计算机图形学实验报告

孙韶言 SA12006051

一、 实验环境

本次实验使用了三种开发环境，分别为：

1. Windows 7 + Visual Studio 2010 + CImg Library[1]
2. Windows 8 + Mingw + CImg Library
3. Linux + g++ + CImg Library

以上各版本均测试通过，在所附代码的 ReadMe 文档中有具体说明。

CImg Library 是一个轻量级的开源 C++图像处理库，提供基本的图像 IO 操作、简单用户图形界面及交互、简单图形绘制操作。整个库只包括一个大小约 2Mb 的 C++头文件 CImg.h，使用该库只需添加一个头文件包含声明与一个使用命名空间声明：

```
#include "CImg.h"
using namespace cimg_library;
```

本次实验除了使用该库显示图形与进行简单的用户交互功能外，仅使用了其画点函数：

```
CImg< T >  & draw_point(const int x0, const int y0, const tc *const color, const float opacity = 1)
```

之所以选取 CImg Library 进行实验，是因为它不需要像 OpenCV 和 OpenGL 一样较多复杂的配置，不像 MFC 一样只支持 Windows 平台。其简洁原生的 C++语法比较适合自由定制工程架构，且其画点函数和一些简单的用户交互足够支持完成本次实验。

为进行实验搭建的工程采用了简单工厂的设计模式。每一个小实验内容都实现一个共同的接口：

```
class IGraphicsTask
{
public:
    virtual void executeTask();
};
```

用户在主菜单选择一个实验内容的时候，会生成一个相应的实验内容实例并执行其

```
virtual void executeTask();
```

函数以完成功能。这样做的好处是每个实验内容能够很容易地集成在整个项目中又保持

[1] 项目主页：http://cimg.sourceforge.net/

相互独立，非常利于扩展。工程运行时首先弹出主菜单，由于 CImg Library 不提供任何 GUI 控件，因此主菜单完全是绘制出来的（样式如图 1）。点击菜单中的每个条目可以打开相应内容的演示。

由于不支持中文显示，因此每个实验内容的说明都只有一句英文说明。关闭窗口或者按 ESC 键可以退出主窗口。

图 1 主菜单样式

二、 实验内容

(一) 图元的生成

1. 直线生成

直线生成的目标是在栅格显示器上确定最佳逼近直线的像素集合。数学上直线表示在平面上会有一定的覆盖范围，在栅格化的显示器上生成直线的过程中需要对这一覆盖范围进行栅格量化，确定需要绘制哪些像素点。图 2 给出了直线生成时的像素表示。

本次实验中采用的直线生成算法为计算机图形学领域使用最广泛的直线扫描转换算法：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，避免了乘除操作和浮点运算。

算法的具体解释为：

假设直线方程为：

$$y_{i+1} = y_i + k \times (x_{i+1} - x_i) = y_i + k, k = \mathbf{d}y/\mathbf{d}x$$

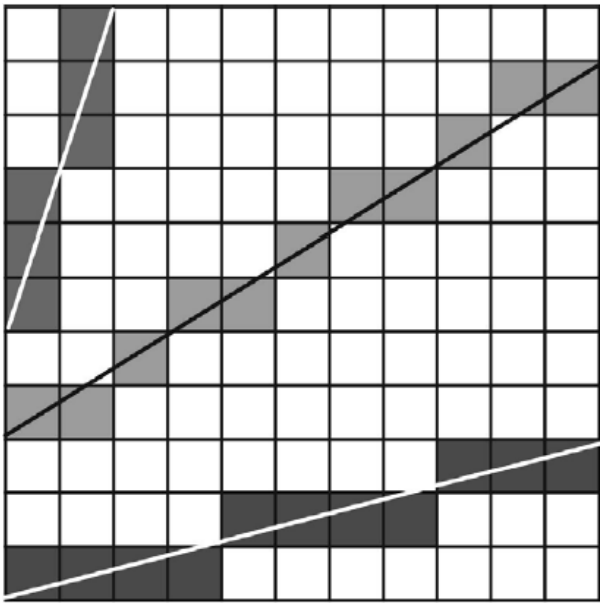

图 2 直线的像素表示

首先假定直线的斜率在 0 到 1 之间，其余的情况都可以通过交换坐标来实现。如果当前像素点为(x_i, y_i)，则下一个像素的横坐标为$x_i + 1$，所以纵坐标或者不变，或者递增 1 变为$y_i + 1$，而是否增加 1 取决于如图 3 所示误差项d的值。因为直线的起始点在像素中心，所以误差项初始值$d_0 = 0$。

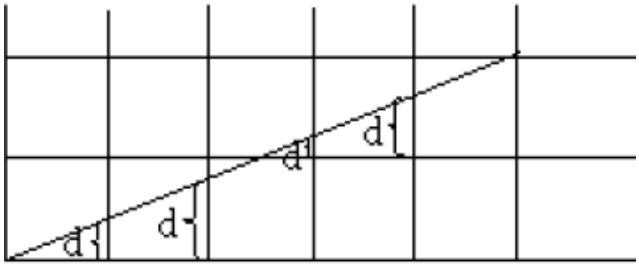

图 3Bresenham 算法误差项示意图

横坐标每增加 1，d相应增加直线的斜率k。一旦$d \geq 1$，就将其减去 1，保证d在 0 和 1 之间。当$d \geq 0.5$时，直线在下一列更接近当前像素右上方的像素(x_{i+1}, y_{i+1})，否则更接近当前像素右方的像素(x_{i+1}, y_i)。为方便计算，令$e = d - 0.5$，以其正负决定下一个绘制点的位置。为了避免浮点数运算，观察在决定下一个绘制点位置的时候只用到了e的正负符号，且$2\mathbf{d}x$总是正值，可以作如下替换：$e = e \times 2\mathbf{d}x$。此时有：

a) e的初始值为：$e_0 = -0.5 \times 2\mathbf{d}x = -\mathbf{d}x$

b) e的增量为：$k \times 2\mathbf{d}x = \frac{\mathbf{d}y}{\mathbf{d}x} \times 2\mathbf{d}x = 2\mathbf{d}y$

c) 当$e > 0$时，e的减量为：$1 \times 2\mathbf{d}x = \mathbf{d}x$

这样就得到了只包含整数加法运算的 Bresenham 算法：

Bresenham 算法：

输入：直线端点(x0, y0)，(x1, y1)
初始化：

```
dx = x1 - x0; dy = y1 - y0;
e = -dx;
x = x0; y = y0;
```

绘制：

```
for i = 0 ~ dx
    setpixel(x, y);
    x ++;
    e = e + 2 * dy;
    if e >= 0
        y ++;
        e = e - 2 * dx;
    end
end
```

