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：

供参考：

一个用OpenGL和C/C++实现裁剪算法的程序框架clip.cpp。请注意其中标示的需添加算法的部分；

用GLUT、OPENGL进行鼠标键盘交互定义画出直线段、矩形的例子（glut\_mouse\_keyboard.rar）；

用MFC进行界面编程的cohen-sutherland算法演示例子（cohen-sutherland.rar）。

实现以下任务：

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。

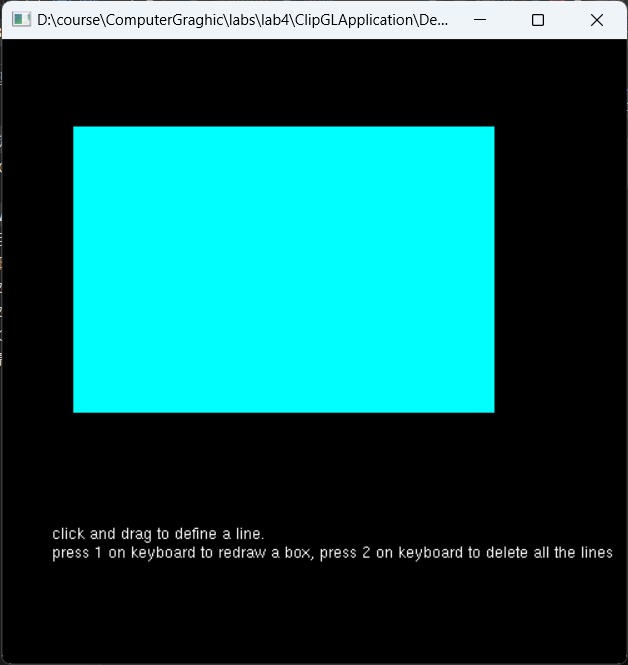
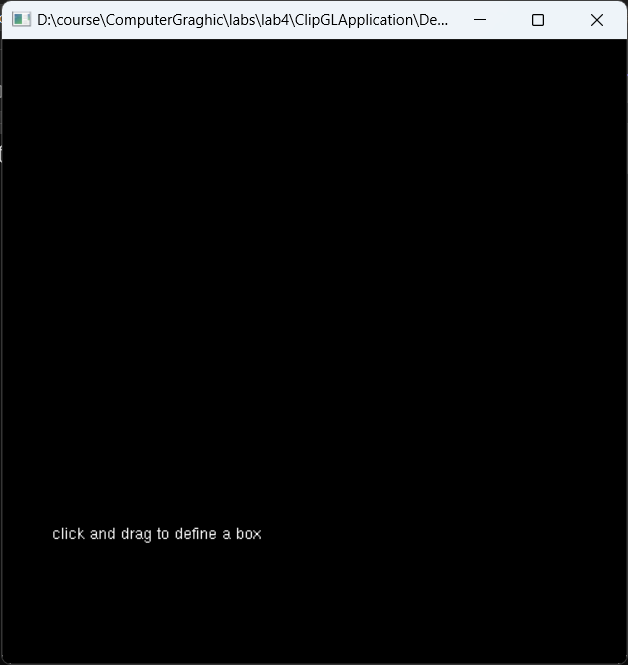
实现方法：

1. 实现功能概述

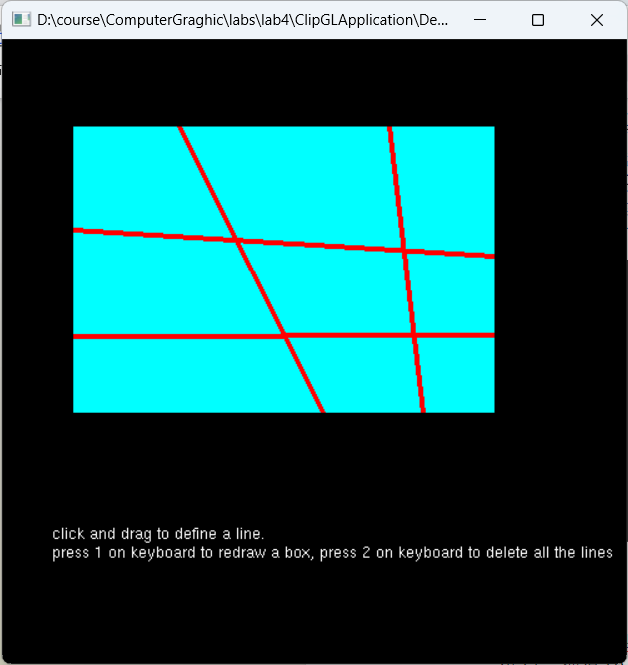
本实验中实现了以glut库为窗口处理框架的，用鼠标、键盘输入并具有交互效果的直线裁剪程序。环境配置为相对路径（解决方案同级的Dependences路径下），不必另外配置即可使用。

**使用说明如下：**

* 用鼠标**左键点击并拖动**便可以定义用来裁剪的**矩形**窗口（模拟渲染中的viewport）。



* 创建了矩形后，用鼠标**左键点击并拖动**创建**原始直线（白色）**。
* **松开左键**，进行直线裁剪操作，程序绘制**裁剪后的直线（红色）**保留在矩形中。



* 用户继续点击并拖动鼠标左键可以创建**更多直线**并自动裁剪。
* 当用户要**清除**当前矩形中保留的**直线**时，按下键盘上的**“2”键**，便可以清空直线重新绘制。
* 当用户需要**更改**当前设置的**矩形**窗口时，按下键盘上的**“1”键**，再次用鼠标左键点击并拖动便可以重新定义用来裁剪的矩形窗口，接下来创建直线的方法与前面一致。

1. 算法的说明

本实验采用Liang-barsky算法实现直线裁剪功能。

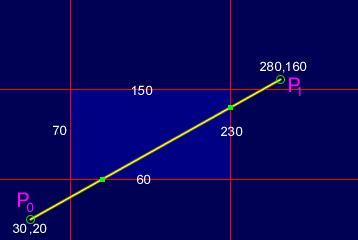
Liang-barsky直线裁剪算法的目的是找出两个在[0,1]区间上的值t0和t1，使得t0和t1能够表示被裁剪的直线线段的起始和终点值在原线段上所占的比例。（在[t0,t1]上的部分为被裁剪后保留部分）

下图为例，绿点所在的两点坐标分别设为(green\_x0, green\_y0),(green\_x1, green\_y1),p0坐标为(x0,y0),p1坐标为(x1,y1)，则：

