实验目的：

1. 通过实现图形学经典的二维裁剪算法，深入理解场景中对几何对象进行裁剪的原理。
2. 锻炼实践算法的能力。
3. 进一步熟悉OpenGL编程。

实验内容：

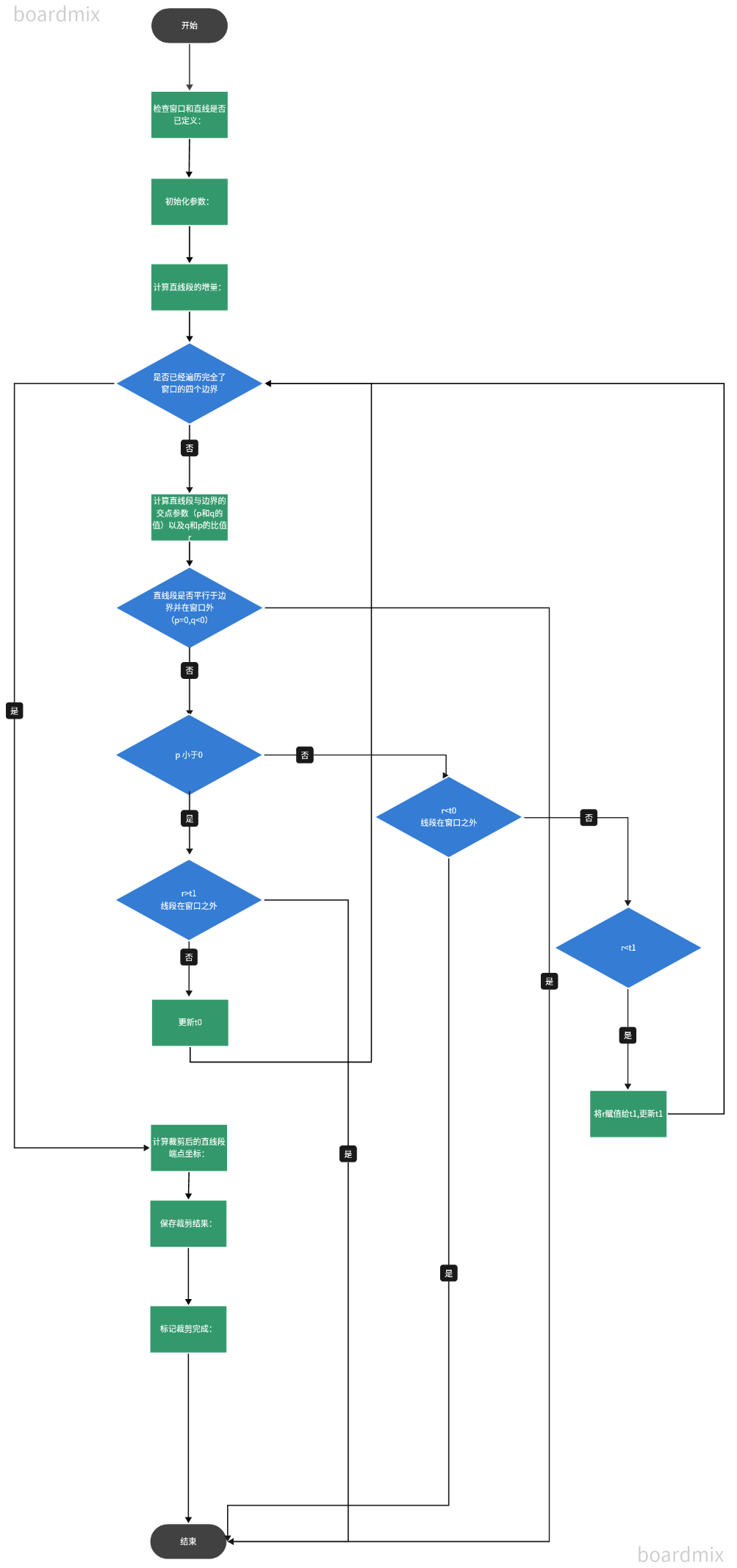
1. 用梁友栋－barsky算法或者中点分割法等其它算法（除cohen-sutherland直线裁剪算法外）实现直线段相对于给定窗口的裁剪。
2. 采用C/C++ 、OpenGL编写程序（参考所提供的程序代码clip.cpp及第三次实验提供的建立Project的过程说明）。
3. 选作：

