实验3　交互式绘制多边形

3.1　实验目的

（1）掌握双缓冲绘图技术。

（2）掌握人机交互技术。

（3）掌握填充动态多边形的有效边表算法。

3.2　实验要求

（1）使用鼠标在屏幕客户区绘制任意点数的多边形。要求使用橡皮筋技术动态绘制每条边；鼠标移动过程中按住 Shift 键时可绘制垂直边或水平边；将多边形的终点移动到多边形的起点时自动封闭多边形；在绘制多边形的过程中，状态栏动态显示鼠标光标的位置坐标。

（2）当开始绘制多边形时，更改鼠标光标为十字光标，多边形绘制完毕后恢复为箭头光标。

（3）多边形闭合后自动调用有效边表算法填充多边形内部区域。

3.3　效果图

交互式绘制多边形效果如图 3-1 所示。

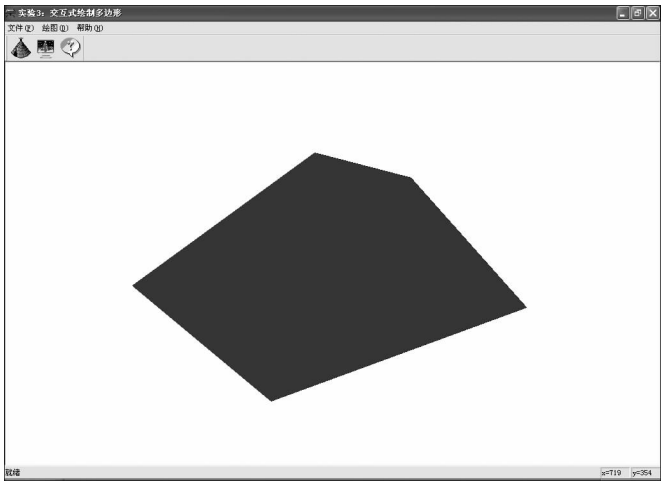

图 3-1　交互式绘制多边形效果图

3.4 实验准备

(1) 在学习完主教材第 4.2 节后进行本实验。

(2) 熟悉 Visual C++ 中 CPtrArray 类的使用方法。

(3) 熟悉静态多边形的有效边表填充算法。

3.5 实验步骤

本实验由三部分构成。第一部分是人机交互技术绘图,使用双缓冲技术实现。第二部分是创建动态数组,使用 MFC 的 CPtrArray 类数组实现。第三部分是填充多边形,使用有效边表算法实现。

3.5.1 人机交互技术

人机交互绘图技术主要包括回显、约束、网格、引力域、橡皮筋、拖动、草拟和旋转等技术,这些技术协调使用可以完成图形的交互操作。本实验主要使用了橡皮筋、回显、约束和引力域技术。

(1) 橡皮筋技术。橡皮筋技术是将绘图过程动态、连续地表现出来,直到产生用户满意的结果为止的技术。实验中借助于双缓冲技术,在 WM_MOUSEMOVE 消息处理函数中实现了橡皮筋技术。

(2) 回显技术。回显技术就是将对图形的交互操作用某种方式表达出来的技术。实验中进行了状态栏编程,当移动鼠标绘制多边形时,在状态栏上显示鼠标光标当前位置的坐标,方便用户确定多边形的顶点。在 WM_MOUSEMOVE 消息处理函数中实现了回显技术。

(3) 约束技术。约束技术就是在绘图过程中对图形的方向、形状进行控制的技术。本实验在 WM_MOUSEMOVE 消息的处理函数中使用了约束技术。移动鼠标绘制多边形每条边的过程中,如果同时按 Shift 键,则根据鼠标移动的方向确定所绘制的边为水平边或垂直边。

(4) 引力域技术。引力域技术是如何使用鼠标将一条边的终点准确地连接到另一条已经画好的某一条边的顶点上的技术。实验中在闭合多边形时使用了引力域技术。使用鼠标闭合多边形时,准确地将多边形的终点放置到起点上不是一件容易的事情。在实际应用中,常将包含起点的某个范围定义为引力域。图 3-2 中,已知多边形的起点为 $P(x,y)$,其引力域可以定义为 $(x\pm a, y\pm a)$ 的正方形区域 a 为正方形边长的一半。如果鼠标位于该区域内,就强制多边形的终点坐标为 $P(x,y)$。在 WM_MOUSEMOVE 消息处理函数中实现了引力域技术。

