列$(0, f_c(0), f_c(f_c(0)), f_c\left(f_c(f_c(0))\right), \ldots\ldots)$的元素的模（复数具有模的概念）或者延伸到无穷大，或者只停留在有限半径的圆盘内。Mandelbrot 集合就是使以上序列不延伸至无限大的所有c点的集合。

从数学上来讲，Mandelbrot 集合是一个复数的集合。一个给定的复数c或者属于 Mandelbrot 集合 M，或者不属于。比如，取c =1，那么这个序列就是(0, 1, 2, 5, 26, ...)，显然它的值会趋于无穷大；而如果取c = i，那么序列就是(0, i, -1+i, -i, -1+i, -i,...)，它的值会一直停留在有限半径的圆盘内。

事实上，一个点属于 Mandelbrot 集合当且仅当它对应的序列（由上面的二项式定义）中的任何元素的模都不大于 2。这里的 2 就是上面提到的“有限半径”。

计算机的屏幕上的像素只有有限个，而 Mandelbrot 集合中的点则有无限个。由于序列的的元素有无穷多个，我们只能取有限的迭代次数来模拟了，比如取 100 或 1000 次。

实验中取迭代次数为 256。

2) 算法

实验中首先定义了一个 Complex 复数类用以定义复数以及复数基本运算。

从零开始，对每一个坐标(x,y)，有一个复数c，再从$z=(0,0)$开始根据公式迭代。实验中可以指定$f_c(z)$中z的幂次。同时为了图形的美观，根据迭代次数来计算画笔的颜色，从而生成彩色的图形。

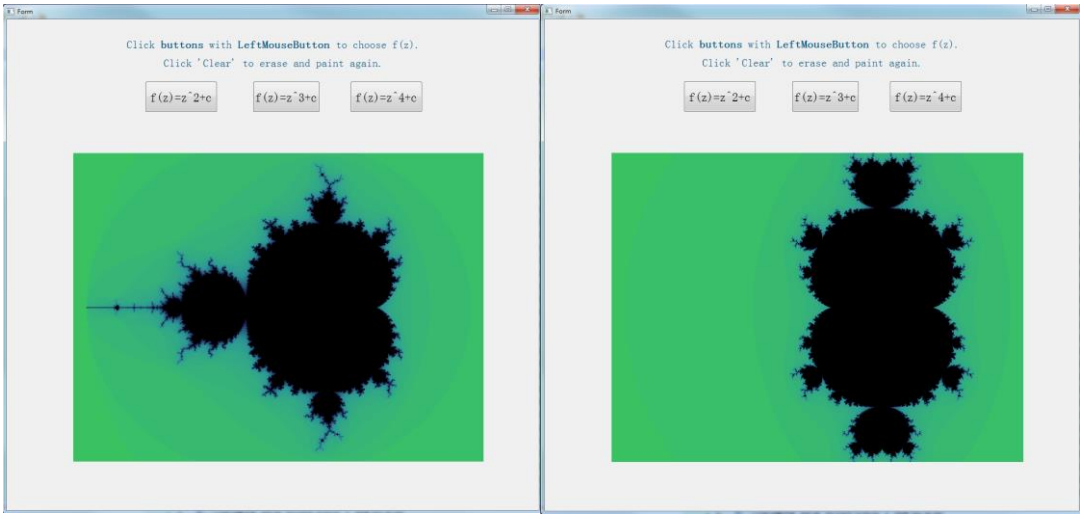
具体算法如下：

```
for x from 0 to width
    for y from 0 to height
        Get(c);//计算复数 c
        z = (0, 0);
        for n from 0 to 256
            if (z.modular > 2) break;//若 z 模值超过 2，跳出迭代
            z = f_c(z);
        end for
        pen.setcolor(n);//根据 n 计算画笔颜色
        drawpoint(x, y);
    end for
end for
```

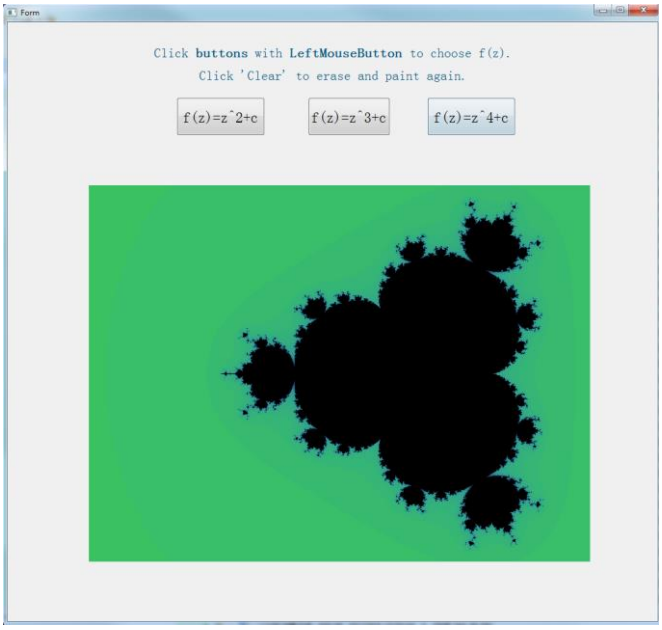
其中，计算画笔颜色是根据迭代次数 n 的值，计算 RGB 颜色，所以画图窗口的图框内每一个点都会有一个颜色值。