实验结果展示如图 4：

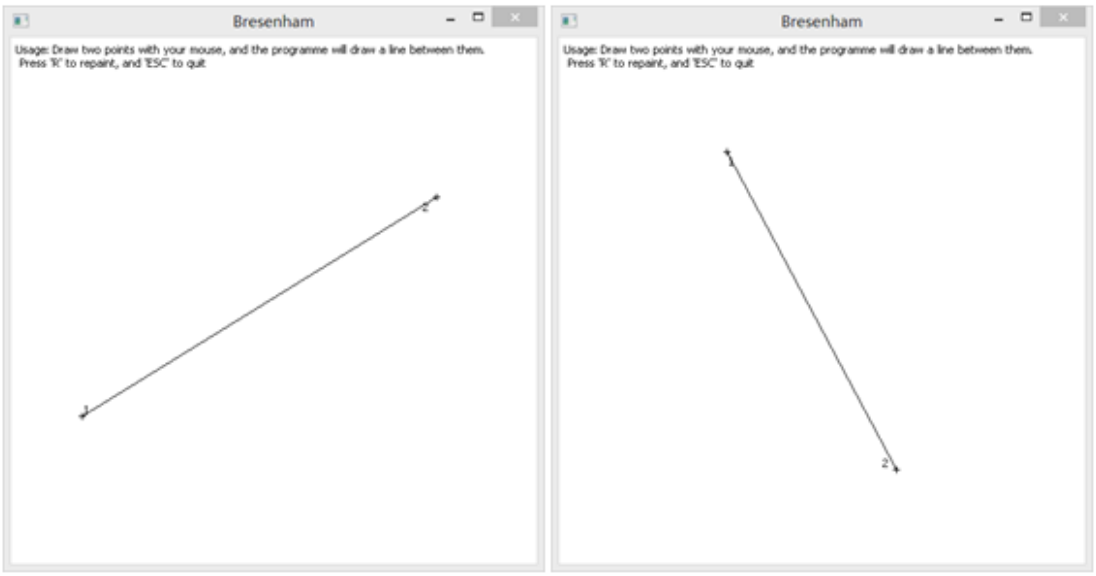

图 4Bresenham 算法直线绘制结果

操作说明：用鼠标在画板上依次点击要画直线的两个端点的位置，程序会给出 Bresenham 算法生成的连接两端点的直线。端点旁边的数字表示端点点击的顺序。在任何时候按 R 键可以清空画板进行重绘，按 ESC 键或者关闭窗口可以退出该实验内容，回到主菜单。截图损失了图像质量无法看清楚细节，事实上可以观察到绘制出的直线均匀、连续、细致，有很高的质量。

2. 椭圆生成

和直线生成类似，椭圆生成的目标也是将椭圆方程的曲线在栅格化的图形画板上进行最优的逼近显示。本次实验进行椭圆生成时采用的算法是使用较为广泛的中点椭圆绘制算法。图

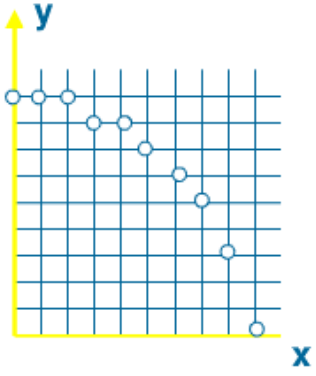

图 5 椭圆绘制示意图

5 给出了在栅格画板上的椭圆的一个部分示意图。

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绘制点的尾数是。构造椭圆函数为：

$$f(x,y)=r_y^2x^2+r_x^2y^2-r_x^2r_y^2$$

则有：

$$f(x,y)\begin{cases}<0,(x,y)\text{位于椭圆边界内}\\=0,(x,y)\text{位于椭圆边界上}\\>0,(x,y)\text{位于椭圆边界外}\end{cases}$$

根据图 6 可以判定：

a) 如果$f(M)<0$，说明点 M 在椭圆内，取点P_1进行绘制。

b) 如果$f(M)\geq 0$，说明点 M 在椭圆外，取点P_2进行绘制。

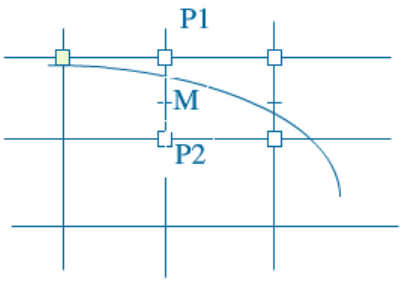

图 6 中点椭圆绘制算法决策示意图

由于椭圆关于横轴和纵轴都对称，因此在绘制的过程中只需要进行第一象限的绘制，其他象限取对称值即可。在第一象限内，又要分为斜率绝对值大于 1 和小于 1 的两个部分分别绘制。在斜率绝对值小于 1 的区域应该在x方向取单位量，否则应该在y方向取单位量。

中点椭圆算法的描述为：

中点椭圆绘制算法：

输入：椭圆的中心点(xc, yc)和长半轴、短半轴长度 rx, ry

初始化：

初始点 (x, y) = (0, ry);

第一区域绘制（斜率绝对值小于 1）：

```
初始决策值 p = ry^2 - rx^2*ry + 1/4 * rx^2;
for n = 0 ~ s.t. (2 * ry^2*x >= 2 rx^2ry)
    setpixel(x+xc, y+yc);
    setpixel(其他象限对应坐标);
    if p < 0
        x = x + 1;
        p = p + 2 * ry^2 * x + ry^2;
    else
        x = x + 1;
        y = y - 1;
        p = p + 2 * ry^2 * x - 2 * rx^2 * y + ry^2;
    end
end
```

第二区域绘制：

```
初始决策值 p = ry^2 * (x + 0.5)^2 + rx^2 * (y-1)^2 - rx^2 * ry^2;
for n = 0 ~ s.t. (y = 0)
```

```
    setpixel(x+xc, y+yc);
    setpixel(其他象限对应坐标);
    if p > 0
        y = y - 1;
        p = p - 2 * rx^2 * y + rx^2;
    else
        x = x + 1;
        y = y -1;
        p = p + 2 * ry^2 * x - 2 * rx^2 * y + rx^2;
    end
end
```

实验结果如图 6 图 7：

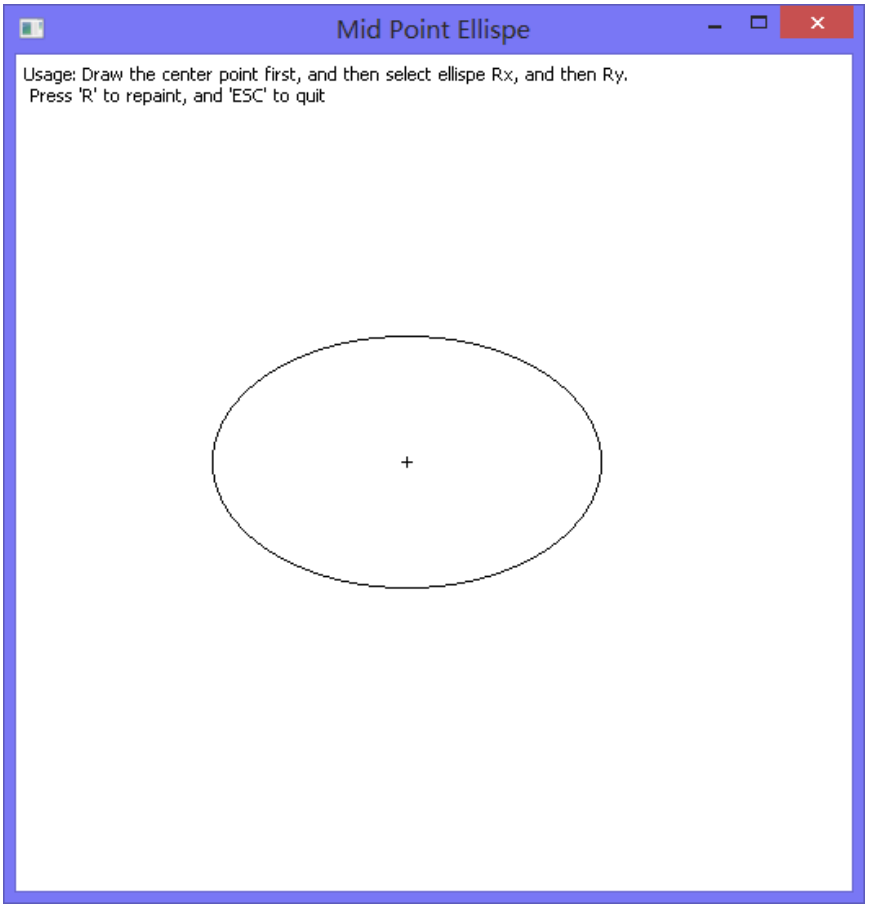

图 7 椭圆绘制结果

操作说明：首先在画板上点击一个点作为椭圆的中心，然后画板上会出现两条跟随光标的纵向的直线，用来限定椭圆横轴的半径。在合适的位置点击鼠标，会出现两条固定的短线标记横轴半径的位置，此时跟随光标的是两条水平的直线，用来限定椭圆纵轴的半径。在合适的位置点击鼠标，程序就会根据以上输入生成椭圆并显示出来。根据实验结果可以看出，画出的椭圆非常细致，质量很高。