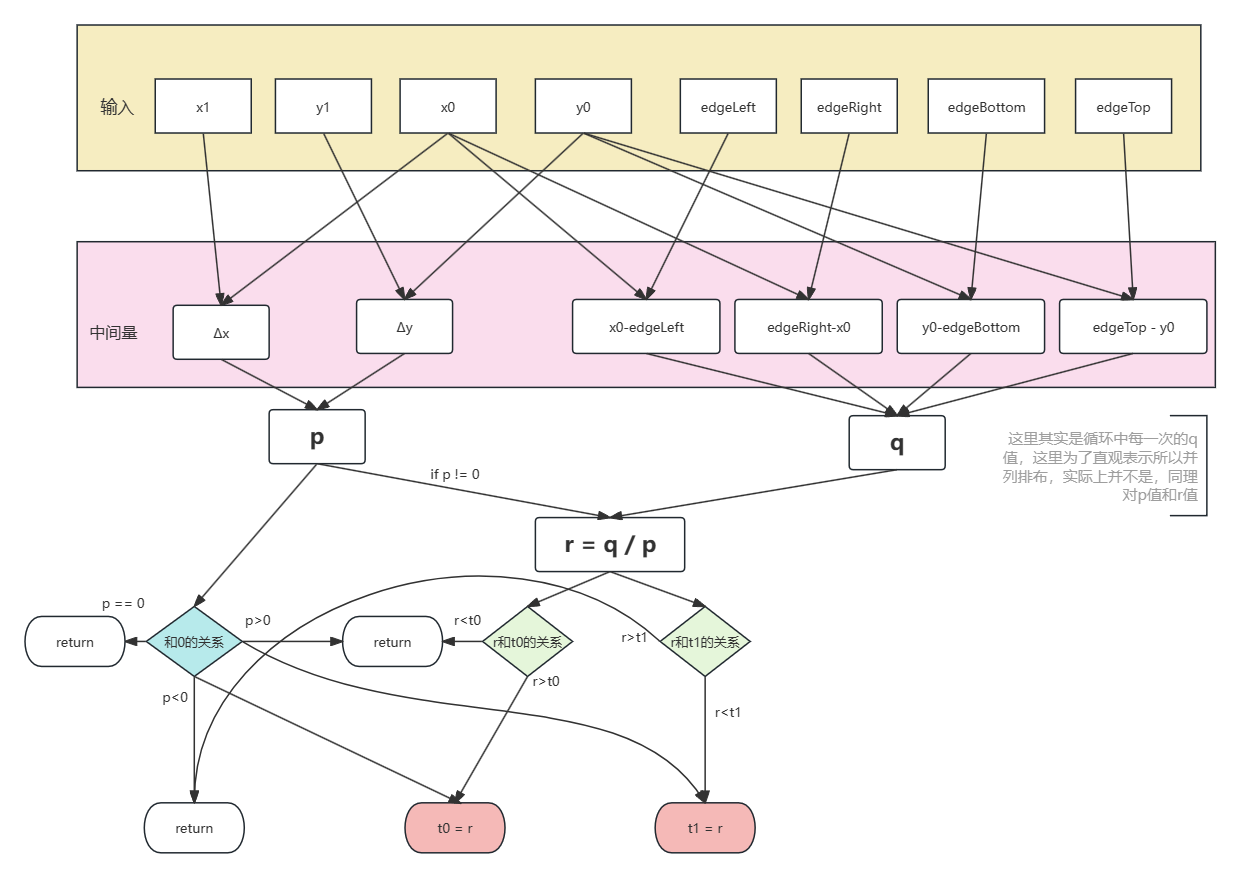
green\_x0 = x0 + t0 \* deltaX, green\_y0 = y0 + t0 \*deltaY;

green\_x1 = x0 + t1 \* deltaX, green\_y1 = y0 + t1 \*deltaY;

其中deltaX和deltaY分别为p0p1线段上横纵坐标在各自坐标轴上的增量。



我们设置矩形的四个边界量用来分别对左右上下进行直线裁剪。中间我们用p、q、r三个量来判断直线所在的位置。具体流程图如下：



为了理解我们用下面的具体例子解释：

1. 首先设置初始值t0 = 0, t1 = 1
2. 左边界检测：

p = -deltaX = -(280-30) = -250

q = -(edgeLeft-x0) = -(70-30)= -40

r = q/p = 0.16

此时p<0且r>t0,更新t0 = 0.16, t1 = 1



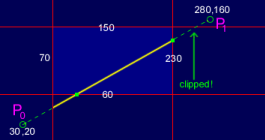
1. 右边界检测:

p = deltaX = (280-30)= 250

q = edgeRight-x0 = (230-30) = 200

r = q/p = 0.8

此时p>0且r<t1,更新t0 = 0.16, t1 = 0.8



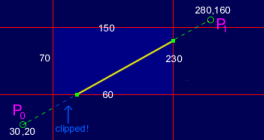
1. 下边界检测:

p = -deltaY = -(160-20) = -140

q = -(edgeBottom-y0) = -(60-20) = -40

r = q/p = 0.2857

此时p<0且r>t0,更新t0 = 0.2857,t1 = 0.8



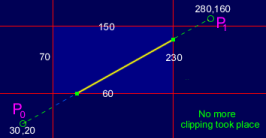
1. 上边界检测:

p = deltaY= 160-20= 140

q = edgeTop-y0= 150-20 = 130

r = q/p = 0.9286

此时p>0,但是且r>t1，不更新t1值。



1. 最终的结果为：

t0 = 0.2857

t1 = 0.8

1. Liang-barsky算法的代码实现：
2. 获取矩形边界和线段端点坐标：

获取矩形的左、底、右、上边界（edgeLeft, edgeBottom, edgeRight, edgeTop）。

获取线段起始点和终点的坐标（x0, y0, x1, y1）。

//the viewport edge

float edgeLeft = rect[0] > rect[2] ? rect[2] : rect[0];

float edgeBottom = rect[1] > rect[3] ? rect[3] : rect[1];

float edgeRight = rect[0] > rect[2] ? rect[0] : rect[2];

float edgeTop = rect[1] > rect[3] ? rect[1] : rect[3];

//the start and end point of the original line

float x0 = line[0];

float y0 = line[1];

float x1 = line[2];

float y1 = line[3];

1. 初始化变量和参数：

初始化裁剪参数 t0 和 t1 为 0 和 1，用于记录线段的裁剪比例。

计算线段的增量 deltaX 和 deltaY，以及参数 p, q, r。

float t0 = 0.0f, t1 = 1.0f;

float deltaX = x1 - x0;

float deltaY = y1 - y0;

float p, q, r;

1. 遍历矩形边界进行裁剪：

使用循环遍历矩形的四条边（左、右、底、上）。

对于每条边，计算前面设定的参数p，q，r，根据这些参数判断线段是否在矩形外部，若在外部则直接返回 false，表示线段被完全裁剪。如果不在外部则更新t0和t1的值。

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

{

if (edge == 0) { p = -deltaX; q = -(edgeLeft - x0); } //left clip

else if (edge == 1) { p = deltaX; q = (edgeRight - x0); }//right clip

else if (edge == 2) { p = -deltaY; q = -(edgeBottom - y0); }//bottom clip

else if (edge == 3) { p = deltaY; q = (edgeTop - y0); }//top clip

r = q / p;

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

return false;

}

else if (p < 0) {

if (r > t1) {

return false;