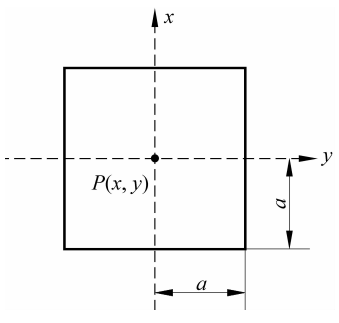

图 3-2 引力域

```
void CTestView::OnMouseMove(UINT nFlags, CPoint point)
```

```
{
    // TODO: Add your message handler code here and/or call default
    if(m_Arrow)
        ::SetCursor(AfxGetApp()->LoadStandardCursor(IDC_ARROW));
    else
        ::SetCursor(AfxGetApp()->LoadStandardCursor(IDC_CROSS));
    CString strx,stry;                          //状态栏显示鼠标位置
    CMainFrame * pFrame=(CMainFrame*)AfxGetApp()->m_pMainWnd;
                                                //要求包含 MainFrm.h 头文件
    CStatusBar * pStatus=&pFrame->m_wndStatusBar;
                                                //需要将 m_wndStatusBar 属性修改为公有
    if(pStatus)
    {
        strx.Format("x=%d",point.x);
        stry.Format("y=%d",point.y);
        CDC * pDC=GetDC();
        CSize sizex=pDC->GetTextExtent(strx);
        CSize sizey=pDC->GetTextExtent(stry);
        pStatus->SetPaneInfo(1,ID_INDICATOR_X,SBPS_NORMAL,sizex.cx);
                                                //改变状态栏风格
        pStatus->SetPaneText(1,strx);
        pStatus->SetPaneInfo(2,ID_INDICATOR_Y,SBPS_NORMAL,sizey.cx);
                                                //改变状态栏风格
        pStatus->SetPaneText(2,stry);
        ReleaseDC(pDC);
    }
    int index=m_ptrarray.GetSize()-1;
    if(m_LBDown)
    {
        if(!m_IsInsert)                         //如果是第一次移动,则插入新的顶点
        {
            CPointArray * pPointArray=new CPointArray(point);
            m_ptrarray.Add(pPointArray);
            m_IsInsert=TRUE;
        }
        else                                    //修改上次插入的顶点数据
        {
            ((CPointArray *)m_ptrarray.GetAt(index))->pt=point;
        }
    }
    if(m_LBDown)
    {
        if(MK_SHIFT==nFlags)                    //约束:测试按下 Shift 键
        {
            CPoint* pt1=&(((CPointArray *)m_ptrarray.GetAt(index))->pt);
```

```
            CPoint * pt2=&(((CPointArray *)m_ptrarray.GetAt(index-1))->pt);
            if(abs(pt1->x-pt2->x)>=abs(pt1->y-pt2->y))
            {
                pt1->y=pt2->y;              //x 方向的垂线
            }
            else
            {
                pt1->x=pt2->x;              //y 方向的垂线
            }
        }
    }
    if(index>3)
    {
        CPoint pt=((CPointArray*)m_ptrarray.GetAt(0))->pt;
        if((abs(point.x-pt.x)<=5) && (abs(point.y-pt.y)<=5))
                                            //引力域:边长为 10 的正方形
        {
            ((CPointArray *)m_ptrarray.GetAt(index))->pt=pt;    //修改数据
            m_Arrow=TRUE;
            m_LBDown=FALSE;
            m_MState=TRUE;
            m_Flag=FALSE;
        }
    }
    Invalidate(FALSE);
    CView::OnMouseMove(nFlags, point);
}
```

3.5.2 双缓冲技术

实验要求使用橡皮筋技术动态地将多边形的绘制过程表现出来,直到产生满意的结果为止。这需要借助双缓冲技术实现。双缓冲是在内存中创建一个与屏幕客户区大小一致的内存对象,先将图形绘制到内存中的这个对象上,再一次性将这个内存对象上的图形复制到屏幕客户区,这样能避免移动边的终点过程中,将边的移动过程全部绘制出来。实现时先在内存设备上下文 MemDC 中作图,然后使用 BitBlt()函数将做好的图形复制到屏幕客户区显示设备上下文 pDC 中,同时禁止背景刷新,消除了屏幕闪烁。

```
void CTestView::DoubleBuffer()              //双缓冲
{
    CRect rect;                             //定义客户区
    GetClientRect(&rect);                   //获得客户区的大小
    CDC * pDC=GetDC();
    CDC MemDC;                              //内存设备上下文
    CBitmap NewBitmap, * pOldBitmap;        //内存中承载图像的临时位图
    MemDC.CreateCompatibleDC(pDC);          //建立与屏幕 pDC 兼容的 MemDC
```

```
    NewBitmap.CreateCompatibleBitmap(pDC,rect.Width(),rect.Height());
                                                    //创建兼容位图
    pOldBitmap=MemDC.SelectObject(&NewBitmap); //将兼容位图选入 MemDC
    MemDC.FillSolidRect(rect,pDC->GetBkColor());
                                                    //按原来背景填充客户区,否则是黑色
    DrawObject(&MemDC);
    pDC->BitBlt(0,0,rect.Width(),rect.Height(),&MemDC,0,0,SRCCOPY);
                                                    //将内存位图复制到屏幕
    MemDC.SelectObject(pOldBitmap);                 //恢复位图
    NewBitmap.DeleteObject();                       //删除位图
    MemDC.DeleteDC();                               //删除 MemDC
    ReleaseDC(pDC);                                 //释放 DC
}
```