（1） 利用Glut处理鼠标、键盘输入的功能，实现用鼠标交互输入的方式来定义窗口、被裁剪线段的功能。

（2） 改进提供的cohen-sutherland算法演示界面（cohen-sutherland.rar），写入你的裁剪算法代码。

实现方法：

* 写出算法流程图：



梁友栋－barsky算法

* 说明算法中所采用的数据结构；

利用的都是基本的数据结构和类型。

1. 基本数据类型：

使用了基本数据类型 int、float 来表示坐标和计算过程中的一些中间变量。

1. 数组：

使用了 m\_window、m\_ClippingLine 和 m\_ClippedLine 这些数组，分别用于表示窗口的两个端点、直线段的两个端点和裁剪后的直线段的两个端点。

1. 标志变量：

使用了 myWindowDefined、myLineDefined 和 myClipDone 这些标志变量，用于表示窗口是否定义、直线是否定义以及裁剪是否完成的状态。

总体而言，代码主要使用了基本的数据结构和控制结构，没有涉及复杂的数据结构。这符合裁剪算法的特性，因为梁友栋－Barsky算法主要涉及数学计算和条件判断，对于数据结构的需求相对较少。

* 说明算法各函数的功能；

该函数功能实现主要是**定义了MyChip类**并且利用该类中的成员函数实现二维切割。

**变量：**

1. CPoint m\_ClippedLine[2]:用于存储裁剪后的线段的两个端点。
2. CPoint m\_window[2]:存储窗口的两个角点（左下角和右上角）。
3. CPoint m\_ClippingLine[2]:存储待裁剪的线段的两个端点。
4. bool myWindowDefined:表示窗口是否已经被定义。如果为true，则窗口已经被定义；否则，窗口未定义。
5. bool myLineDefined:表示线段是否已经被定义。如果为true，则线段已经被定义；否则，线段未定义。
6. bool myClipDone:表示裁剪是否已经完成。如果为true，则裁剪已经执行；否则，裁剪尚未执行。

**函数：**

1. 构造函数 MyClip():

将私有成员变量 myWindowDefined、myLineDefined 和 myClipDone 初始化为 false。这表示最初窗口、线和剪切都未定义。

1. void setWindow(CPoint point1, CPoint point2):

使用两个角点 (point1 和 point2) 设置剪切区域的窗口。

1. void setLine(CPoint point1, CPoint point2):

使用两个端点 (point1 和 point2) 设置要进行剪切的线。

1. void clearWindow():

清除定义的窗口。

1. void clearLine():

清除定义的线。

1. void clearClip():

将剪切状态 (myClipDone) 重置为 false，表示需要重新执行剪切。

1. void doMyClipping():

执行剪切算法（根据注释假定为Cohen-Sutherland算法），将线相对于定义的窗口进行剪切。

1. bool isClipDone():

返回剪切状态 (myClipDone)。如果为 true，表示已经执行了剪切；否则，表示尚未执行。

1. unsigned char myEncode(CPoint thePoint):

实现了相对于当前窗口的给定点的Cohen-Sutherland编码。这种编码用于确定点相对于窗口的位置。

1. void swapCode(unsigned char &code1, unsigned char &code2):

交换两个编码的值。

1. void swapPts(CPoint &point1, CPoint &point2):