3. 区域填充

区域填充的目标是给定一个封闭的区域，填充位于这一区域内部的每一个像素。本次实验采用的算法是改进的边界填充算法。最基本的边界填充算法的思路是，从区域的一个内部点开始，由内向外绘制点直到边界。在使用基本边界填充算法进行 4-连通区域填充的过程中，主要算法思想是：

基本边界填充算法

输入：封闭边界，区域内一点(x, y)

绘制过程：

```
function boundaryFill4(x, y)
    if (x, y) 不是边界点
        setpixel(x, y);
        boundaryFill4(x + 1, y);
        boundaryFill4(x - 1, y);
        boundaryFill4(x, y + 1);
        boundaryFill4(x, y - 1);
    end
end
```

这种算法虽然很简单，但是需要大量的堆栈执行递归过程，一方面降低了运算速度，以方便需要很大的堆栈空间，在使用这种算法进行实验时，当要绘制的点数比较多的时候，就经常会报栈溢出的异常。为了节省堆栈空间，实验中采用的方法是：沿扫描线填充水平像素区段，仅将每个水平像素区段的起始位置放进堆栈。从初始内部点开始，首先填充该像素所在扫描行的连续像素区段，然后将相邻扫描线上各段的起始位置放进堆栈，然后递归进行绘制。

改进的边界填充算法

输入：封闭边界，区域内一点(x, y)

绘制过程：

```
function boundaryFill4_improved(x, y)
    if (x, y) 未被绘制
        向左绘制该行直到边界，获得最左边的起点(startX);
        向右绘制该行直到边界，获得最右边的终点(endX);
        for i = startX ~ endX
            if (i, y + 1) 未被绘制
                boundaryFill4_improved(i, y + 1);
            end
            if (i, y - 1) 未被绘制
                boundaryFill4_improved(i, y - 1);
            end
        end
    end
end
```

实验结果如图 8：

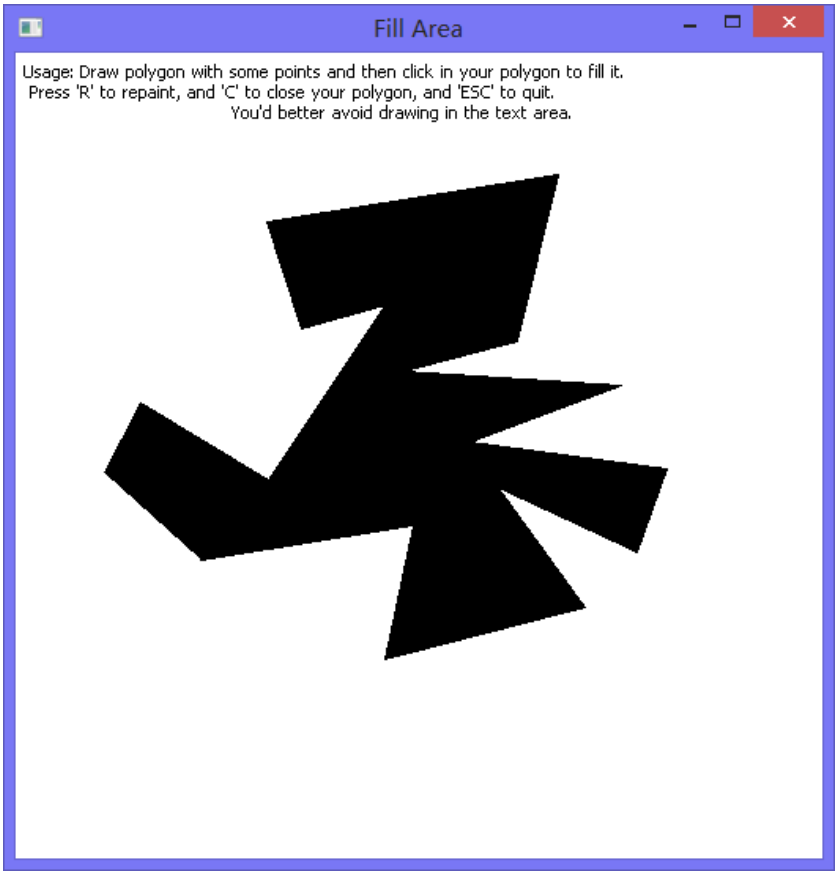

图 8 边界填充结果

操作说明：首先通过点击鼠标确定任意多边形的各个顶点，在最后一步需要封闭多边形的时候按下 C 键，程序会自动封闭该多边形。此时在多边形内部任意一点点击鼠标，就能够将多边形内部填充完毕。理论上该算法不仅支持多边形的区域填充，同样也适用于任意形状的多边形的区域内部填充。

(二) 样条曲线的生成

1. Bezier 曲线生成

Bezier 曲线是一种性质非常好的样条曲线，它的混合函数是 Bernstein 多项式：

$$BEZ_{k,n}(u) = C(n,k)u^k(1-u)^{n-k}$$

其中

$$C(n,k) = \frac{n!}{k!\,(n-k)!}$$

而 Bezier 曲线则可以表示为：

$$x(u) = \sum_{k=0}^{n} x_k BEZ_{k,n}(u)$$

$$y(u) = \sum_{k=0}^{n} y_k BEZ_{k,n}(u)$$

其中$\mathbf{p}_k = (x_k, y_k), k = 0,1,\ldots,n$是 Bezier 曲线的控制点。Bezier 曲线的一个性质是将每两个相邻控制点的中点取出来，它们仍然是 Bezier 曲线的控制点。这样就可以很容易得到一种递归绘制算法，即 Casteljau 算法。该算法的基本思想是一级一级地取控制点的中点作为新的控制点，直到所有控制点都相邻了，递归结束，最终的控制点一定落在所要绘制的 Bezier 曲线上。现在首先给出这种算法的描述和实验结果：

