

吉林大学

计算机科学与技术学院

《计算机图形学》 实验报告

班级： 212016

学号： 21200612

姓名： 郭冠男

实验项目 1	边标志算法的实现		
实验性质	<input type="checkbox"/> 演示性实验 <input checked="" type="checkbox"/> 验证性实验 <input type="checkbox"/> 操作性实验 <input checked="" type="checkbox"/> 综合性实验		
实验地点	线上	机器编号	N/A

一、实现的功能

概述

本实验旨在完成实验要求的基础上，尽可能方便使用者控制多边形的绘制并可根据自己的意愿修改一个已绘制的多边形的顶点位置，所有操作全部为鼠标操作。程序源代码详见文件夹 **CG20612**。由于丰富的功能需要使用视频才能直观方便地展示，下文中提及的所有**视频**均位于 VIDEO 文件夹中。这些视频没有经过任何后期处理，均为程序运行时的效果。本项目将程序源代码与实验报告一同提交，因此在本实验报告中并不会大段引用程序的源代码以减小篇幅。如果评审老师有合作编写教学用演示软件的意向，欢迎联系 guogn2120@mails.jlu.edu.cn。


理念


为了清楚地介绍本实验的实现逻辑，下面我将简述这些实验中经常被用到两个理念。如果不阐明这些理念，我将很难介绍我的程序中一些复杂的结构。


控制点理念：本实验报告中的实验大都基于一套“控制点”的理念。用户在绘制图形时，需要先指定一系列的“控制点”，然后再根据这些控制点绘制出整个图形。例如，我们可以用圆心和圆周上的任意一点作为控制点绘制一个圆形，利用椭圆曲线外包围盒的左上角和右下角两个点作为控制点绘制一个椭圆。当一个几何图形被绘制到屏幕上后，用户可以使用鼠标拖拽这些控制点，以改变这些控制点的位置。当控制点的位置改变时，画面会实时更新，以保证绘制到屏幕上的几何图案实时满足控制点的约束。除此之外，多个几何图形可以共用一些控制点。**视频 MP4-0001.mp4** 形象地展示了控制点以及控制点的共用带来的效果。这种随着鼠标移动而实时变换的几何效果并不是使用橡皮线实现的，而是使用双缓冲实现的，这种方法也被广泛地应用到了本实验报告中的其他实验当中，双缓冲算法在复杂图形的实时绘制中具有一定的普适性。

自动机理念：当用户移动鼠标时，有时目的是拖动控制点，有时目的是改变视角，有时目的仅仅是移动鼠标。在 CView 对象中，我们将维护一个有限状态自动机以描述某一时刻、某个具体的消息应该以何种方式进行解释，会触发哪些函数，又将会使得自动机状态转移至何处。

功能

点击工具栏上的“绘制多边形”按钮  进入多边形绘制状态。其后，使用鼠标点击，在屏幕上绘制若干个控制点。当输入了最后一个控制点后，再次点击该控制点，或者单击整个多边形的第一个控制点都可以结束多边形的输入。这种设计既可以兼容课上老师介绍的双击结束绘制的方法，又提供了一种通过“闭合”完成多边形绘制的输入方式。当一个多边形的输入完成后，会被立刻填充，除非当前程序自动机处于**不填充状态**。视频 **MP4-0002.mp4** 给出了一个多边形绘制过程的示例。我们将在后文中具体定义什么是“不填充状态”。

点击工具栏上的“切换填充模式”按钮  可以让程序在**填充模式**和**不填充模式**之间进行切换。程序启动时，默认处于填充模式。在填充模式下，所有绘制在屏幕上的多边形都会被填充，在不填充模式下程序只绘制多边形的边框以及控制点。视频 **MP4-0003.mp4** 给出了一个切换填充模式按钮的使用示例。

点击工具栏上的“删除控制点”按钮  可以删除屏幕上的某一个控制点，如果一个多边形依赖于该控制点，那么这个多边形也会被一起删除。视频 **MP4-0004.mp4** 给出了一个删除控制点的示例，当我们删除一个被共用的控制点时，所有依赖于这个控制点的实体都会被一同删除。

当程序使用者没有点击任何工具栏上的按钮时，自动机处于自由状态（STATE_FREE）在该状态下，鼠标拖动控制点可以实现控制点位置的移动。由于控制点的移动过程中多边形的形状会实时改变，而多边形填充算法的效率不高，为了避免影响用户的使用体验，我们规定“不填充状态”为“鼠标按下”或“不填充模式”。这样的设计可以给用户顺畅的操作体验。视频 **MP4-0005.mp4** 中给出了一个鼠标拖动控制点的示例。