交换两个点的值。

**其中domychip函数为核心函数，实现过程基于梁友栋－Barsky算法，用于在给定窗口内裁剪直线段。**

**其详细步骤如下：**

1. 初始化参数：
   * t0 和 t1 初始化为线段的两个端点，即完整的未裁剪线段。
   * 计算 dx 和 dy，它们表示线段在x和y方向上的变化量。
2. 循环遍历窗口的四个边界：
   * 对于每个边界，计算 p 和 q 的值，这些值用于计算参数 r。
     + 如果当前边界是左边界，p 是线段的x方向变化，q 是线段起点相对于窗口左边界的偏移。
     + 如果当前边界是右边界，p 是线段的x方向变化，q 是线段起点相对于窗口右边界的偏移。
     + 如果当前边界是上边界，p 是线段的y方向变化，q 是线段起点相对于窗口上边界的偏移。
     + 如果当前边界是下边界，p 是线段的y方向变化，q 是线段起点相对于窗口下边界的偏移。
   * 计算 r，它表示线段与当前边界的交点在整个线段上的相对位置。
3. 处理特殊情况：
   * 如果 p 等于0且 q 小于0，说明线段与当前边界平行且在边界之外，整个线段应该被裁剪掉，函数直接返回。
4. 更新参数 t0 和 t1：
   * 如果 p 小于0，表示线段与边界平行，且在窗口内。
     + 如果 r 大于当前的 t1，说明线段在窗口之外，直接返回。
     + 如果 r 大于当前的 t0，更新 t0 的值为 r。
   * 如果 p 大于0，表示线段与边界平行，且在窗口内。
     + 如果 r 小于当前的 t0，说明线段在窗口之外，直接返回。
     + 如果 r 小于当前的 t1，更新 t1 的值为 r。
5. 计算裁剪后的线段端点：
   * 使用计算得到的参数 t0 和 t1，分别计算线段起点和终点在窗口边界上的交点。
6. 更新裁剪后的线段信息：
   * 将计算得到的裁剪后的线段端点更新到 m\_ClippedLine 中。
   * 设置标志 myClipDone 表示裁剪完成。

* 提供程序源代码并进行必要的注释；

void MyClip::doMyClipping()

{

//liang-barsky clipping alogrithm

if (!myWindowDefined || !myLineDefined)

{

// 窗口或直线未定义，无法进行裁剪

return;

}

// 初始化参数

double t0 = 0.0, t1 = 1.0; // 参数化的线段端点

int dx = m\_ClippingLine[1].x - m\_ClippingLine[0].x;

int dy = m\_ClippingLine[1].y - m\_ClippingLine[0].y;

// Liang-Barsky参数计算

// 循环遍历窗口的四个边界

for (int i = 0; i < 4; i++)

{

int p, q;

// 计算p和q的值，用于计算参数r

if (i == 0)

{

p = -dx;

q = -(m\_window[0].x - m\_ClippingLine[0].x);

}

else if (i == 1)

{

p = dx;

q = m\_window[1].x - m\_ClippingLine[0].x;

}

else if (i == 2)

{

p = -dy;

q = -(m\_window[0].y - m\_ClippingLine[0].y);

}

else

{

p = dy;

q = m\_window[1].y - m\_ClippingLine[0].y;

}

// 计算r，它表示线段与当前边界的交点在整个线段上的相对位置

double r = static\_cast<double>(q) / p;

// 处理特殊情况：线段与当前边界平行且在边界之外

if (p == 0 && q < 0)

{

// 线段平行于裁剪边界且在裁剪边界之外，被裁剪掉

return;

}

// 更新参数t0和t1

if (p < 0)

{

if (r > t1)

{

// 线段在窗口之外

return;

}

else if (r > t0)

{

// 更新t0

t0 = r;

}

}

else if (p > 0)

{

if (r < t0)

{

// 线段在窗口之外

return;

}

else if (r < t1)

{

// 更新t1

t1 = r;

}

}

}

// 计算裁剪后的线段端点

int x0 = m\_ClippingLine[0].x + static\_cast<int>(t0 \* dx);

int y0 = m\_ClippingLine[0].y + static\_cast<int>(t0 \* dy);

int x1 = m\_ClippingLine[0].x + static\_cast<int>(t1 \* dx);

int y1 = m\_ClippingLine[0].y + static\_cast<int>(t1 \* dy);