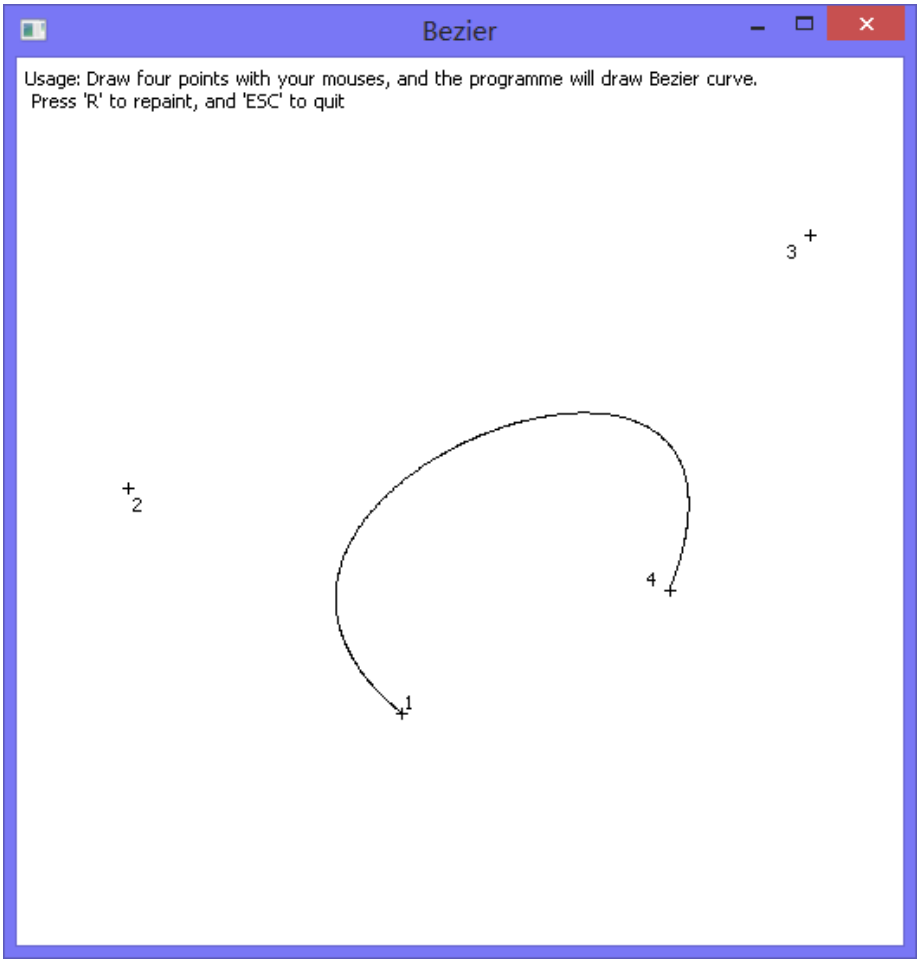

图 9Bezier 实验结果 1

Casteljau 算法一

输入：控制点(x[], y[]) = (xk, yk), k = 0, ..., n - 1

绘制过程：

```
function casteljau1(x, y)
    if 控制点全部相邻
        setpixel(xk, yk), k = 0 ~ n - 1;
    else
        (x_new1[], y_new1[]) = (x[0], y[0]), (x[ceil(n/2)],
y[ceil(n/2)]) 与 前半部分中点;
        (x_new2[], y_new2[]) = (x[floor(n/2)], y[floor(n/2)]),
(x[n-1], y[n-1]) 与 后半部分中点;
        casteljau1(x_new1, y_new1);
```

```
            casteljau1(x_new2, y_new2);
        end
    end
```

采用这种算法得到的实验结果如图 9。

根据实验结果可以看出，这样得到的 Bezier 曲线比较光滑，但是存在一个问题，在一些细节的地方会出现重影（图 10），这是因为该算法的收敛条件是相邻的控制点互相连通，这就很容易导致几个控制点聚积在了一起，导致线条在一些地方显得比较臃肿。

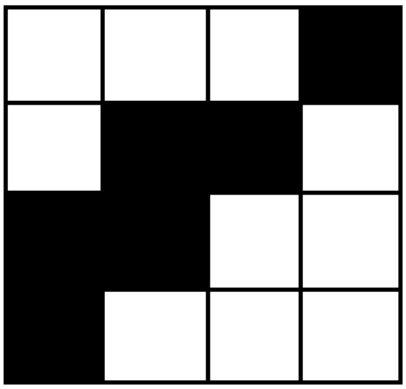

图 10Casteljau 算法出现的重影

这种问题可以通过控制递归的层数得到一定程度的解决。基本思路是避免递归一直进行下去直到聚积到一起，当递归达到一定层数的时候，就不再进行递归，如果此时相邻的控制点没有连通，就在两个相邻控制点之间通过一个线段连接起来。这种处理可以一定程度上解决重影问题。

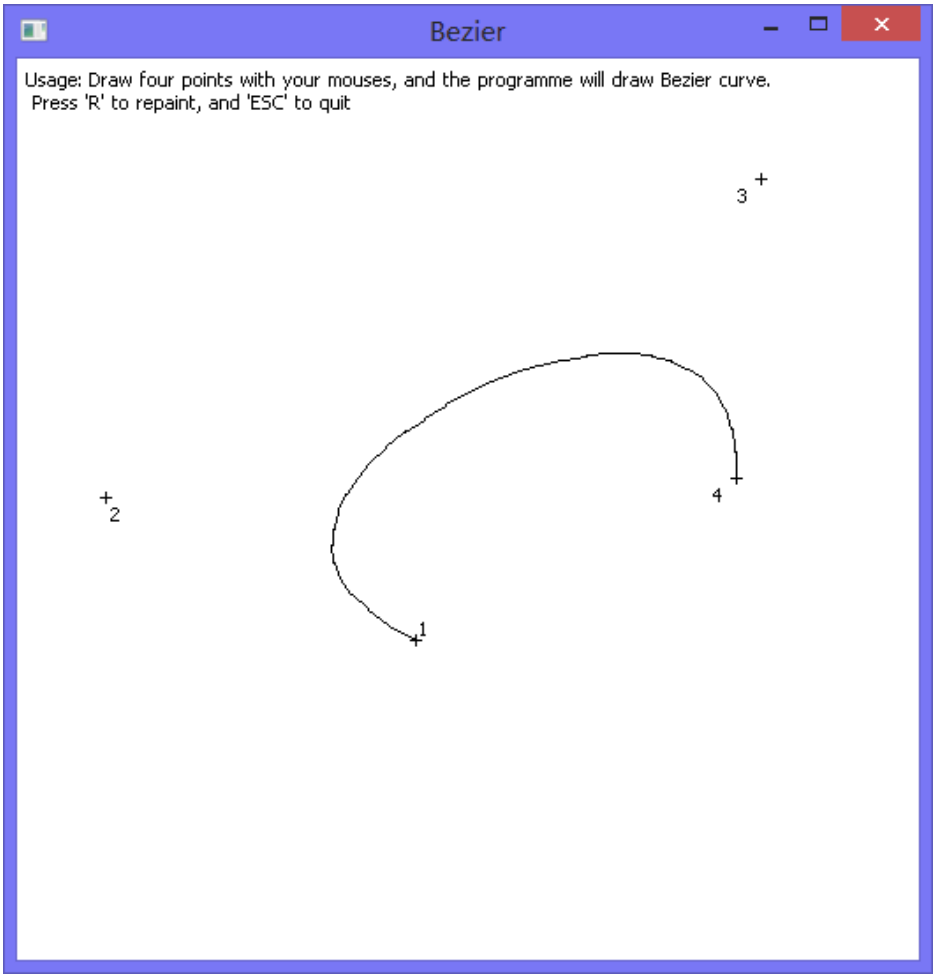

图 11Bezier 实验结果 2

Casteljau 算法二

输入：控制点(x[], y[]) = (xk, yk), k = 0, ..., n - 1

绘制过程：

```
casteljau2(x, y ,0);
function casteljau1(x, y, level)
    if 控制点全部相邻
        setpixel(xk, yk), k = 0 ~ n - 1;
    else if level == MAXLEVEL
        for i = 0 ~ n - 1
            drawline((x[i], y[i]), (x[i+1], y[i+1]));
        end
    else
        (x_new1[], y_new1[]) = (x[0], y[0]), (x[ceil(n/2)],
y[ceil(n/2)]) 与 前半部分中点;
        (x_new2[], y_new2[]) = (x[floor(n/2)], y[floor(n/2)]),
(x[n-1], y[n-1]) 与 后半部分中点;
        casteljau2(x_new1, y_new1, level + 1);
        casteljau2(x_new2, y_new2, level + 1);
    end
end
```