}

else if (r > t0)

t0 = r; //replace t0 with r

}

else if (p > 0) {

if (r < t0) {

return false;

}

else if (r < t1)

t1 = r;//replace t1 with r

}

}

1. 计算裁剪后线段的端点坐标：

根据参数 t0 和 t1 计算裁剪后的线段端点坐标。

将计算出的新端点坐标存储在 clip[index][0], clip[index][1], clip[index][2], clip[index][3] 中。

返回 true，表示线段在矩形内部或者与矩形有交点，成功进行了裁剪。

clip[index][0] = x0 + t0 \* deltaX; //new x0

clip[index][1] = y0 + t0 \* deltaY; //new y0

clip[index][2] = x0 + t1 \* deltaX; //new x1

clip[index][3] = y0 + t1 \* deltaY; //new y1

return true;

完整代码见附件ClipGLApplication.cpp。

1. 代码中所采用的数据结构

在上面的算法中未采用特殊的数据结构，下面对整体代码中的部分全局变量做一些说明：

int \*\*line; // original lines

int \*\*clip; //clip lines

line 和 clip 是二维数组，用于存储线的端点坐标。二维数组的每一行为一个线段，一开始设置最大可绘制线段数为50条，每个线段用四个整数表示，即起点和终点的 x、y 坐标。

另外设置了int型变量n记录现有的线段条数，每当绘制一条直线开始的时候，就将n++，一开始的n初始化为0，所以有效的线段编号从1开始。

n也充当了index的作用，方便在当前线段line[n]绘制的时候对其进行处理以及计算出clip[n]的四个值。

因为glut窗口上display是对每一帧进行绘制，如果每次绘制新线段的时候只绘制当前线段，那么就无法显示之前的线段，所以在display函数调用的drawLine函数中设置了下标从1到n的循环来绘制从开始到当前绘制的所有线段（裁剪后）。

这两个数组（或者说指针）的初始化分配空间在ReallocateLine()中进行，此外如果按下ketboard的2键选择清空绘制的直线线段后，n重置为0，调用ReallocateLine()使得两个指针被重新分配内存。

bool lineStart; //state of line has defined the start point but with undifined end point

bool lineEnd; //state of that the current line draw process has complete

bool RectExist; //state of undefined viewport(rectangle/box)

为了设置每个时刻的绘制状态，设置了lineStart, lineEnd, RectExist 这些布尔变量来用于跟踪线段和矩形的状态，以确定绘制线段或执行裁剪操作。

其中lineStart和lineEnd是为了跟踪直线绘制过程中的操作，lineStart为true 的时候代表正在绘制原始直线，lineEnd为true的时候代表已经绘制完毕，接下来就可以进行裁剪操作和裁剪后的绘制新线段操作。

RectExist是为了控制先绘制定义裁剪区域的矩形后才能进行直线绘制，在RectExist为false的时候，控制鼠标点击并拖动为绘制矩形，绘制完毕后设置RectExist为true，接下来鼠标点击并拖动就为绘制直线的操作了。

在keyboard的1按下后，RectExist重置为false，这样可以定义新的矩形裁剪窗口。

1. 分模块说明代码中各函数的功能
   1. **初始化：**

**myinit()：**初始化。

该函数主要是设置了 OpenGL 的一些渲染和清空设置，并初始化了程序运行时的状态，包括全局变量的初始值和线段数组内存的分配。

lineStart = false;、lineEnd = false;、RectExist = false;、n = 0;设置了程序运行时的一些初始状态。lineStart、lineEnd 和 RectExist 是布尔变量，用于追踪线段和矩形的状态；n 是用于追踪绘制的线段数量的计数器，初始值为0。之后调用了 ReallocateLine() 函数，动态分配内存给存储线段和裁剪结果的数组line和clip。

**ReallocateLine()**：动态分配内存给line和clip数组。

new 操作符在堆内存中动态地分配内存。

如果先前已经分配了内存，这个函数会重新分配内存以防止内存泄漏或者重新初始化数组。

这两个函数一起确保了程序运行所需的初始状态和内存分配，为绘制和处理线段以及裁剪提供了基础。

* 1. **鼠标键盘处理函数：**

**keyboard()：**处理键盘输入。

当按下 '1' 键时，会重置程序状态，设置RectExist为false，n设置为0，线段数组重新分配内存，接下来用户可以重新绘制矩形和线段。

当按下 '2' 键时，会清除所有已绘制的线段，但保留矩形，具体操作为n设置为0，线段数组重新分配内存。

当按下 'Esc' 键时，会退出程序。

**mymouse()：**处理鼠标点击事件。

button 和 state 参数表示了鼠标按钮的状态，包括按下和释放，以及是左键、右键等等。x 和 y 参数表示鼠标点击事件发生的坐标位置。

根据鼠标点击操作记录矩形的边界或者线段的起始点和终点，具体如下。

按下左键时：如果还没有定义矩形，根据鼠标点击位置定义矩形的左下角坐标（rect[0] 和 rect[1]）；如果已经定义了矩形，先将计数器n++，创建新线段，并根据鼠标点击位置定义线段的起始点坐标（line[n][0] 和 line[n][1]）。

释放左键时：如果还没有定义矩形，根据鼠标释放位置定义矩形的右上角坐标（rect[2] 和 rect[3]），表示矩形的另外一个角；如果已经定义了矩形，根据鼠标释放位置定义线段的终点坐标（line[n][2] 和 line[n][3]），同时更新线段的状态。

处理完上面的操作后调用 glutPostRedisplay() 函数重新绘制窗口内容。

**mymotion()**：处理鼠标移动事件，记录矩形的边界或者线段的终点。

捕捉鼠标移动，产生绘制过程中的交互动态显示效果。

如果还没有定义矩形，根据鼠标移动的位置更新矩形的右上角坐标（rect[2] 和 rect[3]），这样可以动态地调整矩形的大小。

如果已经定义了矩形，根据鼠标移动的位置更新当前线段的终点坐标（line[n][2] 和 line[n][3]），这允许用户在绘制线段时动态地调整线段的终点。

这三个函数用于捕捉用户的键盘和鼠标操作，实现对程序中图形和状态的交互操作。

* 1. **显示和绘制函数：**

**display()：**显示整个场景

glClear(GL\_COLOR\_BUFFER\_BIT)清除颜色缓冲区，准备绘制新的图形。调用 drawbox() 函数绘制代表裁剪窗口边界的矩形。根据程序状态调用 drawlines() 函数绘制线段，同时执行线段裁剪（调用 myclip() 函数）并绘制裁剪后的线段。

实现用户界面的操作友好性，根据程序的状态准备要显示的文本信息。如果矩形尚未定义，则显示提示用户拖动鼠标以定义矩形的消息，否则显示提示用户拖动鼠标以定义线段的消息，并显示删除线段和重绘矩形的键盘操作说明。