3.5.3 绘制多边形

多边形使用 CLine 类绘制直线,使用 new 运算符动态创建的直线对象指针,要使用 delete 运算符删除,才能防止内存泄漏。

```
void CTestView::DrawObject(CDC * pDC)               //绘制多边形
{
    int index=m_ptrarray.GetSize();
    CLine * line=new CLine;
    if(index)
    {
        line->MoveTo(pDC,((CPointArray*)m_ptrarray.GetAt(0))->pt);
        for(int i=1;i<index;i++)
        {
            line->LineTo(pDC,((CPointArray*)m_ptrarray.GetAt(i))->pt);
        }
        if(FALSE==m_Flag)                           //线段闭合,填充图形
        {
            FillPolygon(pDC);
        }
    }
    delete line;
}
```

3.5.4 设计 CPtrArray 类

本实验绘制的是任意顶点数的多边形,可以借助于一维动态数组实现。数组的大小由屏幕上所绘制的顶点个数确定。数组的定义使用了 MFC 提供的 CPtrArray 集合类实现,该集合类类似于一维数组的功能,但可以动态地增减。

(1) 首先创建一个 CPointArray 类,来保存 CPoint 类型的点。

```
class CPointArray :public CObject
{
```

```
public:
    CPointArray();
    CPointArray(CPoint);
    virtual ~CPointArray();
public:
    CPoint pt;
};
```

CObject 为 MFC 的根类,在创建 CPointArray 类时,没有出现在基类列表中,需要手工添加。

(2) CPointArray 类重载了构造函数:

```
CPointArray::CPointArray(CPoint pt)
{
    this->pt=pt;
}
```

(3) 在鼠标左键按下消息处理函数中保存当前点到 CPointArray 对象 m_ptrarray 中

```
void CTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    if(TRUE==m_Flag)                          //绘图状态
    {
        m_LBDown=TRUE;
        CPointArray * pPointArray=new CPointArray(point);
        m_ptrarray.Add(pPointArray);          //添加新顶点
    }
    m_IsInsert=FALSE;
    CView::OnLButtonDown(nFlags, point);
}
```

3.5.5 有效边表填充算法

多边形的填充使用有效边表算法实现,为此定义了 CAET 类、CBucket 类和 CFill 类。

1. 定义边结点类

```
class CAET
{
public:
    CAET();
    virtual ~CAET();
public:
    double    x;                              //当前扫描线与有效边的交点的 x 坐标
    int       yMax;                           //边的最大 y 值
    double    k;                              //斜率的倒数(x 的增量)
    CAET      * next;
};
```

2. 定义桶结点类

```
class CBucket
{
public:
    CBucket();
    virtual ~CBucket();
public:
    int     ScanLine;                          //扫描线
    CAET    * pET;                             //边表
    CBucket * next;
};
```