二、 采用的图形学算法及实现

（算法的实现函数是什么（函数名，参数，返回值，函数功能等）以及采用了哪些数据结构（数组，链表等））

数据结构

本项目中主要使用 C++ 标准模板库（STL）中提供的 `Map` 和 `Vector` 作为存储各类数据的容器。为了有效的实现控制点的共用，我们需要定义一个抽象层次来掩盖具体的坐标与各图形的关系。简而言之，我们需要给每一个控制点一个编号，对于每个多边形，这个多边形并不记录它的每个顶点的坐标，而只记录它的每个控制点的编号。因此我们使用一个 `Vector<Int>` 来记录一个多边形由哪些控制点控制。

任何将被绘制到屏幕上的几何图形，我们都称之为“实体”，我们为每一个实体分配一个独一无二的实体编号。之后，可以使用 `Map<Int,Type>` 为各类型实体存储一个从实体编号到数据信息的映射（其中 `Type` 指一个自定义的类型），使得我们可以快速找到某个编号所对应实体的信息。对于数据结构的定义详见文件 **CG20612View.h** 的前 35 行代码。`Type` 位置的可能值为 `CMyNode`, `CEllipse`, `CCircle`, `CPolygon` 等四个自定义的结构体。

算法的实现函数

算法的实现包含了多个函数，`MyFunc_FillPolygon` 是多边形算法的入口函数，每当程序需要对多边形进行填充时，这个函数都会被调用。函数 `MyFunc_GetPolygonRect` 用来获得一个多边形的外包络盒，`MyMathFunc_XorBuffer` 用于将一条边对掩码缓冲区 `buffer` 的“贡献”异或到数组中，`MyMathFunc_GetRgbByHsv` 用于实现从 HSV 颜色表示到 RGB 颜色表示的转换。利用这一转换，我们可以实现对多边形的彩虹式填充。

在算法中我们使用了一种差分技术优化了对缓冲区的异或过程，我们将在输出时对得到的差分 `buffer` 通过一次前缀和运算还原得到正确的掩码 `buffer`。通过实验，我们能够证明这种差分优化在平均意义上能将填充的时间效率提升一倍。经历了这样的前缀和过程后，在逻辑上，`buffer` 中等于 1 的位置对应着应该被填充的位置，也就是多边形内的位置，`buffer` 中等于 0 的位置对应着多边形外的位置。而实际上，我们并不真的将前缀和存回 `buffer` 数组，而是使用变量 `posNow` 暂存前缀和的临时值（详见 `MyFunc_Fillpolygon` 中注释“Output”对应的部分）。当屏幕上有多个多边形要被填充时，

MyFunc_FillPolygon 将被调用多次。上述功能的具体代码实现详见文件 **CG20612View.cpp**。

三、采用的交互方式及实现

（采用了哪些交互方式来完成绘制，这些交互方式应用到了哪些系统消息，是如何实现的）

自动机状态

正如第一部分“实现的功能”所述，本实验程序完全使用鼠标进行操作。当用户使用鼠标点击工具栏上的按钮时，会引起自动机状态的改变。变量 `m_State` 记录当前程序的自动机状态。文件 **CG20612View.h** 中定义了所有的自动机状态，如下图所示：

```
42  #define STATE_FREE (0)          /* 自由光标 */
43  #define STATE_SETNODE (1)       /* 设置结点 */
44  #define STATE_MOVENODE (2)      /* 移动节点 */
45  #define STATE_DELETENODE (3)    /* 移动节点 */
46  #define STATE_SETELLIPSE (4)    /* 绘制椭圆 */
47  #define STATE_SETCIRCLE (5)     /* 绘制圆形 */
48  #define STATE_SETPOLYGON (6)    /* 绘制多边形 */
```

在所有的消息处理函数中，我们均使用一个 Switch-Case 语句来根据当前 `m_State` 的值确定自动机状态将如何转移，并调用相应功能函数。例如，`m_State = STATE_MOVENODE` 表示正有一个控制点被用户“拿起”，当用户鼠标左键抬起的消息到来时，`m_State` 将被赋值为 `STATE_FREE`。每个状态的含义可以“顾名思义”，在此不再赘述。

`STATE_MOVENODE` 状态有利于圆形和椭圆的绘制。视频 **MP4-0001.mp4** 中圆形半径的设置过程可以展现 `STATE_MOVENODE` 状态的具体作用。这一状态不仅在我们移动某个控制点时才起作用。在“两点图形”绘制时，自动机也会进入这一状态，其中“两点图形”指恰依赖两个控制点的图形，包括圆形、椭圆、矩形等。

Windows 消息

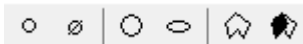
要实现上述自动机状态转移功能，需要对下列事件进行监听：

```

145    	afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
146    	afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
147    	afx_msg void OnMouseMove(UINT nFlags, CPoint point);
148    	afx_msg void OnSetnode();
149    	afx_msg void OnDeletenode();
150    	afx_msg void OnSetellipse();
151    	afx_msg void OnSetcircle();
152    	afx_msg void OnSetpolygon();
153    	afx_msg void OnTogglefill();