**drawlines()**：绘制线段，进行线段裁剪

**绘制原始线段**（如果线段处于定义状态）：如果 lineStart 为真，则以白色和指定的线宽绘制当前正在绘制的线段。使用 glBegin(GL\_LINES) 开始绘制线段，指定起始点和终点坐标，并使用 glEnd() 结束绘制。

**裁剪线段**（如果线段处于结束状态 lineEnd）：调用 myclip() 函数对当前线段进行裁剪。如果线段被完全裁剪，则将裁剪后的线段的起始点设置为 -1，否则输出裁剪后线段的坐标信息。

**绘制裁剪后的线段**：遍历已经绘制的线段（从 1 到 n），检查裁剪后线段的坐标是否有效（不等于 -1）。如果线段有效，则以红色和指定的线宽绘制裁剪后的线段，使用 glBegin(GL\_LINES) 和 glEnd() 指定线段的起点和终点坐标。

**drawbox()：**绘制矩形，表示裁剪窗口的边界。

设置绘制矩形的颜色后，glBegin(GL\_POLYGON) 函数开始绘制多边形（矩形），glVertex2f() 函数定义矩形的四个顶点（ rect 数组中存储的矩形的左下角和右上角坐标），然后用glEnd() 结束绘制。

这三个函数完成了整个绘图和显示的过程，在窗口中绘制矩形、线段以及执行线段的裁剪操作，并将这些内容显示在程序的图形界面上。

* 1. **裁剪函数：**

**myclip()：**实现线段裁剪算法。

基于线段与矩形边界的位置关系，确定线段是否在矩形内部或者是否与矩形相交。如果线段在矩形内部，计算裁剪后的线段端点坐标。根据判断结果，返回布尔值（true 或 false）表示线段是否在矩形内部，并且更新裁剪后的线段端点坐标。具体算法解释在上文的Liang-barsky算法的代码实现中。

* 1. **其他函数：**

**myReshape()：**处理窗口大小变化，设置 OpenGL 的视口和投影矩阵。

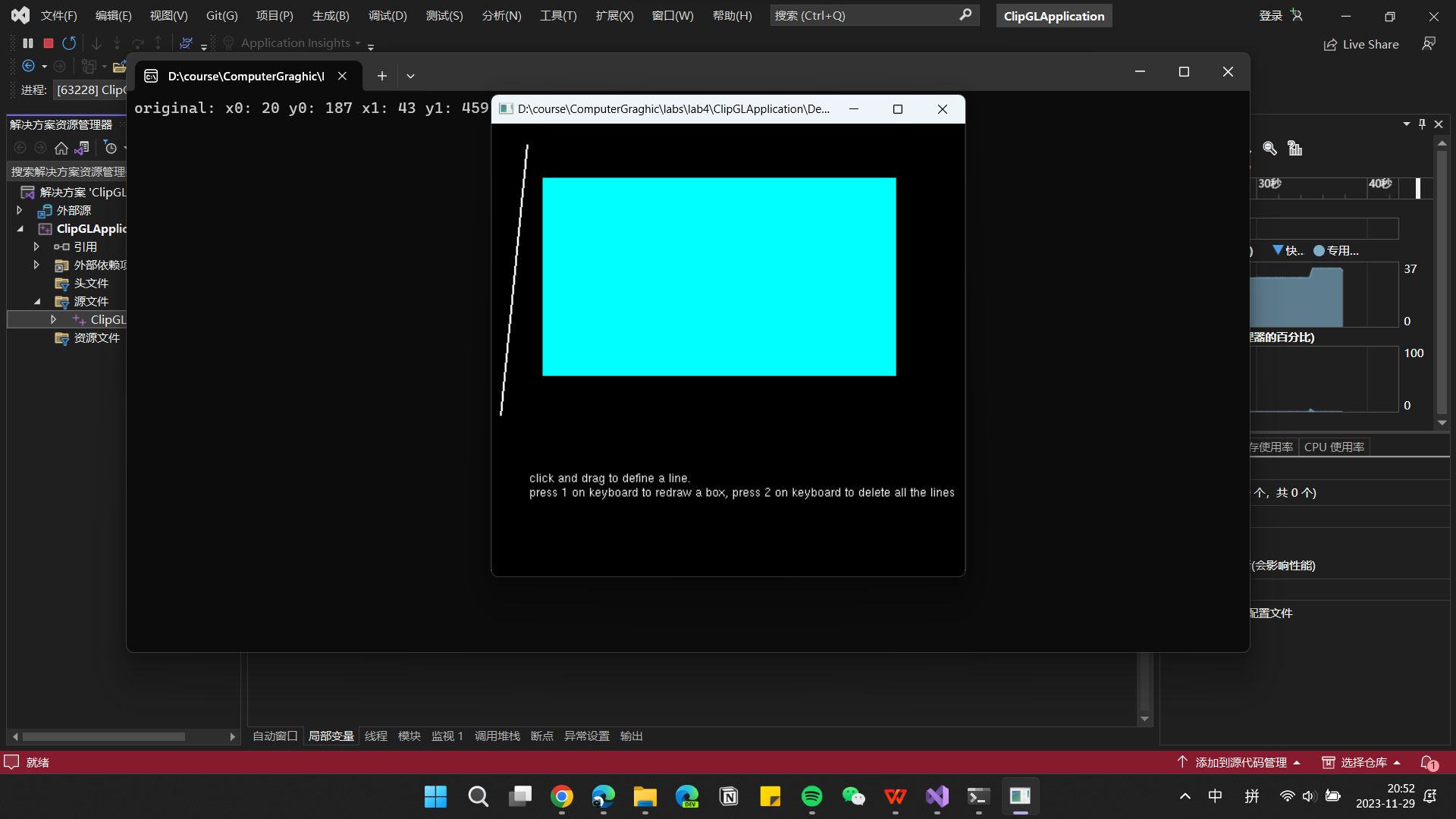
根据新的窗口大小重新设置 OpenGL 的投影矩阵，以确保渲染的内容能够正确地适应新的窗口尺寸。保证在调整窗口后，所绘制的对象的坐标值保持不变。