根据实验结果图 11 可以看出，这种解决方案基本消除了重影的问题。但是这种方案也有一定的问题，因为 MAXLEVE 的选取影响实验的结果，如果 MAXXLEVE 取得过大，就又退化为第一种算法，如果 MAXLEVEL 选取得过小，则会使得曲线上很多地方是由线段连接起来的，因而使得曲线不够光滑。

最后，在实验中又尝试了另外一种思路。

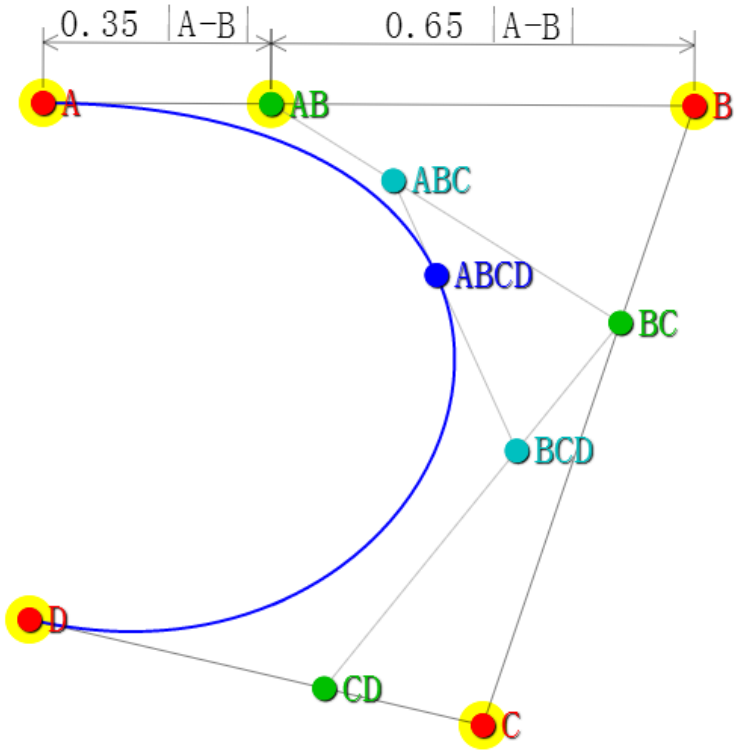

图 12 遍历法绘制 Bezier 曲线

如图 12，A、B、C、D 是 Bezier 曲线的四个控制点，点 AB、BC、CD 分别是它们按一定

比例的分割点；点 ABC、BCD 又分别是 AB、BC 的同比例分割点和 BC、CD 的同比例分割点；最后，ABCD 是点 ABC、BCD 的同比例分割点，而这一点就在所要绘制的 Bezier 曲线上。那么，当 AB 在线段 A-B 上从 A 点移动到 B 点的时候，ABCD 点的轨迹就是要绘制的 Bezier 曲线。算法描述为：

遍历法绘制 Bezier 曲线：

输入：控制点 A, B, C, D

绘制过程：

```
STEP = 0.01;
oldPoint = A;
for u = 0.0 : STEP : 1 - STEP
    AB = A + u * (B - A);
    BC = B + u * (C - B);
    CD = C + u * (D - C);
    ABC = AB + u * (BC - AB);
    BCD = BC + u * (CD - BC);
    ABCD = ABC + u * (BCD - ABC);
    drawline(oldPoint, ABCD);
    oldPoint = ABCD;
end
```

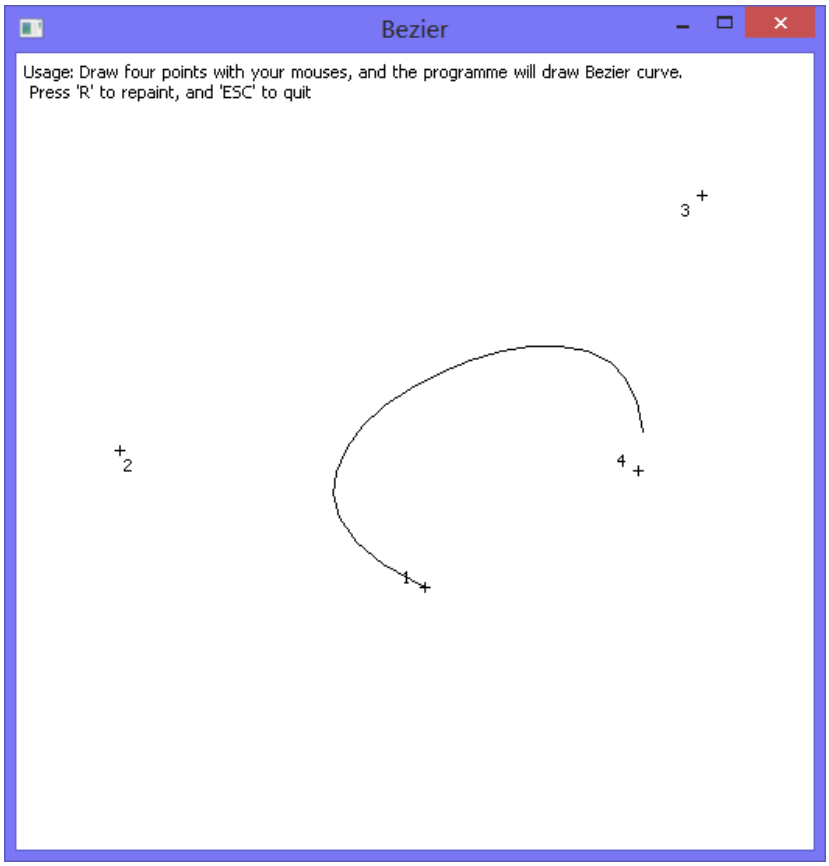

图 13Bezier 实验结果 3

这时的绘制结果如图 13，可以看出此时已经没有了重点现象，同时也没有很强的分段效应，结果比较理想。在算法实现的时候调用的画线函数是之前实现的 Bresenham 算法。

2. B-样条曲线生成

B-样条是一种使用更为广泛的逼近样条，它的多项式次数独立于控制点的数目，且允许局部控制曲线或曲面 ，这时 Bezier 曲线所没有的。对 B-样条曲线，同样可以写出基于混合函数的表达式：

$$\mathbf{P}(u)=\sum_{k=0}^{n}\mathbf{p}_k N_{k,d}(u),t_{min}\le t\le t_{max},2\le d\le n+1$$

混合函数满足 Cox-deBoor 递推公式，即：

$$\mathrm{N_{i,1}}(t)=\begin{cases}1, & \mathrm{t_i}\le t\le t_{i+1}\\ 0, & others\end{cases}$$

$$\mathrm{N_{i,k}}(t)=\frac{t-t_i}{t_{i+k-1}-t_i}N_{i,k-1}(t)+\frac{t_{i+k}-t}{t_{i+k}-t_{i+1}}N_{i+1,k-1}(t)$$

对于 B-样条曲线的绘制，实验中采用的是 Deboor 算法。

先将 t 固定在区间$[\mathrm{t_j},\mathrm{t_{j+1}}](\mathrm{k}-1\le\mathrm{j}\le\mathrm{n})$，由 Cox-deBoor 公式有：

$$P(t)=\sum_{i=0}^{n}P_iN_{i,k}(t)=\sum_{i=j-k+1}^{j}P_iN_{i,k}(t)$$

$$=\sum_{i=j-k+1}^{j}P_i\left[\frac{t-t_i}{t_{i+k-1}-t_i}N_{i,k-1}(t)+\frac{t_{i+k}-t}{t_{i+k}-t_i}N_{i+1,k-1}(t)\right],t\in\left[t_j,\mathrm{t_{j+1}}\right]$$

$$=\sum_{i=j-k+1}^{j}[\frac{t-t_i}{t_{i+k-1}-t_i}P_i+\frac{t_{i+k-1}-t}{t_{i+k-1}-t_i}P_{i-1}]N_{i,k-1}(t)$$