```

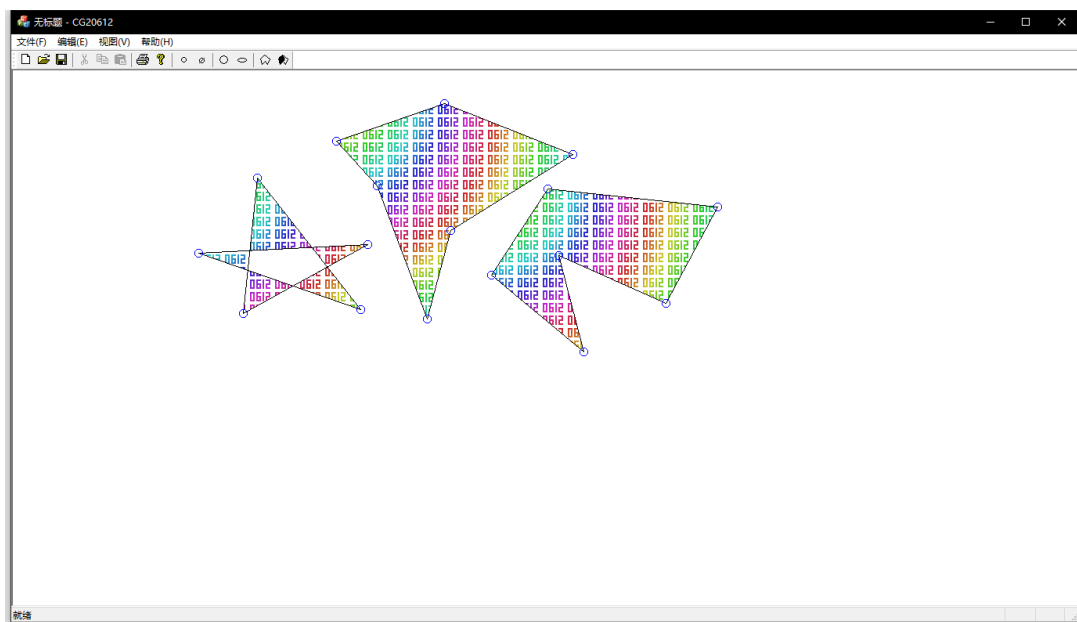
其中前三个函数是 Windows 系统消息对应的消息处理函数，后六个函数是工具栏中按钮对应的用户自定义消息处理函数，它们与工具栏中的按钮一一对应。工具栏截图如下：



四、实验结果

（程序的运行结果）

程序运行截图



程序运行视频

详见 VIDEO 文件夹下的 **MP4-0002.mp4** 至 **MP4-0005.mp4** 四个视频文件。其中视频 **MP4-0002.mp4** 给出了绘制一个多边形的操作过程，视频 **MP4-0003.mp4** 介绍了切换填充模式按钮的使用效果。视频 **MP4-0004.mp4** 演示了删除一个控制点的效果，视频 **MP4-0005.mp4** 演示了鼠标拖动控制点的效果。