// 更新裁剪后的线段

m\_ClippedLine[0] = CPoint(x0, y0);

m\_ClippedLine[1] = CPoint(x1, y1);

myClipDone = true;

}

* 选做部分代码实现：

**1-利用Glut处理鼠标、键盘输入的功能，实现用鼠标交互输入的方式来定义窗口、被裁剪线段的功能。**

在cohen sutherland clipping文件夹中，我实现了利用cohen-sutherland算法演示界面框架和OpenGL框架下编写程序。

在这个工程文件中，我通过keyboard函数实现了键盘输入的功能：

* 如果按下键盘上的数字键'1'，设置绘制模式为绘制直线（**draw = 1**）。
* 如果按下键盘上的数字键'2'，将直线计数器 **n** 和直线索引 **k** 重置为0，设置绘制模式为绘制矩形框（**draw = 2**）。
* 如果按下键盘上的Esc键（ASCII码为27），退出程序。
* 其他按键不进行特殊处理。
* 刷新显示，以便及时反映按键操作的变化。

在mymouse函数中，处理鼠标点击事件，根据绘制模式（draw）进行不同的操作。

* 如果鼠标左键按下（**GLUT\_DOWN**）且当前绘制模式是绘制直线（**draw == 1**），则记录直线的起始坐标。
* 如果鼠标左键抬起（**GLUT\_UP**）且当前绘制模式是绘制直线，记录直线的结束坐标，并输出直线的坐标信息。
* 如果鼠标左键按下且当前绘制模式是绘制矩形框（**draw == 2**），记录矩形框的起始坐标。
* 如果鼠标左键抬起且当前绘制模式是绘制矩形框，记录矩形框的结束坐标，并输出矩形框的四个顶点坐标信息。

在mymotion函数中，处理鼠标移动事件，根据绘制模式更新正在绘制的对象的结束坐标。

* 如果当前绘制模式是绘制直线（**draw == 1**），更新正在绘制的直线的结束坐标。
* 如果当前绘制模式是绘制矩形框（**draw == 2**），更新正在绘制的矩形框的结束坐标。

**2-直线剪裁算法的实现：**

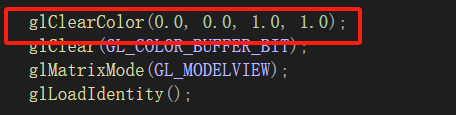
在mychip函数中，实现对于直线的剪裁：

具体的实现步骤如下：

1. **初始化矩形窗口边界**：通过 **boxXstart**、**boxYstart**、**boxXend** 和 **boxYend** 定义裁剪窗口的左下角和右上角坐标。
2. **初始化线段参数和边界参数**：
   * **d** 数组存储线段的斜率的倒数，即 pk，pk 是裁剪线段和窗口边界的交点的参数。
   * **q** 数组存储线段与窗口边界的关系，即 tpk ≤ qk 的条件。
3. **计算交点参数**：计算线段与裁剪窗口边界的交点参数 u1、u2、u3、u4。
4. **处理线段与X轴平行的情况**：
   * 如果线段与X轴平行，判断是否在窗口外。若在窗口外，则裁剪结束。
   * 否则，比较交点参数和起始点参数大小，更新裁剪区间。
5. **处理线段与Y轴平行的情况**：
   * 如果线段与Y轴平行，判断是否在窗口外。若在窗口外，则裁剪结束。
   * 否则，比较交点参数和起始点参数大小，更新裁剪区间。
6. **处理线段不平行于窗口的情况**：
   * 根据线段的斜率确定裁剪区间。
   * 更新裁剪区间。
7. **检查裁剪区间是否有效**：若 t1 大于 t2，则表示线段完全在窗口外，裁剪结束。
8. **计算裁剪后的线段端点**：根据裁剪区间 t1 和 t2，计算线段在窗口内的部分。
9. **使用OpenGL绘制裁剪后的线段**：调用 **glBegin(GL\_LINES)** 和 **glEnd()** 函数，设置颜色为红色，使用 **glVertex2f** 绘制裁剪后的线段。