3. 定义填充多边形类

```
class CFill
{
public:
    CFill();
    virtual ~CFill();
    void SetPoint(CPoint p[],int);             //设定多边形顶点
    void CreateBucket();                       //创建桶
    void CreateEdge();                         //边表
    void AddEt(CAET * );                       //合并 ET 表
    void EtOrder();                            //ET 表排序
    void FillPolygon(CDC * );                  //填充
    void ClearMemory();                        //清理内存
    void DeleteAETChain(CAET* pAET);           //删除边表
private:
    int     PNum;                              //顶点个数
    CPoint  * P;                               //多边形顶点数组
    CAET    * pHeadE, * pCurrentE, * pEdge;    //有效边表结点指针
    CBucket * pHeadB, * pCurrentB;             //桶表结点指针
};
CFill::CFill()
{
    PNum=0;
    P=NULL;
    pEdge=NULL;
    pHeadB=NULL;
    pHeadE=NULL;
}
CFill::~CFill()
{
    if(P!=NULL)
    {
        delete[] P;
        P=NULL;
```

```
    }
    ClearMemory();
}
void CFill::SetPoint(CPoint p[],int m)          //动态创建多边形顶点数组
{
    P=new CPoint[m];
    for(int i=0;i<m;i++)
    {
        P[i]=p[i];
    }
    PNum=m;
}
void CFill::CreateBucket()                      //创建桶表
{
    int yMin,yMax;
    yMin=yMax=P[0].y;
    for(int i=0;i<PNum;i++)                     //查找多边形所覆盖的最小和最大扫描线
    {
        if(P[i].y<yMin)
        {
            yMin=P[i].y;                        //扫描线的最小值
        }
        if(P[i].y>yMax)
        {
            yMax=P[i].y;                        //扫描线的最大值
        }
    }
    for(int y=yMin;y<=yMax;y++)
    {
        if(yMin==y)                             //建立桶头结点
        {
            pHeadB=new CBucket;                 //pHeadB 为 CBucket 的头结点
            pCurrentB=pHeadB;                   //CurrentB 为 CBucket 当前结点
            pCurrentB->ScanLine=yMin;
            pCurrentB->pET=NULL;                //没有链接边表
            pCurrentB->next=NULL;
        }
        else                                    //建立桶的其他结点
        {
            pCurrentB->next=new CBucket;
            pCurrentB=pCurrentB->next;
            pCurrentB->ScanLine=y;
            pCurrentB->pET=NULL;
            pCurrentB->next=NULL;
        }
    }
}
```

```
void CFill::CreateEdge()                          //创建边表
{
    for(int i=0;i<PNum;i++)
    {
        pCurrentB=pHeadB;
        int j=(i+1)%PNum;                         //边的第二个顶点,P[i]和 P[j]构成边
        if(P[i].y<P[j].y)                         //边的起点比终点低
        {
            pEdge=new CAET;
            pEdge->x=P[i].x;                      //计算 ET 表的值
            pEdge->yMax=P[j].y;
            pEdge->k=(double)(P[j].x-P[i].x)/((double)(P[j].y-P[i].y));
                                                  //代表 1/k
            pEdge->next=NULL;
            while(pCurrentB->ScanLine!=P[i].y)//在桶内寻找该边的 yMin
            {
                pCurrentB=pCurrentB->next;    //移到 yMin 所在的桶结点
            }
        }
        if(P[j].y<P[i].y)                         //边的终点比起点低
        {
            pEdge=new CAET;
            pEdge->x=P[j].x;
            pEdge->yMax=P[i].y;
            pEdge->k=(double)(P[i].x-P[j].x)/((double)(P[i].y-P[j].y));
            pEdge->next=NULL;
            while(pCurrentB->ScanLine!=P[j].y)
            {
                pCurrentB=pCurrentB->next;
            }
        }
        if((P[j].y)!=P[i].y)
        {
            pCurrentE=pCurrentB->pET;
            if(pCurrentE==NULL)
            {
                pCurrentE=pEdge;
                pCurrentB->pET=pCurrentE;
            }
            else
            {
                while(NULL!=pCurrentE->next)
                {
                    pCurrentE=pCurrentE->next;
                }
                pCurrentE->next=pEdge;
            }
```

```
        }
    }
}
void CFill::AddEt(CAET * pNewEdge)              //合并 ET 表
{
    CAET * pCE=pHeadE;
    if(pCE==NULL)
    {
        pHeadE=pNewEdge;
        pCE=pHeadE;
    }
    else
    {
        while(pCE->next!=NULL)
        {
            pCE=pCE->next;
        }
        pCE->next=pNewEdge;
    }
}
void CFill::EtOrder()                           //边表的冒泡排序算法
{
    CAET * pT1=NULL, * pT2=NULL;
    int Count=1;
    pT1=pHeadE;
    if(NULL==pT1)
    {
        return;
    }
    if(NULL==pT1->next)                         //如果该 ET 表没有再连 ET 表
    {
        return;                                 //桶结点只有一条边,不需要排序
    }
    while(NULL!=pT1->next)                      //统计结点的个数
    {
        Count++;
        pT1=pT1->next;
    }
    for(int i=1;i<Count;i++)                    //冒泡排序
    {
        pT1=pHeadE;
        if(pT1->x>pT1->next->x)                 //按 x 由小到大排序
        {
            pT2=pT1->next;
            pT1->next=pT1->next->next;
            pT2->next=pT1;
            pHeadE=pT2;
```