3) 实验结果

实验结果如图 30。



a . 2 次幂　　b. 3 次幂



c. 4 次幂

图 30. 各幂次的 Mandelbrot 集

用户可以通过点击窗口中的按钮来选择不同幂次的 Mandelbrot 集。幂次越高，画图时间越长。

由于迭代次数有限，Mandelbrot 集的细节信息并不清晰。如果有时间的话，我将会对该分形图形加入放大功能以观察细节信息。

3.4.3. Julia 集

1) 原理

原理上，Julia 集与 Mandelbrot 集非常相似。Julia 集由一个复变函数

$$f(z) = z^2 + c.$$

的迭代生成。根据c的不同，经过简单的迭代过程，就能生成各种各样复杂而又神奇的分形图形。

Julia 集是动力系统的一个斥子（repeller）。斥子的反义是吸引子，所谓吸引子，就是存在一个开集V和闭集F和一个映射f，如果满足$f(F) = F$，而且V里面所有点x，都有$f(x)$迭代次数越多，结果离F越远；则称F为f的吸引子，V为F的吸引域。
类似的，对一个闭不变子集F，如果F附近的所有点经迭代后远离F，则F是f的斥子。

2) 算法

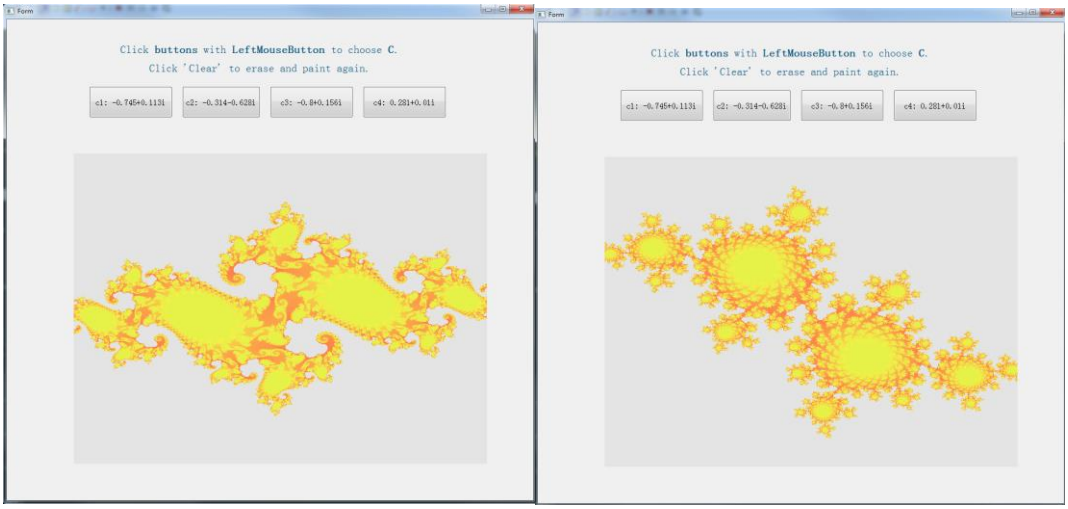
算法与生成 Mandelbrot 的算法相似，存在细微差别。

```
选择一个固定的c;
for x from 0 to width
    for y from 0 to height
        Get(z);//计算复数 z
        for n from 0 to 256
            if (z.modular > 2) break;//若 z 模值超过 2，跳出迭代
            z = f(z);
        end for
        if (n>32) pen.setColor(QColor(228, 228, 228));
        pen.setcolor(n);//根据 n 计算画笔颜色
        drawpoint(x, y);
    end for
end for
```