令

$P_i^{[r]}(\mathrm{t})$

$$=\begin{cases}P_i, & r=0;i=j-k+1,j-k+2,\dots,j\\ \dfrac{t-t_i}{t_{i+k-r}-t_i}P_i^{[r-1]}(t)+\dfrac{t_{i+k-r}-t}{t_{i+k-r}-t_i}P_{i-1}^{[r-1]}(t), & r=1,2,\dots,k-1;i=j-k+r+1,j-k+r+2,\dots,j\end{cases}$$

则有

$$P(t)=\sum_{i=j-k+1}^{j}P_iN_{i,k}(t)=\sum_{i=j-k+2}^{j}P_i^{[1]}(t)N_{i,k-1}(t)$$

上式是同一条曲线 P(t)从 k 阶 B 样条表示到 k－1 阶 B 样条表示的递推公式，复用此公式，得到$\mathrm{P(t)}=\mathrm{P_j^{[k-1]}(t)}$。

于是，P(t)的值可以通过递推关系求得，这就是著名的 De Boor 算法。De Boor 算法的递推关系如图 14。

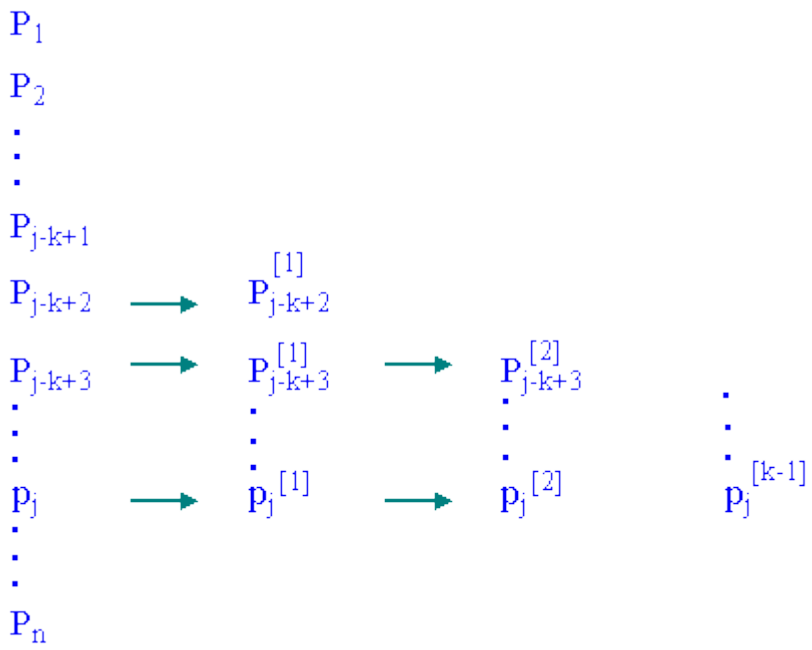

图 14Deboor 算法递推关系图

实验的结果如图 15。

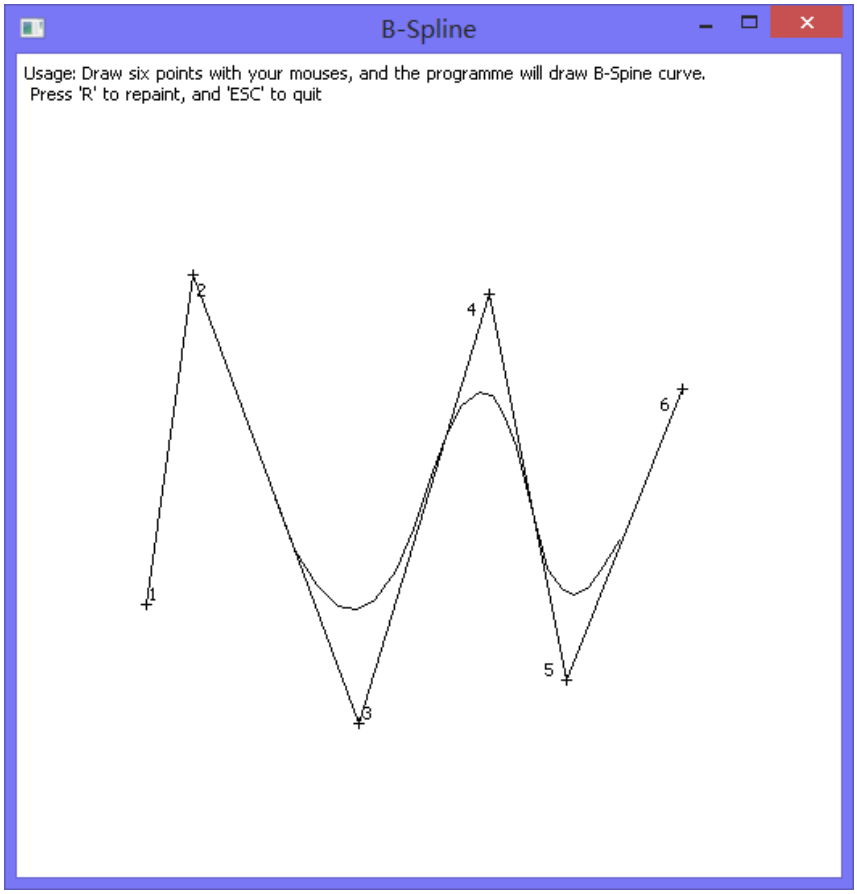

图 15B-样条实验结果

操作说明：在画板上用鼠标点六个点，程序会绘制出控制多边形和相应的 B-样条曲线。同样在任何时候按 R 键可以进行重绘。绘制出的 B-样条曲线有一些小的棱角，原因是在递归的过程中，取整操作损失一定的精度，另外在两点之间进行连线时，也会导致一定的棱角

出现。

(三) 分形图形的生成

1. Mandelbrot 集生成

Mandelbrot 集合是在复平面上组成一种分形的点的集合，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$z = 0$开始对$f_c(z)$进行迭代。序列$[0, f_c(0), f_c(f_c(0)), f_c\left(f_c(f_c(0))\right), \ldots]$的元素的模要么延伸到无穷大，要么停留在有限半径的圆盘内。而 Mandelbrot 集合就是使该序列停留在有限圆盘内的所有c的集合。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”。

在绘制 Mandelbrot 分形的时候，基本的思想就是对区域范围内的每一个点，通过判断它的相应序列$[0, f_c(0), f_c(f_c(0)), f_c\left(f_c(f_c(0))\right), \ldots]$的模来判断它是否在 Mandelbrot 集之内。但是由于不可能将序列延伸到无穷，可以只进行有限次迭代，实验中是进行 100 次迭代，如果序列前 100 个元素都满足模不大于 2，则认为该点在 Mandelbrot 集合之内。

对 Mandelbrot 集最简单的着色方案是：如果点在集合以内，则给这一点着色为黑色，否则着色为白色。但是为了让生成的分形图案更好看，一般会根据迭代的值将图形上的每一点着为彩色。假设一个点在集合之外，且判定这一结论时已经进行迭代的次数为 k，实验中的着色方案为：

a) 如果点在集合以内，则着色为黑色。

b) 如果 k <= 50，R = k*4 + 50，G = k * 4 + 50，B = k * 3 + 100。

c) 如果 k > 50，R = k + 150，G = 0，B = k + 150。

为了能够看清楚更细节的地方，实验中允许对局部进行放大显示。根据局部选框的位置和大小，可以用同样的算法计算新的区域中每一个点如何进行着色，并进行绘制。

实验得到的整体的图形如图 16，在一个局部进行放大后得到的结果如图 17。由于计算时仅做到了 100 次的迭代，因此得到的图形的细节是有限的，并不能无限放大，当放大到一定程度时，就无法看到更细的细节了。