实验结果：

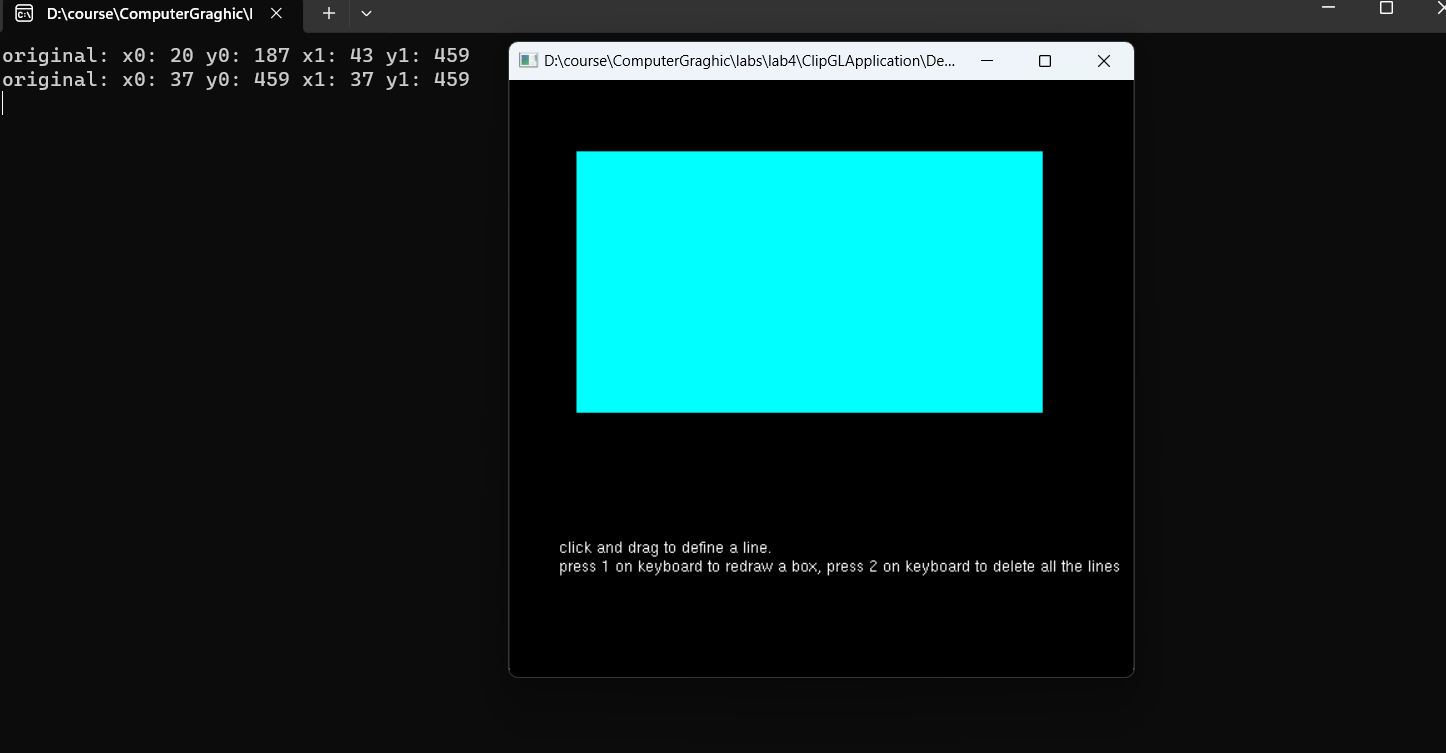
针对各种情况进行算法检验。

1. **在矩形外的情况。**

原线段绘制。

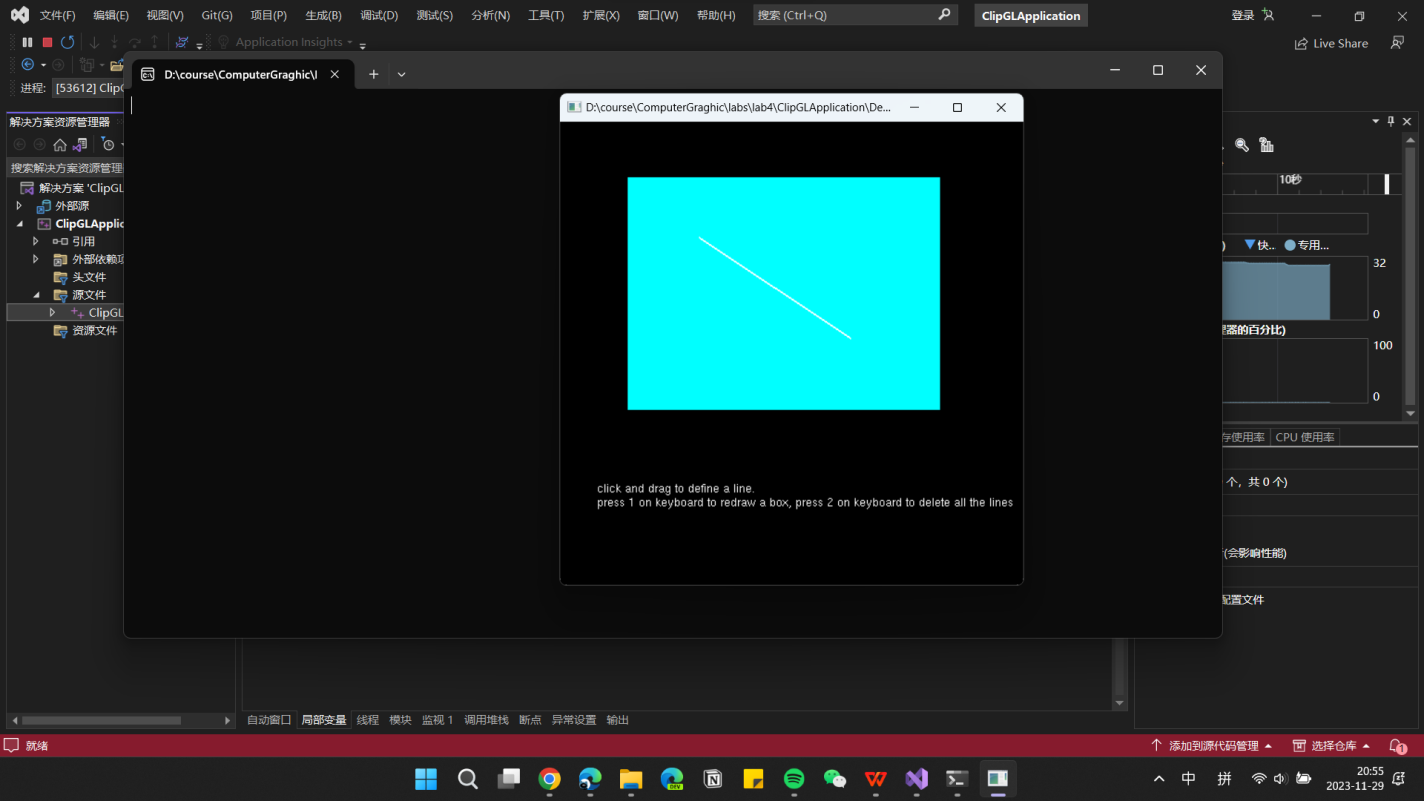


裁剪后，控制台并没有输出clip相关信息，且矩形中并无显示。

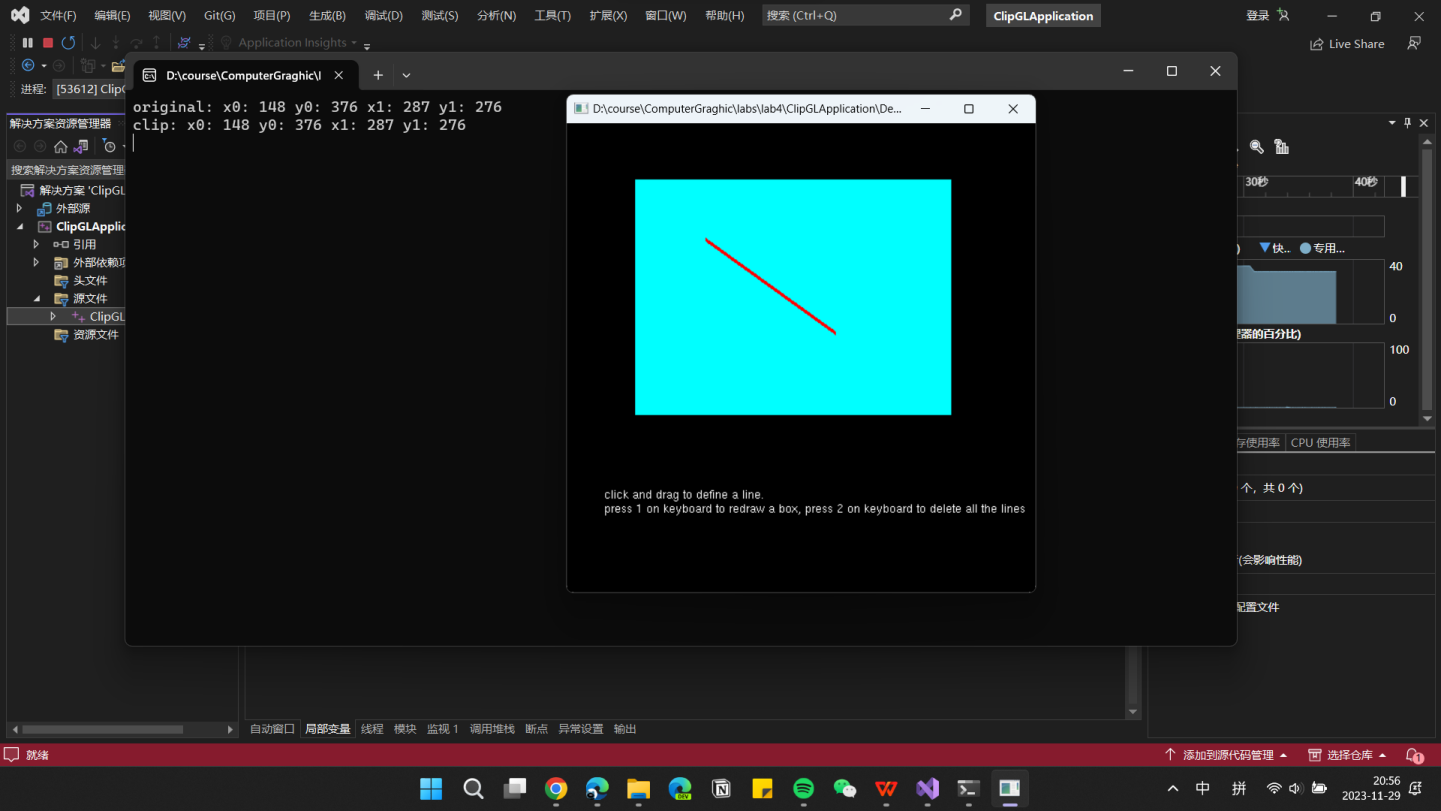


1. **在矩形内部**

原线段绘制。