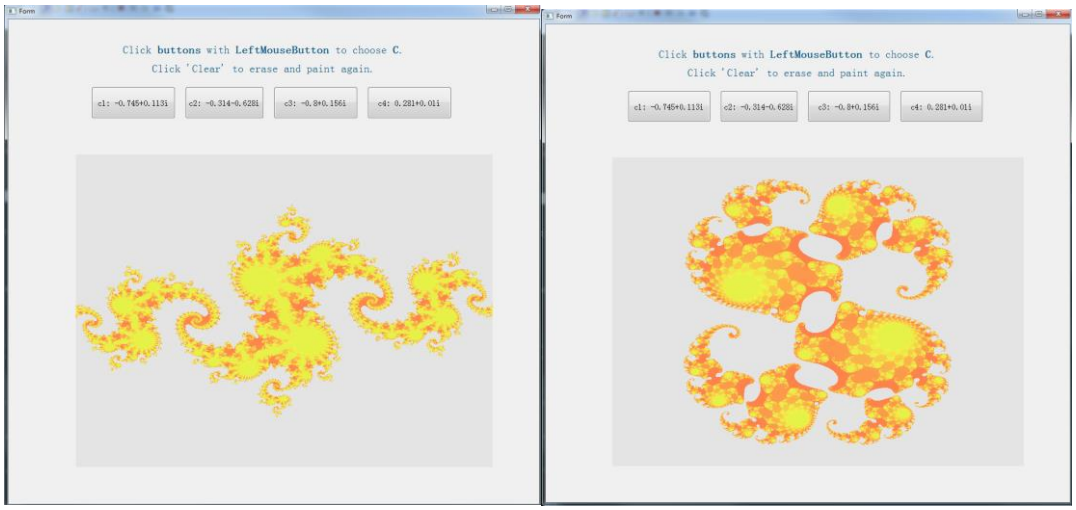
注意到算法较 Mandelbrot 多了红色标注的一行，这是因为在实验中发现：如果直接套用 Mandelbrot 集的配色方案，则得到的图形容易和背景色混为一体，视觉效果较差。所以增加这句以对背景的颜色进行渲染，得到更佳的视觉效果。

3) 实验结果

实验结果如图 31 所示。



a. C = -0.745 + 0.113i　　b. C = -0.314 – 0.618i



c. C = -0.8 + 0.156i　　d. C = 0.281 + 0.01i

图 30. 几种不同的 Julia 分形

用户可以点击窗口中的按钮选择不同的 C 以生成不同的 Julia 分形图形。

观察发现，本实验得到的 Julia 分形图形颜色略微单调，这是以后可以改进的一点。另外如同前面的 Mandelbrot 集，也可以增加图形缩放的功能。

3.4.4. 蕨类植物

1) 原理

蕨类植物是一种常见的自相似图案，这意味着它们的图案能以任何放大率或缩小率生成和复制。描述蕨类植物的数学公式是根据迈克尔·巴恩斯利的名字命名的，它是第一个显示这种混沌状态是不可预知的，而且一般是遵循确定性法则(以非线性循环方程组为基础)。也就是说，利用巴恩斯利的蕨类植物公式反复生成的任意数字，最终生成一个独一无二的蕨类植物形状的物体。

蕨类植物图形生成是基于迭代函数系统（IFS）的一个典型样例。IFS 是分形理论的一个重要分支，它将待生成的图像看作是由许多与整体相似（自相似）或经过一定变换与整体相似的（自仿射）小块拼贴而成。

仿射变换定义为：

$$x = a * x + b * y + e;$$
$$y = c * x + d * y + f.$$

其中a, b, c, d, e, f为仿射变换系数。

用多个仿射变换式表达一个图形，使用每一个仿射变换式的概率可以不同，一般面积越大，概率就越大。因此，只要获得仿射变换参数和概率值，就能够生成要表达的图形。例如：

a	b	c	d	e	f	p
0.00	0.00	0.00	0.16	0.00	0.00	0.01
0.85	0.04	-0.04	0.85	0.00	1.60	0.85
0.20	-0.26	0.23	0.22	0.00	1.60	0.07
-0.15	0.28	0.26	0.24	0.00	0.44	0.07