操作说明：程序运行时会首先显示整体的 Mandelbrot 集的形状。之后可以用鼠标在一个区域上画出一个方框，程序会更新放大被框住的区域。需要注意的是，由于 CImg Library 提供的用户交互功能是有限的，在画框的时候，所画的框是不显示的。

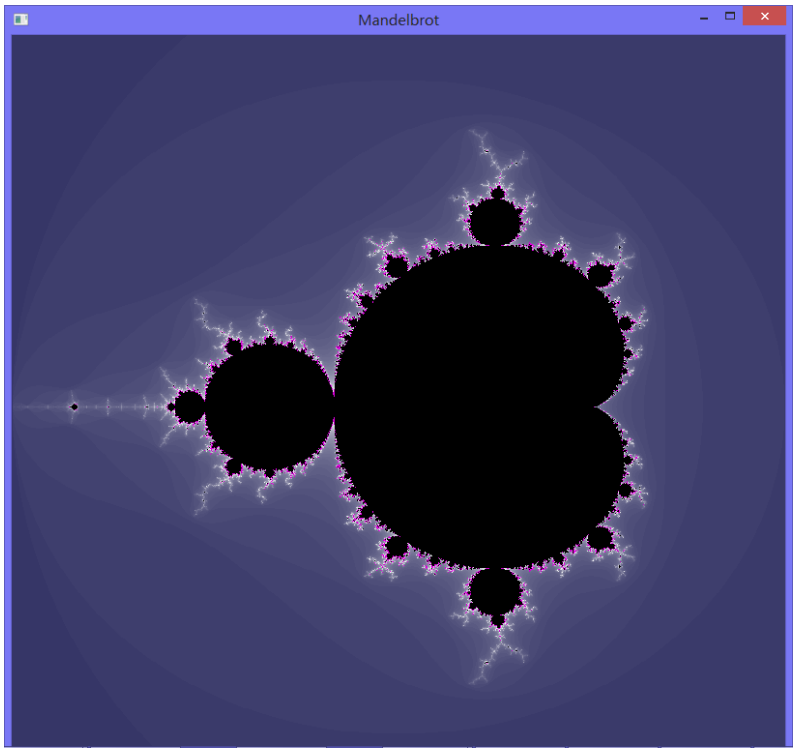

图 16mandelbrot 集整体形状

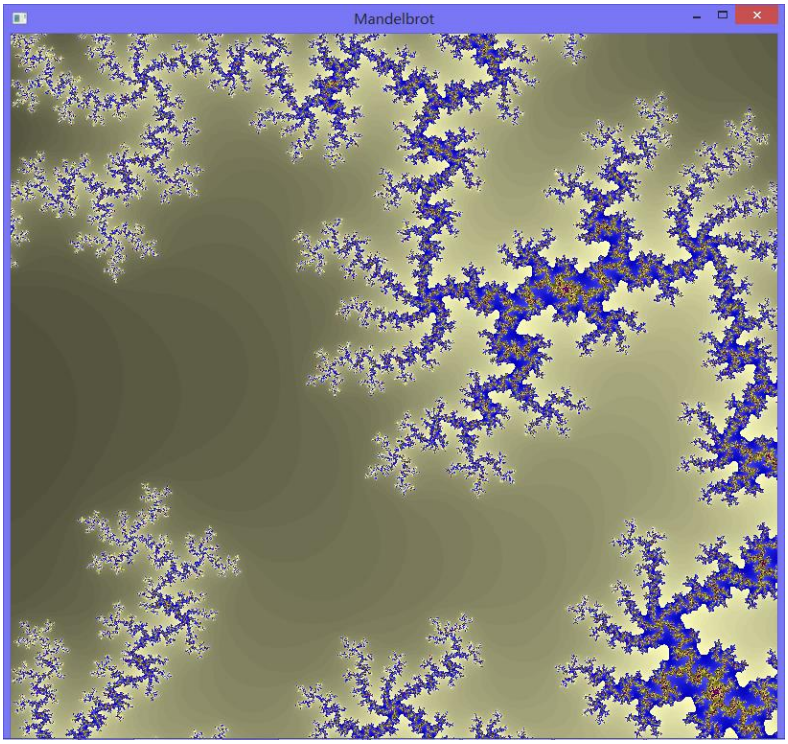

图 17mandelbrot 集一个局部细节

2. 蕨类植物生成

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：

$$x' = ax + by + c$$
$$y' = cx + dy + f$$

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$w_1, w_2, w_3, \ldots$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：

$$w_1 \begin{cases} x' = a_1 x + b_1 y + c_1 \\ y' = d_1 x + e_1 y + f_1 \end{cases} \rightarrow p_1$$
$$w_2 \begin{cases} x' = a_2 x + b_2 y + c_2 \\ y' = d_2 x + e_2 y + f_2 \end{cases} \rightarrow p_2$$
$$w_3 \begin{cases} x' = a_3 x + b_3 y + c_3 \\ y' = d_3 x + e_3 y + f_3 \end{cases} \rightarrow p_3$$

生成 IFS 分形的算法描述为：

IFS 分形算法

输入：仿射变换系数矩阵 m[n][6]，概率向量 p[n]

初始化：

```
N_iter = 100000;
```

绘制过程：

```
while N_iter > 0
    cp = p[0] * 100;
    t = rand() % 101;
    if t <= cp
        [a, b, c, d, e ,f] = m[0]{*};
    else
        for i = 1 ~ n-1
            cp += p[i] * 100;
            if t <= cp
                [a, b, c, d, e, f] = m[i]{*};
                break;
            end
        end
    end
    x' = a * x + b * y + e;
    y' = c * x + d * y + f;
    setpixel(x', y');
    N_iter --;
end
```

有了 IFS 分形生成算法，只需要给出仿射变换系数矩阵 m 和概率向量 p，即可绘制出相应的分形图案来。实验中共选取了 4 种分形图案的参数进行绘制，分别是：

```
double m1[] =
{0.06,0,0,0.6,0,0,0.04,0,0,-0.5,0,1,0.46,-0.34,0.32,0.38,0,0.6,0.48,0
.17,-0.15,0.42,0,1,0.43,-0.26,0.27,0.48,0,1,0.42,0.35,-0.36,0.31,0,0.
8};
double p1[6] = {0.1,0.1,0.1,0.23,0.23,0.24};

double m2[]
={0,0,0,0.16,0,0,0.85,0.04,-0.04,0.85,0,1.6,0.2,-0.25,0.23,0.22,0,1.6
,-0.15,0.28,0.26,0.24,0,0.44};
double p2[] = {0.01, 0.85, 0.07, 0.07};

double m3[] =
{0.05,0,0,0.4,0,0,0.5,0,0,0.5,0,0.35,0.3536,0.3536,-0.3536,0.3536,0,0
.15,0.433,-0.25,0.25,0.433,0,0.15};
double p3[] = {0.0259,0.3247,0.3247,0.3247};

double m4[] =
{0.001,0,0,0.25,0,0,0.8488,0.04344,-0.04449,0.8289,0,1,0.1,0.2165,-0.
1732,0.125,0,0.6,0.1,-0.2165,0.1732,0.125,0,0.6};
double p4[] = {0.01, 0.87, 0.06, 0.06};
```