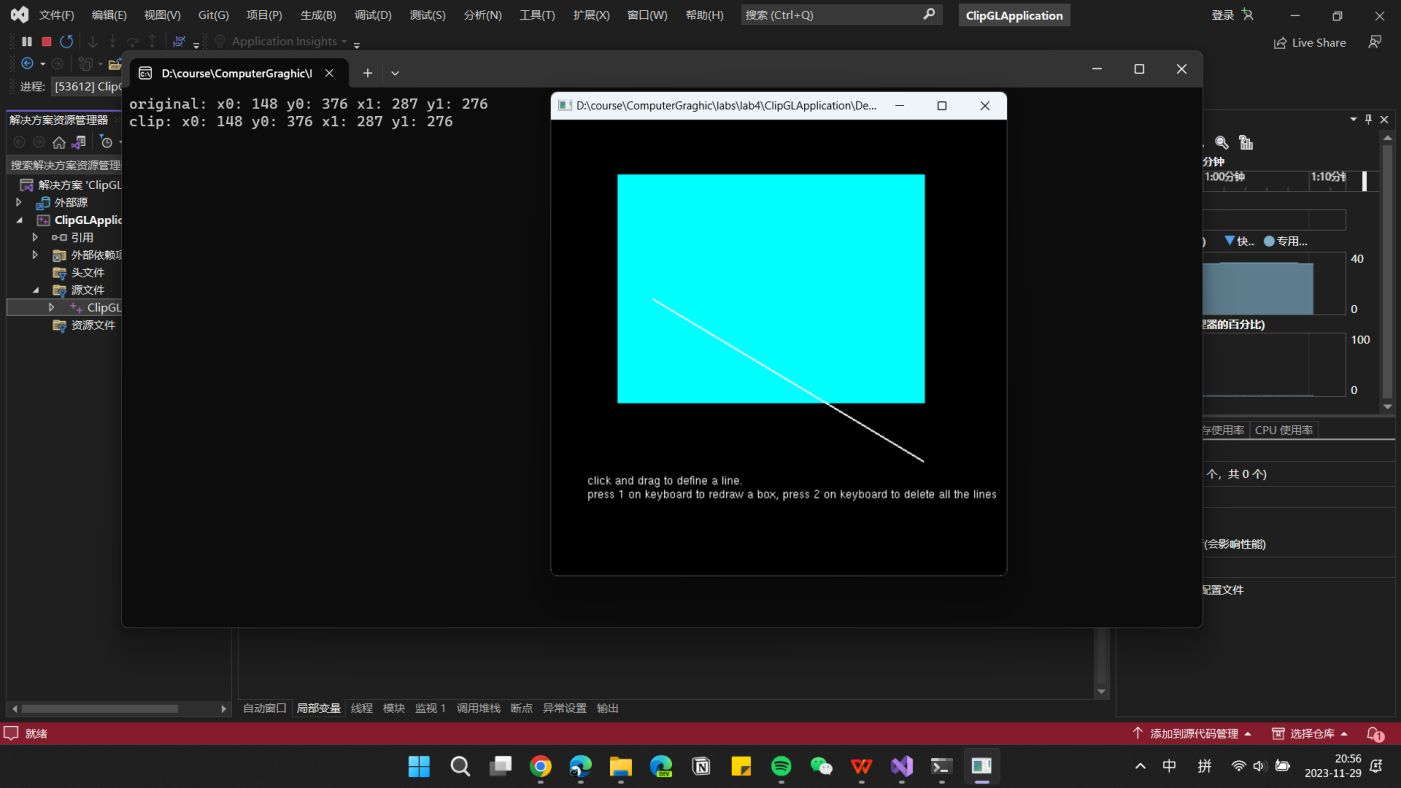


裁剪后结果，可以看到控制台中输出的clip线段的坐标和矩形上绘制的clip线段。

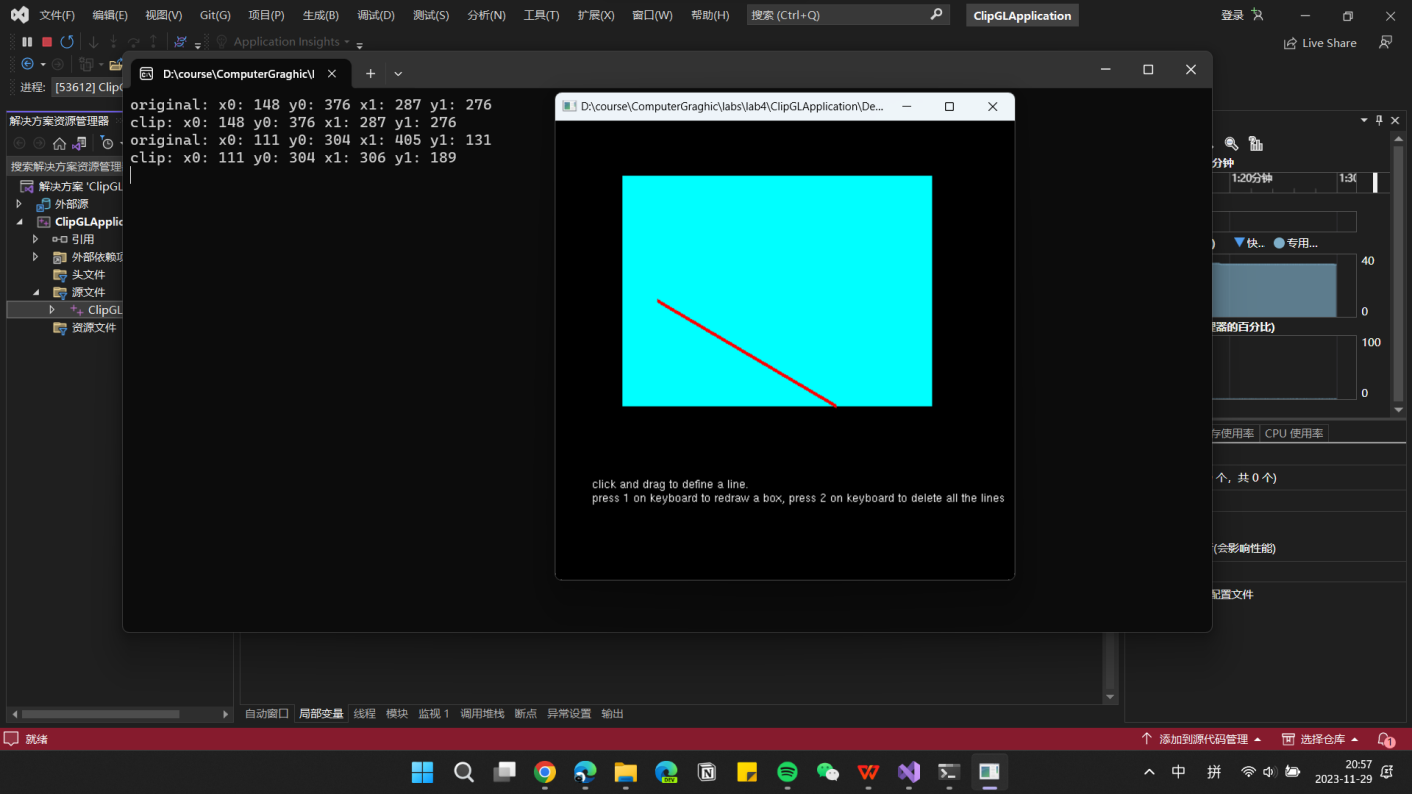


1. 一端在外部一端在内部

原线段绘制。

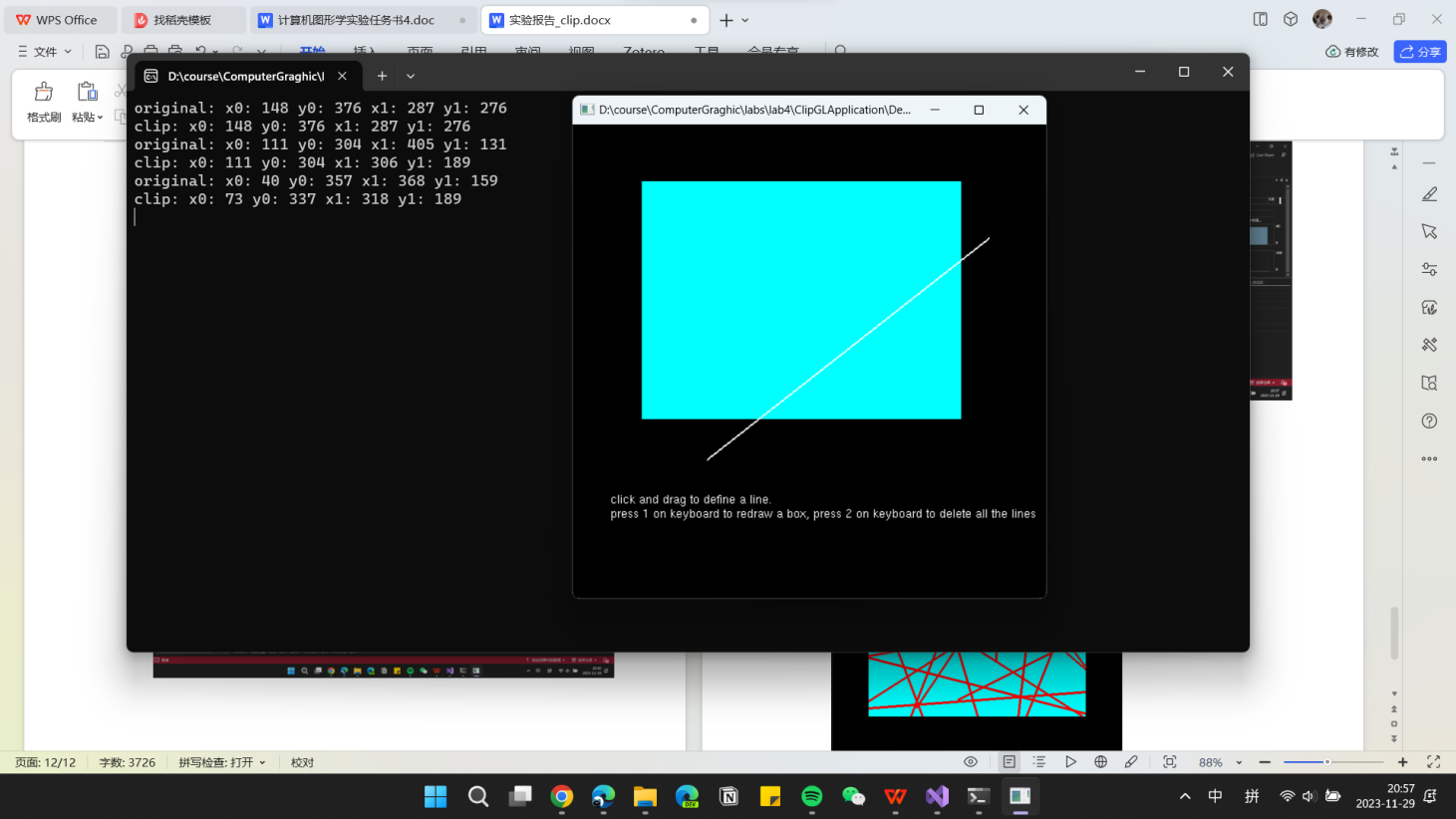


裁剪后结果，可以看到控制台中输出的clip线段的坐标和矩形上绘制的clip线段。