五、遇到的问题及解决办法

（在实现过程中遇到了什么样的问题，及采用了何种解决办法）

极值点填充泄露问题

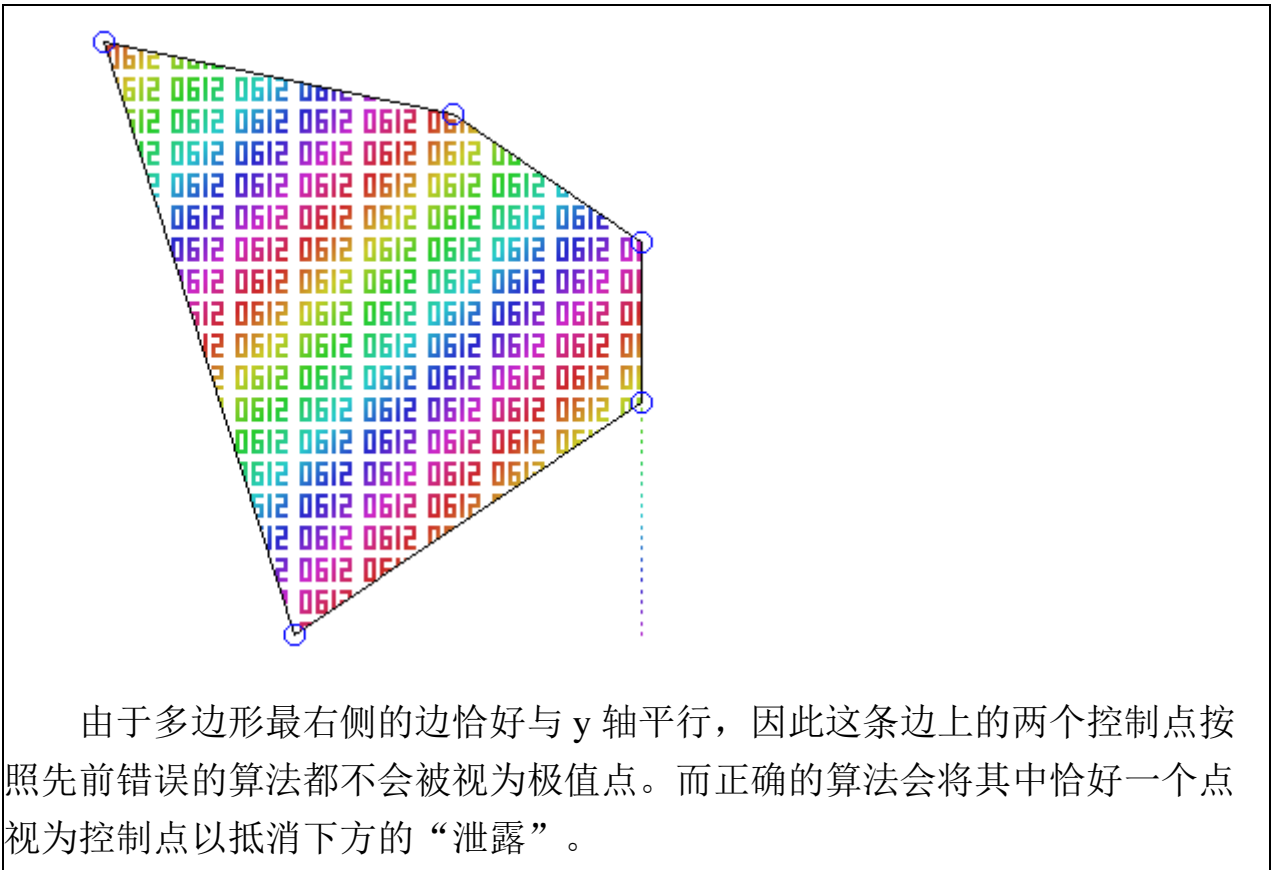
在实现的过程中果然遇到了老师上课时介绍的极值点泄露问题。我尝试自己寻找一种解决极值点问题的方法，而没有采用老师上课给出的方法。起初我对极值点进行了这样的特判：




```
/* 极值点 */  
if (lastPos.x < nowPos.x && nextPos.x < nowPos.x ||  
    lastPos.x > nowPos.x && nextPos.x > nowPos.x) {  
    buffer[nowPos.x - topleft.x][nowPos.y - topleft.y] ^= 1;  
}
```

这样的做法实际上是错误的，而这一事实直到我完成了实验四时才发现，因为实验四的裁剪算法总能裁剪出平行于坐标轴的整齐边缘，而这一算法仅在多边形存在恰好平行于 y 轴的边时才会出错。手绘的多边形很少出现恰好平行于 y 轴的边界，因此这个错误起初并没有被及时发现。最后这个问题得到了纠正，正确的做法如下：

```
/* 极值点 */  
if (lastPos.x <= nowPos.x && nextPos.x < nowPos.x ||  
    lastPos.x >= nowPos.x && nextPos.x > nowPos.x) {  
    buffer[nowPos.x - topleft.x][nowPos.y - topleft.y] ^= 1;  
}
```

对于极值点的可靠判断，应该是一侧带等号另一侧不带等号的。错误的算法会导致下图所示的错误：



实验项目 2	用矩形窗口对多边形进行裁剪		
实验性质	<input type="checkbox"/> 演示性实验 <input checked="" type="checkbox"/> 验证性实验 <input type="checkbox"/> 操作性实验 <input checked="" type="checkbox"/> 综合性实验		
实验地点	线上	机器编号	N/A
<p>一、实现的功能</p> <p>在实验项目 1 的基础上，我们增加了使用矩形窗口对多边形的裁剪功能。假设用于已经绘制了若干个多边形，点击“裁剪”按钮后可以使用矩形裁剪功能再屏幕上用鼠标绘制一个矩形线框。矩形线框绘制结束后程序会对屏幕上的所有多边形进行裁剪，只保留每个多边形在矩形线框内的部分。裁剪后，我们会以类似实验项目 1 的方式对多边形进行填充。可以点击工具栏中的“显示对比图”按钮以兼容第四次课实验要求的方式给出裁剪前后多边形的线框对比。视频 MP4-0006.mp4给出了一次完整操作流程的示例。</p> <p>在一次裁剪结束后，我们将得到若干个裁剪后的多边形。在此基础之上我们可以继续绘制多边形，并与得到的裁剪后的多边形一同参与到下一次裁剪中，以保证算法的可重入性。显示对比图功能始终显示最后一次裁剪的矩形线框以及裁剪前后多边形形状的对比。点击工具栏上的“删除全部实体”按钮可以清空屏幕上的所有控制点和多边形。除此之外的其他按钮功能与实验项目 1 一致，不再赘述。</p> <p>二、采用的图形学算法及实现</p> <p>（算法的实现函数是什么（函数名，参数，返回值，函数功能等）以及采用了哪些数据结构（数组，链表等））</p> <p>出于实验报告的简洁性，在此我们只具体介绍与裁剪算法有关的数据结构以及函数，本实验中的其他算法介绍从简，具体代码详见文件CG40612View.cpp。</p> <p>当程序要进行裁剪时，程序调用 <code>MyFunc_CutPolygons</code> 函数以完成对所有多边形的裁剪。该函数会对所有多边形在裁剪前的位置信息进行备份，并调用 <code>MyFunc_CutPolygon</code> 依次对每个多边形进行裁剪。在函数 <code>MyFunc_CutPolygon</code> 中，我们分别调用 <code>MyStaticFunc_CutXAbove</code>、<code>MyStaticFunc_CutXBelow</code>、<code>MyStaticFunc_CutYAbove</code>、</p>			