2) 算法

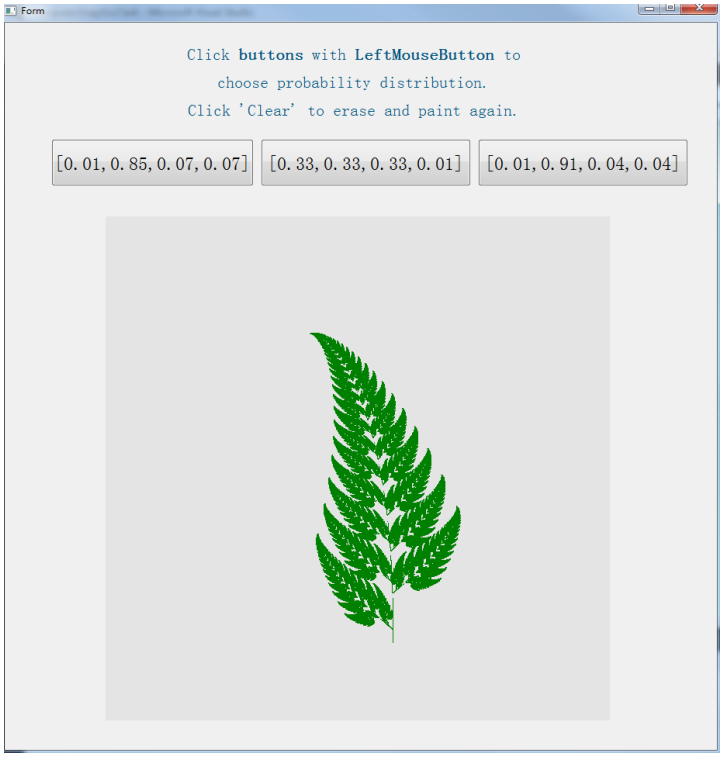
实验中的概率是由均匀分布的随机函数得到的。通过指定随机生成的数值的范围，就可以生成任意概率分布。例如：在[1，100]之间随机生成一个数，那么这个数落在[1，50]之间的概率就近似为 0.50。同理可得其他概率分布。

迭代算法如下：

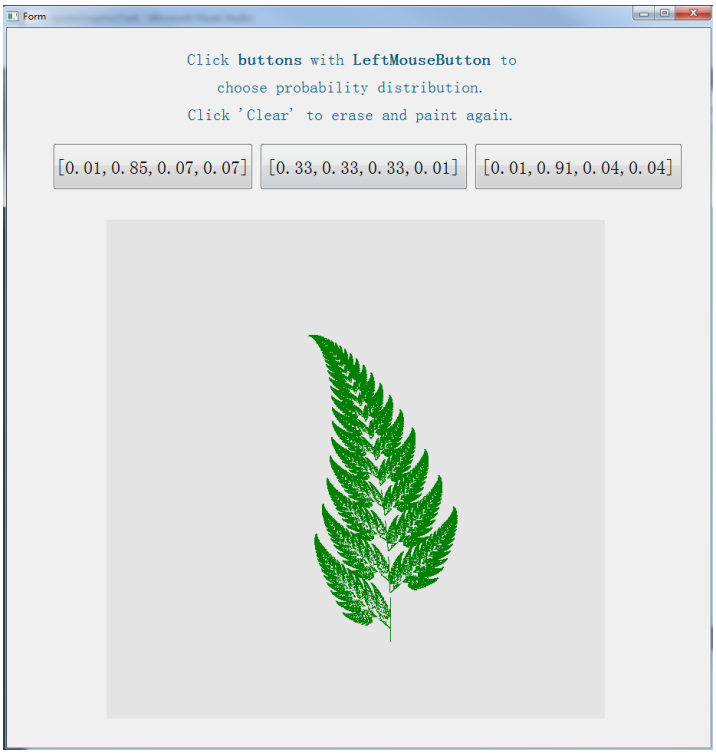
```
Initialize point(x, y);
Iterate(times)
{
    N = rand();
    判断 N 所在的概率区间;
    (x, y) = LearProjection(x, y);
    drawPoint(x, y);
}
```

3) 实验结果

实验结果如图 31 所示。



a. 概率分布[0.01,0.85,0.07,0.07]



b. 概率分布[0.33,0.33,0.33,0.01]

c. 概率分布[0.01,0.91,0.04,0.04]

图 31. 不同概率分布下的蕨类植物

通过点击不同的按钮，可以获得不同概率分布下的蕨类植物。比较发现，不同的概率分布并不会影响蕨类植物的整体形状，而会影响蕨类植物的局部点的疏密分布。生成植物的形状是由仿射变换的系数决定的。

4. 实验总结

通过本次计算机图形学作业，收获颇丰。

首先，学会就基本图元的生成算法、样条曲线的算法以及分形图形的生成算法，感慨于计算机图形学世界的美丽和多样性，更被创造这些算法的科学家们的智慧所折服。

其次，对 c++编程知识和技巧得到进一步掌握。实验中印象最深刻的是对 c++类成员函数的跨类调用。虽然我只在 Line 类中定义并实现了 BresenhamLine 函数进行画直线，但是在进行填充、样条曲线的生成时，同样要用到画直线的函数，这时候如果在不同的类中定义相同的 BresenhamLine 函数就会显得程序的扩展性不强。我首先想到的是:在定义这些类的时候对 Line 类进行继承，但是实现过程中发现，这样会导致 Qt 窗口的混叠。于是我采用第二种方案：在需要画直线的类中，都生命一个 Line 类的变量，并把 Line 类中的 BresenhamLine 函数声明为 public，于是就能在不同类中都调用 BresenhamLine 函数也不需要重复定义，显得更为简洁。