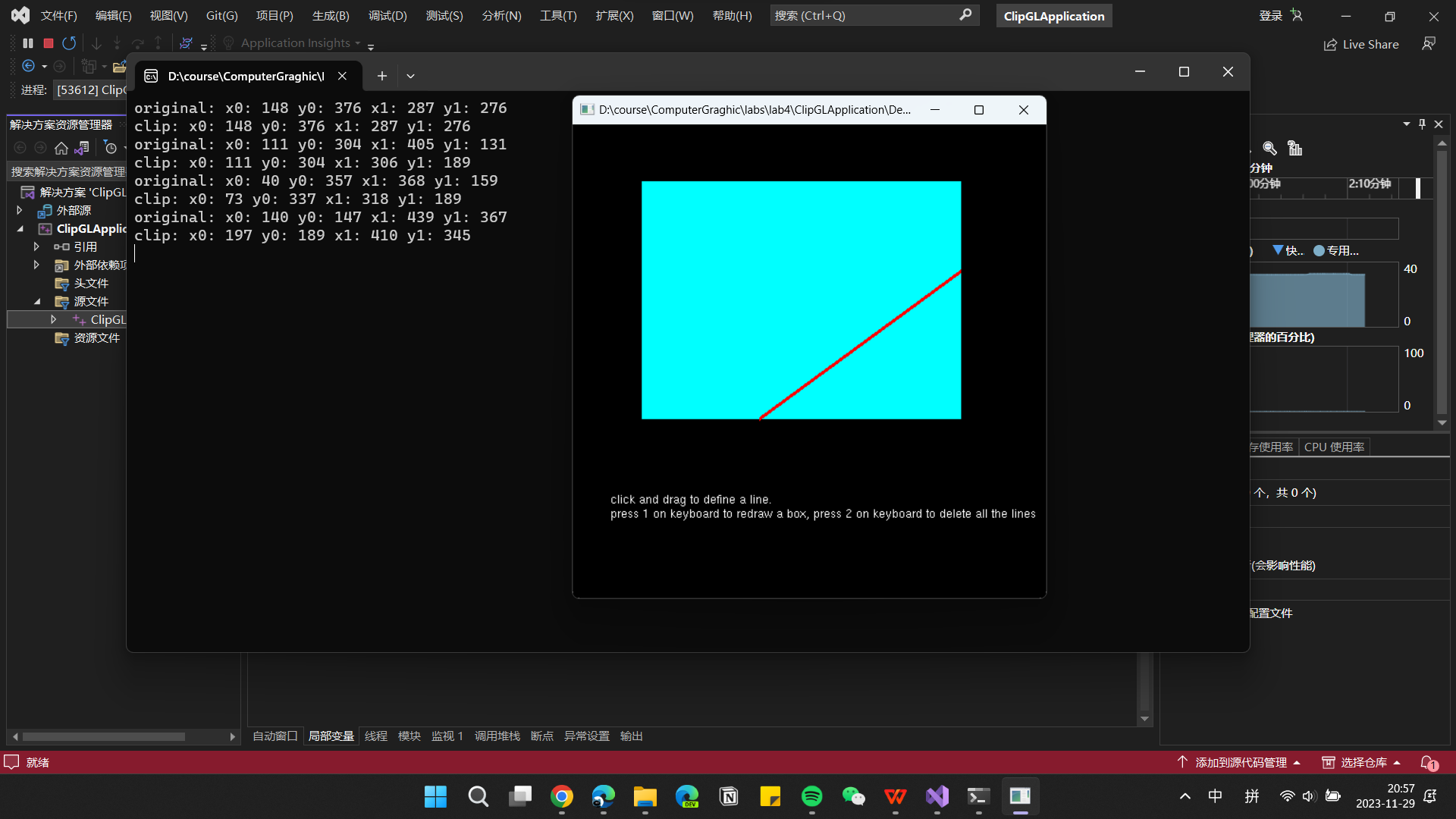


1. 两端都在外部

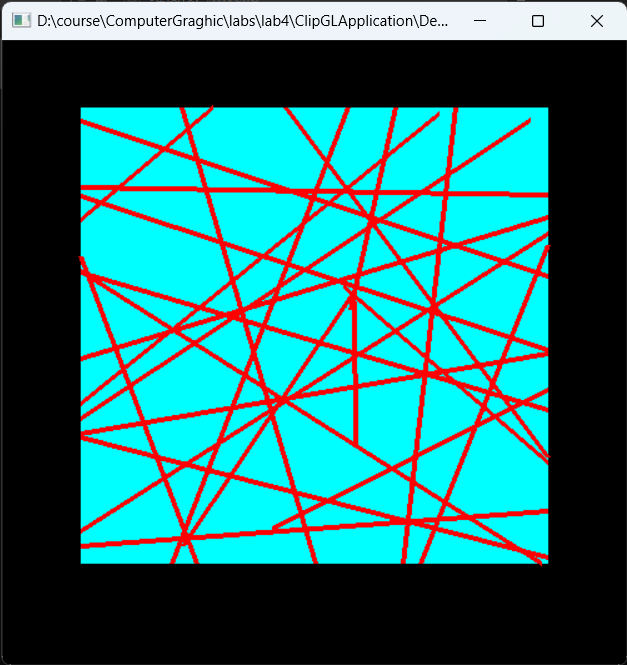
原线段绘制。



裁剪后结果。



1. 多条直线的绘制和裁剪



结论分析：

Liang-Barsky 算法高效地裁剪直线，适用于实时图形界面的交互式应用。

能够清晰地展示裁剪前后直线的效果，用户交互体验可以进一步优化，提供更多功能和操作选项。

在提供的框架上，我做了如下几点改动创新：

1. 将窗口中定义点的方式改用实际的x、y坐标而不采用比例的形式，在reshape函数中也做了对应更改，使得窗口改变后所有绘制物体的坐标值不变。
2. 实现了鼠标和键盘的交互功能，实现了自定义的操作逻辑，使得用户选择能实现的功能更多。
3. 增加了用户提示文本，提高了图形界面用户使用友好型。

参考文献和部分图片来源：

https://www.skytopia.com/project/articles/compsci/clipping.html