MyStaticFunc_CutYBelow 四个函数以实现对多边形的裁剪。在代码中实际上只实现了 MyStaticFunc_CutXAbove 一个函数，另外三个 Cut 函数的实现均间接调用了 MyStaticFunc_CutXAbove 函数。

我认为，在软件开发过程中，让具有相同功能的代码至多只被实现一次是一种有利于软件复用与拓展的设计方式。当我们发现程序中的某个算法存在问题需要修改时，我们只需要修改它对应的唯一实现。如果程序中存在多个功能相近的程序段，那我们不得不去依次修改每一个程序段以使得程序最终符合我们的期望，而这在软件开发过程中很可能是致命的。

三、采用的交互方式及实现

（采用了哪些交互方式来完成绘制，这些交互方式应用到了哪些系统消息，是如何实现的）

自动机状态

同实验项目 1 一样，本程序在宏观的设计上仍然采用自动机的设计思想。CView 的成员变量 m_State 的值表示当前程序所处的自动机状态。当程序中的消息处理函数被调用时，自动机会根据前驱状态和当前事件计算出后继状态，并调用这一事件需要触发的一些功能。所有状态列举如下，限于篇幅我们很难一一介绍它们的具体功能：

```
47 #define STATE_FREE (0)          /* 自由光标 */
48 #define STATE_SETNODE (1)       /* 设置结点 */
49 #define STATE_MOVENODE (2)      /* 移动节点 */
50 #define STATE_DELETENODE (3)    /* 移动节点 */
51 #define STATE_SETELLIPSE (4)    /* 绘制椭圆 */
52 #define STATE_SETCIRCLE (5)    /* 绘制圆形 */
53 #define STATE_SETPOLYGON (6)   /* 绘制多边形 */
54 #define STATE_CUTPOLYGON (7)   /* 剪切多边形 */
```

其中 STATE_CUTPOLYGON 状态表示用户正在输入多边形剪切的矩形框。在这一过程中，用户需要按住鼠标拖动一段距离。鼠标左键按下的位置，鼠标左键抬起的位置，会被视为最终得到的裁剪多边形的两个角。

监听的消息

为了实现自动机状态的转移，我们需要监听一系列 Windows 系统消息以及用户自定义消息。消息处理函数列举如下：

```

170 public:
171     afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
172     afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
173     afx_msg void OnMouseMove(UINT nFlags, CPoint point);
174     afx_msg void OnSetnode();
175     afx_msg void OnDeletenode();
176     afx_msg void OnSetellipse();
177     afx_msg void OnSetcircle();
178     afx_msg void OnSetpolygon();
179     afx_msg void OnTogglefill();
180     afx_msg void OnCutpolygon();
181     afx_msg void OnShowcompare();
182     afx_msg void OnDeleteall();

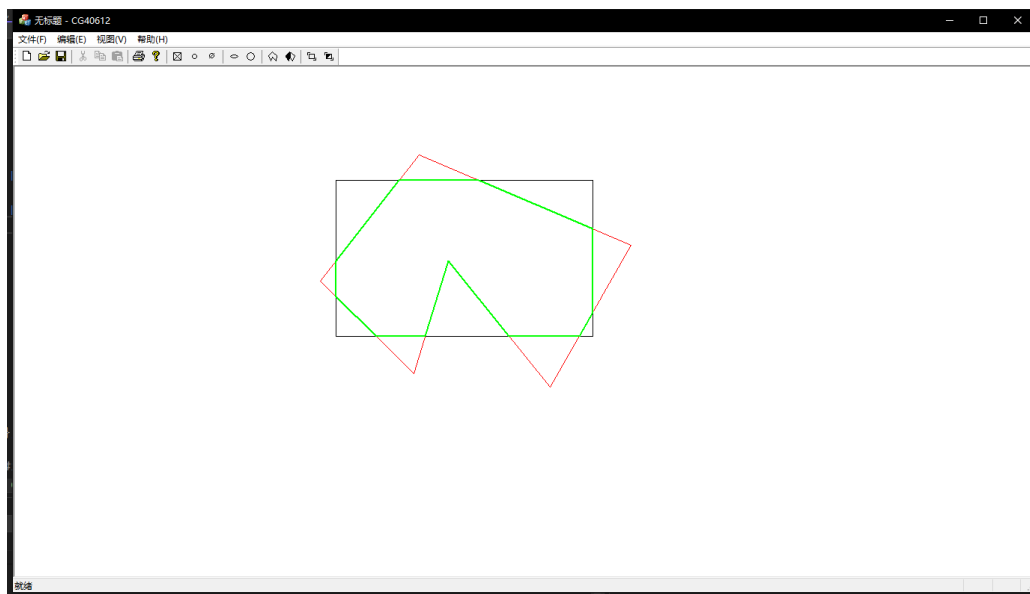
```