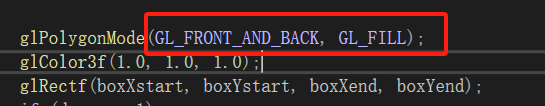
**3-改进算法界面：**

* 修改窗口的背景颜色为蓝色，使得窗口变得更加漂亮。



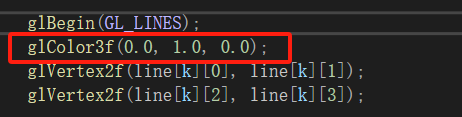
修改背景的B分量为1，不透明度为1

* 为了方便观察，修改矩形框的颜色为白色，填充为完全填充

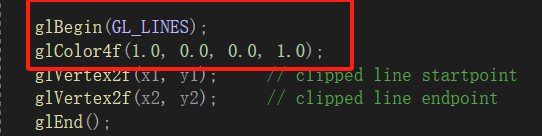


同时在display函数中利用上图中的函数绘制矩形，使得绘制的矩形不会因为模式的切换就消失。

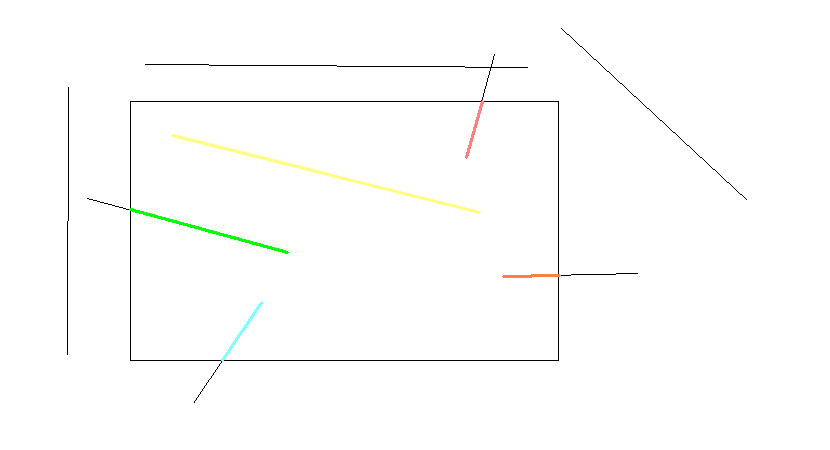
* 绘制的直线设置为绿色：

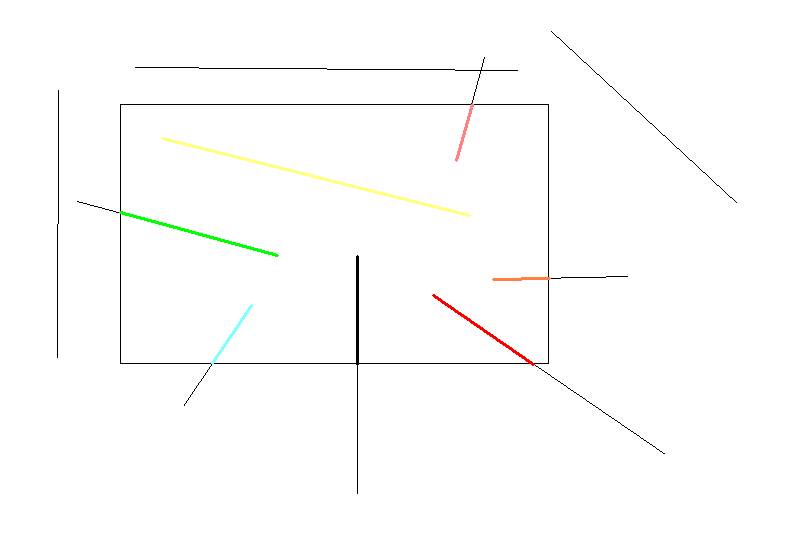


设置直线被剪裁的部分为红色：

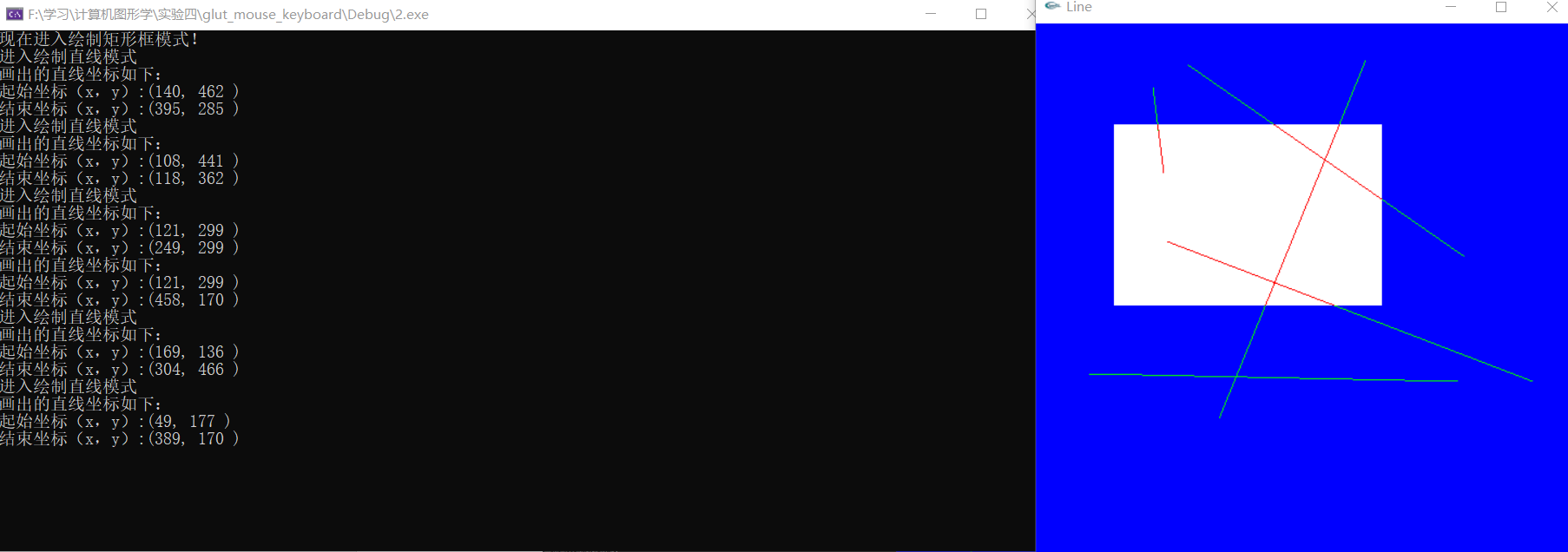


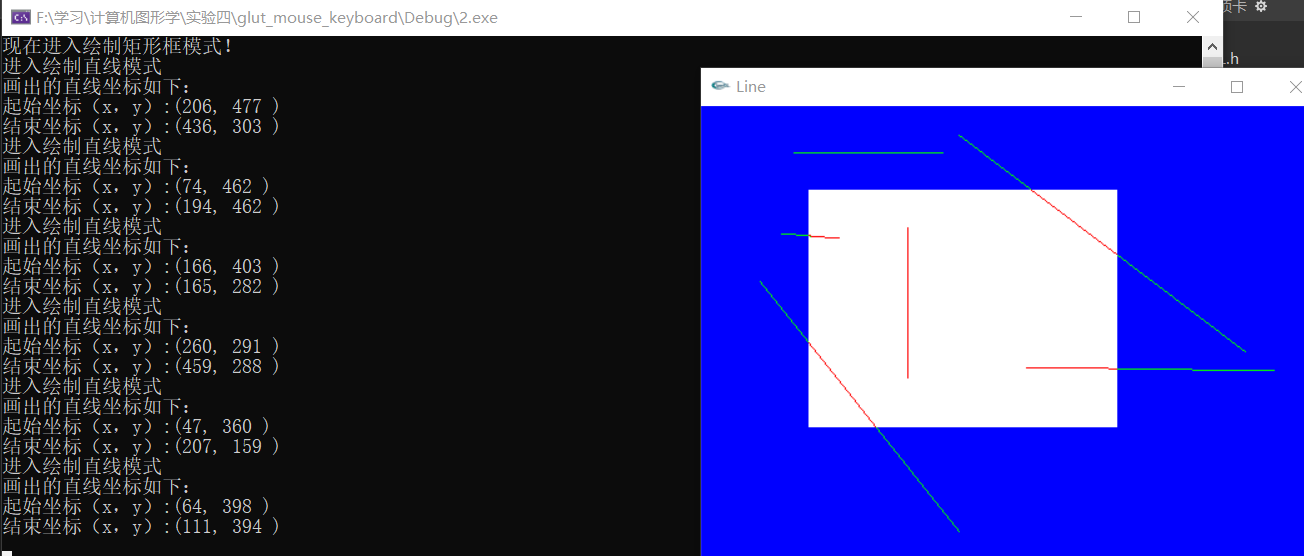
实验结果：





选做部分实现：





在切换模式的过程中，每一次切换绘制模式，在界面中就会提示进入的是绘制直线的模式还是绘制矩形的模式。