绘制的结果如图 18。

图 18IFS 分形

(四) 真实感图形的生成

1. 百合花生成

为了生成一幅具有真实感的图形，需要考虑很多方面的内容。首先需要选定一个几何造型，在绘制该几何造型的时候，需要考虑到消隐、光照以及物体表面的纹理等因素。

图 19 真实的百合花

百合花的几何结构分为两个主要部分：花瓣和花蕊（参照图 19）。现在分别讨论这两种结构。

a) 花瓣

百合花的花瓣可以由简单的四边形变形而来。在x,y方向的变形函数分别为：

$$\Delta x = 0.6a_x u\sin(\pi v + 1)$$

$$\Delta y = a_x(v - 0.5)\sin((u + 0.5)\pi)$$

进一步地，为了显示出褶皱效果，修改变形函数为：

$$\Delta x = 0.6a_x u\sin(\pi v + 1) + 0.05a_x u\sin(0.3a_x v\pi)$$

$$\Delta y = a_x(v - 0.5)\sin\big((u + 0.5)\pi\big) - 0.05a_x(v - 0.5)\sin(10(u + 0.5)\pi)$$

为了给花瓣添加一定的纹理，以及看起来比较自然的扭曲，可以增加变形函数：

$$g = a_x\sin\left(\frac{\pi y}{b_y} + 3\right) + \frac{10x^3}{a_x^3}\sin(0.1\pi y) + 0.15a_x(1 - 2|v - 0.5|)(1 - 2|u|)\sin(24(u + 0.5)\pi) + 0.1a_x e^{-25a_x(u-0.05)^2} + 0.1a_x e^{-25a_x(u+0.05)^2}$$

上式中第一项控制全局的凹凸变形，第二项控制局部边界凹凸变形，第三项控制全局的纹理，第四项与第五项控制局部的纹理。

一种百合花的花瓣颜色是如图 19 的红色，实验中取的色调$H = 340°$，其中间轴颜色较深，两边较浅，接近白色。接近花蕊的部位颜色更深，因此设置亮度为：

$$I = 500|u| + 200v, u \in [-0.5,0.5], v = [0,1]$$

饱和度设置为0.5。另外在接近花蕊的部位有一些随机的深色的红点，另其饱和度为0.8。

为了模拟真实的光照，采用了简单的光照模型。首先计算平面法向量$\vec{n} = [n_x, n_y, n_z]$：

$$a = (y_1 - y_0) \times (z_2 - z_0) - (y_2 - y_0) \times (z_1 - z_0)$$
$$b = (z_1 - z_0) \times (x_2 - x_0) - (z_2 - z_0) \times (x_1 - x_0)$$
$$c = (x_1 - x_0) \times (y_2 - y_0) - (x_2 - x_0) \times (y_1 - y_0)$$
$$n_x = \frac{a}{\sqrt{a^2 + b^2 + c^2}}$$
$$n_y = \frac{b}{\sqrt{a^2 + b^2 + c^2}}$$
$$n_z = \frac{c}{\sqrt{a^2 + b^2 + c^2}}$$

假设光照方向为$v = [v_x, v_y, v_z]$，则平面法向量与光照方向的夹角θ的余弦值为：

$$\cos(\theta) = \frac{v_x n_x + v_y n_y + v_z n_z}{\sqrt{v_x^2 + v_y^2 + v_z^2}}$$

设漫反射系数为k，环境光光强为g，则最终的亮度值为：

$$I_f = I \times k \times \cos(\theta) + g$$

通过这样一系列处理，最终得到的花瓣效果如图 20。

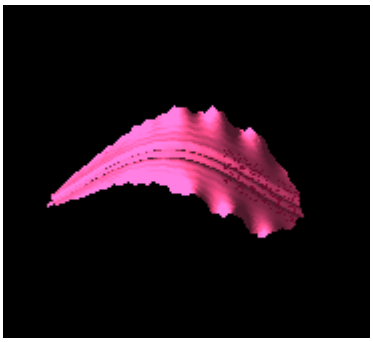

图 20 百合花花瓣

b) 花蕊

百合花的雄蕊可以看作是由一个圆柱和一个椭球的组合变形而来。

为了使得花蕊看起来更加自然，对它们进行一定的变形：

$$x = r\cos(u) + A\sin(2\pi v)$$
$$y = hv$$
$$z = r\sin(u) + A\sin(2\pi v)$$

其中h为圆柱的高度，A为变形的幅度。

雄蕊和雌蕊的差别在于顶端的颜色和形状，雌蕊的顶端偏向于紫色，且更接近于半椭球形。

其中椭球面的计算：

$$x = x_0 + a\cos(u)\cos(v)$$
$$y = y_0 + b\cos(u)\sin(v)$$
$$z = z_0 + c\sin(u)$$

圆柱面的计算：

$$x = r\cos(2\pi v)$$
$$y = hu$$
$$z = r\sin(2\pi v)$$

为了绘制整个图形，首先讨论曲面的绘制方法。在绘制曲面的时候，都可以把整个曲面看作是很多小的曲面贴片的组合。由于百合花每个结构都是由规则的形状变形而来，都可以通过u, v两个参数对整个曲面进行遍历。

为了处理消隐，实验中采用了 Z 缓冲器的算法，该算法的基本思想是对屏幕上的每一个像素，记录下位于此像素内最靠近观察者一个像素的深度值和颜色。引平行于 Z 轴的射线交物体表面于一个点的集合，取出最靠近观察者的点，存放其深度和颜色信息。具体算法描述为：

Z 缓冲器算法：

初始化：

```
each Z = -MAX;
```

绘制过程：

```
for 多边形面
    计算多边形平面方程系数(a, b, c, d);
    for 扫描线
        for 扫描线上多边形中所有像素
            求像素深度值 z;
            if
                z > Z[this point];
                Z[this point] = z;
                setpixel(this point);
            end
        end
    end
end
```

实验最终得到的百合花的绘制效果如图 21。和真实的百合花图片（图 19）相比较，实验得到的绘制效果有着和真实百合花非常类似的形状结构，颜色上有一定的偏差。

图 21 百合花绘制效果图

2. 西瓜生成

相比于百合花，西瓜有着更简单的几何造型。基本上西瓜可以用一个椭球来表示。在实验中，西瓜的造型为：

$$x = x_0 + 80\cos(u)\cos(v)$$
$$y = y_0 + 80\cos(u)\sin(v)$$
$$z = z_0 + 100\sin(u)$$

为了显得自然一些，对西瓜进行了一定的旋转：x方向旋转50°，z方向旋转20°。

西瓜表面上有条纹，可以通过改变颜色的饱和度来实现。条纹沿西瓜的v参数方向周期出现，用周期函数

$$S = 0.8|\sin(Bv)|$$

控制表面的饱和度条纹纹理出现的概率，实验中取$B = 5$。由于条纹不光滑，可以对其沿参数u方向进行周期性的干扰，控制表面颜色的饱和度函数变为

$$S = 0.8|\sin(Bv + Asin(Cu))| + 0.4$$

实验中选取$C = 40, A = 0.15$。

这样得到的条纹是渐变的，而实际的西瓜条纹边界是清晰的，因此对上述函数进行二值处理：

$$S = \begin{cases} 0.4, S \leq 0.4 \\ 0.8, S > 0.4 \end{cases}$$

按照和百合花类似的面绘制算法和光照模型，绘制出的西瓜如图 22。

图 22 西瓜绘制效果图

三、 实验总结

本次实验实现了包括直线生成、椭圆生成、区域填充、Bezier 曲线生成、B-样条曲线生成、Mandelbrot 集分形实现、迭代函数系统分形实现、真实感图形（百合花、西瓜）生成在内的计算机图形学基本算法。通过实验深化了对各种算法的理解，同时通过实际动手真切感受到计算机图形学引人入胜的魅力。