其中前三个函数为是 Windows 系统消息函数，后九个函数是用户自定义的消息函数，它们与工具栏上的按钮一一对应。

四、实验结果

（程序的运行结果）

视频 **MP4-0006.mp4** 给出了一次完整操作流程的示例。我们在此给出一个程序运行结果的截图。下图中红色边框表示原始多边形，绿色边框表示裁剪后的多边形，黑色矩形框表示裁剪矩形。本程序同样适用于多个多边形同时被裁剪的情况。



五、遇到的问题及解决办法

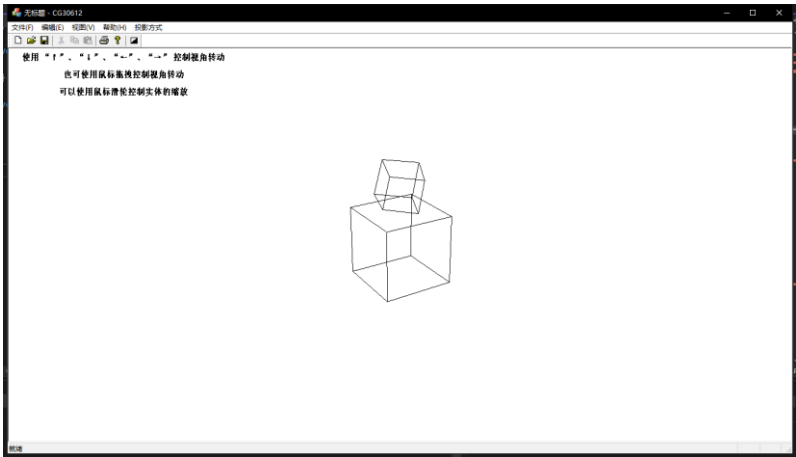
（在实现过程中遇到了什么样的问题，及采用了何种解决办法）

并没有遇到算法层面的问题，只在编写程序的过程中遇到了一些小的逻辑错误。例如在裁剪时忘记了结点所设立的抽象层，误把结点的编号直接当作坐标，导致裁剪结果十分诡异。最后通过单步调试技术定位并修改了程序中的问题代码。

实验项目 3	立方体的比例、平移、旋转变换及投影显示		
实验性质	<input type="checkbox"/> 演示性实验 <input checked="" type="checkbox"/> 验证性实验 <input type="checkbox"/> 操作性实验 <input checked="" type="checkbox"/> 综合性实验		
实验地点	线上	机器编号	N/A


一、实现的功能

本实验项目旨在实现一套可以使用鼠标控制视角变化以及比例变换的立方体旋转演示程序。当用户不做任何操作时，屏幕上将绘制两个正方体：一个大正方体作为底座，一个侧立的小正方体在大正方体的上表面自动绕 z 轴进行旋转。具体效果如**视频 MP4-0007.mp4** 所示，下图给出了自由状态下，程序运行效果的截图：



我们可以使用鼠标拖动切换视角，可以使用鼠标滑轮滚动对显示区域内的实体进行整体缩放。**视频 MP4-0008.mp4** 给出了一个使用鼠标拖动切换视角方向的示例，**视频 MP4-0009.mp4** 给出了一个使用滑轮对实体进行整体缩放的示例。这种操作方式比输入坐标点的方式更加直观，而且在改动视角以及缩放的过程中，程序能够保持小正方体的转动不受影响。

用户也可以使用键盘上的“上”、“下”、“左”、“右”四个方向键控制视角的转动。每当程序接收到 KeyDown 消息时，我们的程序并不直接处理 KeyDown 消息中得到的字符而是使用 GetAsyncKeyState 函数确定当前有哪些按键被按下了。使用这种方法可以保证当键盘上同时按下两个键时，例如同时按下了“下键”和“左键”，视角的能平滑地沿着两个方向的“和方向”进行转动，而不是只按最后按下的按键所指出的方向转动。这种设计更符合用户的操作逻辑，也能使用户对操作界面的理解更加直观。