利用cohen-sutherland算法，实现了对于直线的剪裁。

结论分析：

通过以上实验结果可以看出，本次实验较完善的实现了liang-Barsky算法。完成了包括选作在内的所有内容。在实验中也遇到了一些问题，比如OpenGL一开始在绘制矩形后再绘制直线会清空原来的画面，我在display函数中增加了绘制矩形的代码，使得矩形框得以保留。

我在代码中添加了详细的注释，以后复习阅读起来更加清晰。

通过这次实验，我比较了liang-Barsky算法与Cohen—Sutherland算法：

1. 原理和适用范围：
   * Cohen-Sutherland算法：
     + 基于区域编码的原理，将窗口分为九个区域，使用二进制编码来表示线段与窗口边界的关系。
     + 适用于任意平行于坐标轴的矩形裁剪窗口。
   * Liang-Barsky算法：
     + 基于参数化的线段表示，在参数空间中进行裁剪。
     + 适用于任意裁剪窗口，不仅限于平行于坐标轴的矩形。
2. 参数化表示：
   * Cohen-Sutherland算法：
     + 使用线段的斜率信息进行裁剪。
   * Liang-Barsky算法：
     + 使用线段的参数表示，通过参数空间中的比较来确定裁剪区域。
3. 裁剪效率：
   * Cohen-Sutherland算法：
     + 使用二进制编码，可以快速确定线段与窗口边界的关系。
     + 算法简单，适用于快速剔除完全在窗口外的线段。
   * Liang-Barsky算法：
     + 需要进行参数计算和比较，相对于Cohen-Sutherland算法可能稍慢一些。
     + 但对于任意裁剪窗口的情况更灵活。
4. 处理平行于坐标轴的情况：
   * Cohen-Sutherland算法：
     + 对于平行于坐标轴的情况，特别高效。
   * Liang-Barsky算法：
     + 不局限于平行于坐标轴，适用于更一般的情况。
5. 精确性和灵活性：
   * Cohen-Sutherland算法：
     + 算法相对简单，但可能对于一些复杂情况需要额外处理。
   * Liang-Barsky算法：
     + 更灵活，适用于各种裁剪窗口形状，但相应地需要更多的计算。

创新：①在选做中，通过利用max和min函数，保证函数中判断矩形边界的时候永远读入的数据是先读入的数据较小，后读入的数据较大，避免在后期计算过程中遇到的问题。