当用户点击“颜色填充模式切换”按钮  时，程序会暂停小正方体的旋转并且使用 z 缓冲算法对正方体的六个面分别进行着色。六个面的颜色分别为：“红”、“绿”、“蓝”、“紫”、“黄”、“青”。由于 z 缓冲算法在时间上开销很大，如果在颜色填充模式下保持对小正方体的旋转，则会导致显示界面出现明显的卡顿，影响用户体验。因此我们规定，在颜色填充模式下，我们禁用了视角的转动以及小正方体的转动动画。下图给出了一个在平行投影模式下的颜色填充效果图：



我们可以在菜单栏中“投影方式”一栏下选择“平行投影”或者“透视投影”两种不同投影方式。这种切换即使是在颜色填充模式下也是有效的。

二、采用的图形学算法及实现

（算法的实现函数是什么（函数名，参数，返回值，函数功能等）以及采用了哪些数据结构（数组，链表等））

限于篇幅本文在此只介绍代码中与正方体绘制直接相关的函数，略去了数学上的细节以及消息处理机制中的细节。本文的代码没有使用常用的矩阵乘法与齐次坐标的处理方式，而是使用了计算几何中的点乘与叉乘计算各种投影的结果，这是一种与矩阵乘法等价的计算方式。我认为这种做法能够在某种意义上提升代码的可读性，因为每一个数学函数被调用时都带有着一定的主观上的目的，而当我们把一个矩阵表达式写到变量中时我们需要假定这个矩阵表达式在客观上是正确的——在将来自己重新阅读自己的代码时我们讲很难理解这些变换矩阵到底**为什么**是正确的。计算几何的方法从数学的角

度讲更加直观，在某种意义上，能够在算法层面清晰地展现出几何变换的完整流程。

函数 `MyFunc_ShowAllCubes` 负责显示所有的正方体，出于工作量，本程序并没有向用户提供增加正方体的接口，但是从逻辑上这一点是易于实现的——因为我们将正方体的所有信息存储到了名为 `m_CubeMap` 的关联式容器中，当我们想要增加一个新的正方体时，我们只需要在程序运行过程中向这个容器内再追加一个正方体对象即可。`MyFunc_ShowCube` 函数负责绘制正方体的线框，这个函数会在不填充颜色模式下被调用。

`MyFunc_GetSurfaceForCube` 函数负责根据一个正方体对象获取它的八个顶点的三维坐标，对于正方体的各种坐标变换，本质上都可以理解为对于它的八个顶点的坐标变换。这一函数无论是在不填充颜色模式还是在颜色填充模式下都会被调用。

`MyStaticFunc_GetRectFromCSurface2dList` 函数用来确定一个能够恰好包含整个被绘制正方体的包络盒子，程序会根据这个包络盒的尺寸申请对应大小的 `z` 缓冲空间。`CZBuffer` 是一个自定义的类，他有 `AddSurface` 和 `OutputToDC` 两个成员函数，分别用于向 `z` 缓冲中增加一个新的平面，以及将 `z` 缓冲中的内容绘制到屏幕上。

三、采用的交互方式及实现

（采用了哪些交互方式来完成绘制，这些交互方式应用到了哪些系统消息，是如何实现的）

为了实现对小正方体旋转的动态绘制，本程序使用了 Windows 提供的 `Timer` 消息。每接收到一次 `Timer` 消息，程序就会根据小正方体上次绘制的时间与当前时间的时间差计算出小正方体当前应该处于的位置并对小正方体的新位置进行显示。鼠标的抬起与落下消息以及键盘消息主要用于实现视角的转动，视角的转动会以一定的比例，在一定的范围内换算成对 `m_Center` 点的坐标变化，其中 `m_Center` 点用来表示透视投影的投影中心，在平行投影模式下我们使用 `m_Center` 点表示平行投影的投影方向。

四、实验结果

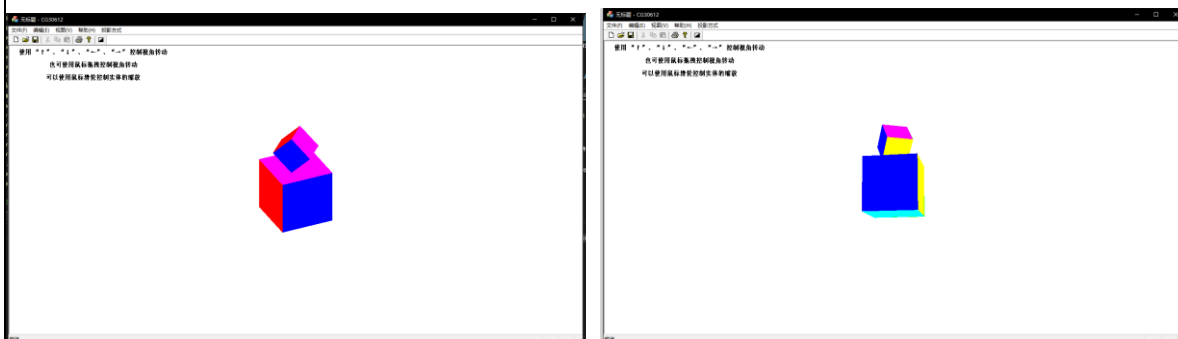
（程序的运行结果）

效果录像

程序的具体运行效果可以参考前文中提及的视频 **MP4-0007.mp4** 至 **MP4-0009.mp4**。视频 **MP4-0007.mp4** 给出了一个用户不进行操作时小正方体匀速转动的效果示例。视频 **MP4-0008.mp4** 给出了一个使用鼠标拖动切换视角方向的示例，视频 **MP4-0009.mp4** 给出了一个使用滑轮对实体进行整体缩放的示例。

效果图

下面的两幅图分别对应了在平行投影下以及透视投影下使用 z 缓冲算法得到的一次消隐填充效果：



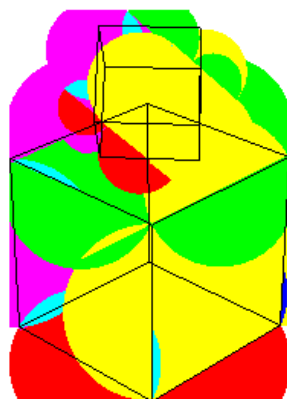
第一幅图采用平行投影算法，视角为俯视，第二幅图采用透视投影算法，视角为仰视。

五、遇到的问题及解决办法

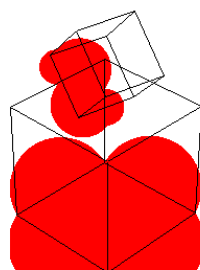
（在实现过程中遇到了什么样的问题，及采用了何种解决办法）

对于角度计算错误的定位

由于本程序使用旋转法确定一个点是否在一个多边形内部，在最初的实现版本中由于角度计算函数编写上的错误导致了涂色的混乱——因为程序在错误的角度计算下不能正确地使用旋转法判断出一个点是否在一个多边形内部。具体而言出现了如下图所示的混乱的染色效果：



上图中的框线本身不属于算法的一部分，是为了便于定位程序具体发生了何种错误而绘制出的参考线。巧妙的是，仅通过观察这个混乱的染色方案就能够推理出程序编写出错的位置。下图给出了只染色每个正方体的零号面时的错误染色效果图：



可以看出，染色区域比正常应该染色的区域多出了四个诡异的半圆弧。而这些圆弧意味着一个原本应该正确得到计算的 90° 夹角被错误地计算成了个相对随机的值（因为圆周上的点到某条边的两个顶点的夹角恰为 90° ），使得最终旋转矢量旋转一周得到的角度加和非零。而这 90° 对应着坐标系下的一个象限，最后在 `math::vrad` 函数中成功找到了出错的象限。对于这个错误的修复是在众多错误修复中比较精彩的一个。

深度计算上的错误

在 z 缓冲算法中，对于 z 值的计算存在一定的困难。先前版本的程序中使用的算法只能保证对平行投影图形进行正确的着色，因为我们为了提升效率采用了一种三角剖分近似的算法，这种算法只在距离函数均匀变化的前提下才能正确计算二维面中某点逆变换回三维后对应点的深度。但对于透视投影而言，这种做法只是一种近似做法，因此会导致透视投影中，正方体的边缘处会出现深度计算的误差，导致一小部分本应被消隐的平面露出。最终我参

考了 Sutherland 的论文 “*A Characterization of Ten Hidden-Surface Algorithms*” 中给出的 “三维透视变换” 的思路解决了这个问题。三维透视变换的本质就是对原先的三维环境中的每个点进行一个变换，使得得到的新三维环境中实施平行投影等价于在原三维环境中实施透视投影。完整代码详见

CG30612View.cpp 中 MyFunc_GetSurfaceForCube，最重要的代码在下面给出：

```
/* d1 是观察点到投影面的距离 */
double d1, d2;
d1 = math::GetDisFromPointToPlane(m_Center, plane);

/* 将多边形放到 Surface 中 */
CSurface2d tmpSurf = {};
tmpSurf.sid = nid++;
for (int i = 0; i <= 3; i += 1) {
    tmpSurf.x[i] = MyFunc_ProjectionWorldToDevice(corners[idList[i]]);

    /* 这里的距离应该特殊考虑透视变换距离 */
    tmpSurf.z[i] = d2 =
        math::GetDisFromPointToPlane(corners[idList[i]], plane);
    if (m_ProjectMethod == PROJECTMETHOD_PERSPECTIVE) {
        tmpSurf.z[i] *= d1 / (d1 + d2);
    }
}
```

对于透视投影的情况，我们使用上述代码中的 “ $d1/(d1+d2)$ ” 对深度进行修正，即可使用类似平行投影的方式正确地进行 z 缓冲绘制。