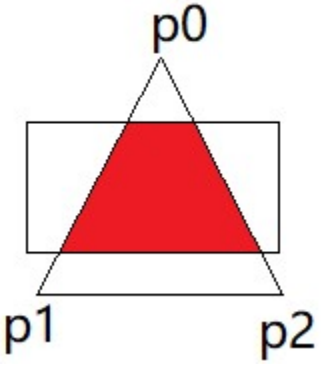
三、裁剪：

裁剪顾名思义就是将多边形裁剪掉一部分(甚至全部)而只保留多边形的一部分(或者全部)，如下面所示：



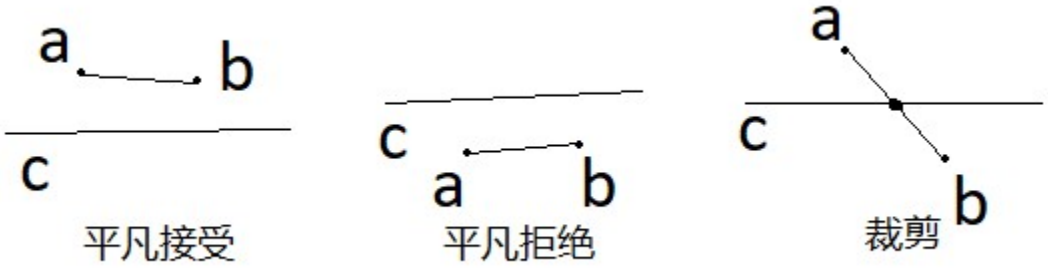
一个多边形<p0,p1,p2>被一个矩形所裁剪，红色部分就是裁剪剩下的部分，其中多边形<p0,p1,p2>叫做被裁剪多边形，用来裁剪的多边形叫做裁剪窗口。为什么要裁剪呢，如果一个多边形我们只想绘制出他的一部分，这时候就需要裁剪了，假设上图中的矩形是屏幕，超出屏幕外面的部分肯定是无法绘制的，甚至在上一章的代码中会导致程序崩溃，因为 NET 的取值范围是[0,屏幕高度]，所以对于超出屏幕范围的部分会无法访问 NET 的相应位置，所以需要对多边形进行裁剪。

1：出点和入点:

假设一个多边形被另一个多边形(或者直线)所裁剪，如果有交点，则必然是成对出现的，其中从被裁剪多边形进入裁剪窗口(或裁剪直线)内部的交点称之为“入点”，从被裁剪多边形离开裁剪窗口(或裁剪直线)的交点叫做出点。

2：直线对线段的裁剪：

直线对线段的裁剪很简单，首先判断线段的两个端点是否在直线的两侧，如果在同侧则说明该线段要么被全部抛弃(平凡拒绝)，要么被全部接受(平凡接受)，如果两个点在直线的两侧才需要裁剪，如下图，现在用一条直线 c 对线段<ab>进行裁剪，保留 c 的上面区域：



如果 ab 两点都没被裁剪掉，则为平凡接受，不需要裁剪。如果 ab 两点都被裁剪掉，则为平凡拒绝，不绘制该线段。如果 ab 两点只被裁剪掉任意一点，需要求出<ab>和直线 c 的交点。裁剪条件需要根据具体情况写出方程，通常条件会为$Ax + By + C > 0$或者$Ax + By + C < 0$等可以表示直线某一侧的方程作为裁剪边界。

直线的交点只需要联立他们的方程组即可解出。设线段<ab>和 c 的方程分别为：

$$\begin{cases} A_1x + B_1x + C_1 = 0 \\ A_2x + B_2x + C_2 = 0 \end{cases}$$

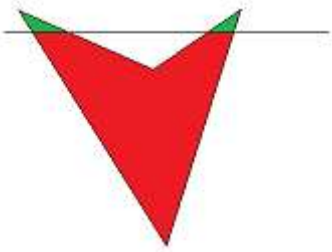
此时联立上面两个方程组可以解出交点 p 的坐标为

$$\left(\frac{B_1 * C_2 - B_2 * C_1}{A_1 * B_2 - A_2 * B_1}, \frac{A_2 * C_1 - A_1 * C_2}{A_1 * B_2 - A_2 * B_1}\right)$$

这时候如果被裁剪掉的是 A 点，则剩下的线段为<pB>，否则剩下的线段为<Ap>。

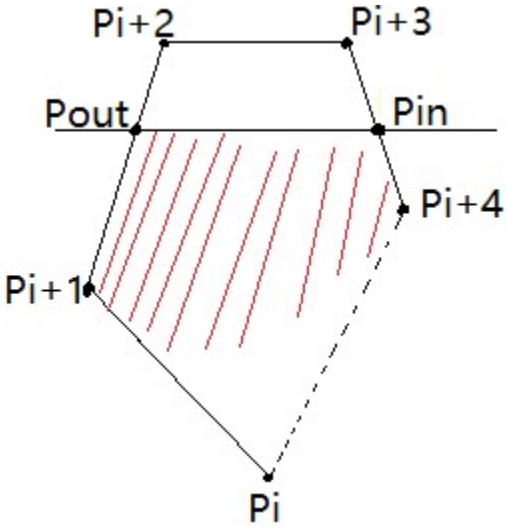
3：直线对多边形的裁剪：

如下图所示，现在用一条直线裁剪一个多边形，直线上面的部分将会被裁剪掉(绿色部分)，保留下面的部分(红色)：



直线裁剪多边形方法如下：

如果多边形的一部分被直线所裁剪，多边形和直线的交点必然是成对出现的，这里的每一对交点是由出点和入点组成，每一对的出点和入点会连接成一条新的边添加到被裁剪三角形中，平凡拒绝的边不保留，平凡接受的边原样保留。考虑一下如下多边形的一部分，红色阴影是被裁剪之后保留的部分(Pi 和 Pi+4 之间用虚线连接表示他们之间可能还有很多点，但是这些点所组成的边都被平凡接受，可以和边<Pi,Pi+1>采用一样的处理方法)：



假设现在用一个列表 points 记录裁剪阴影部分的顶点，则列表中的顶点依次为：Pi+1,Pout,Pin,Pi+4,……,Pi。这个列表可以由下面的规则给出：

假如我们在 points 中记录最终图形每条边的终点(记录起点的方法也类似，可自行思考)，从顶点 Pi 到 Pi+4(反过来从 Pi+4 到 Pi 也行，但是上图中的 Pin 就会变成 Pout，上图中的 Pout 也会变成 Pin)，依次扫描每条边。则每当遇一条被平凡接受的边时记录其终点，如<Pi,Pi+1>记录 Pi+1。当一条边被裁剪时，裁剪点为出点，则记录新出点，如边<Pi+1,Pout>记录 Pout。当一条边的裁剪点为入点时，先记录新的入点，这样入点就会先和之前的出点形成一条新边，如边<Pout,Pin>，再记录边终点，如边<Pin,Pi+4>。当一条边被平凡拒绝时，不记录这条边的信息，如边<Pi+2,Pi+3>。

伪代码如下：

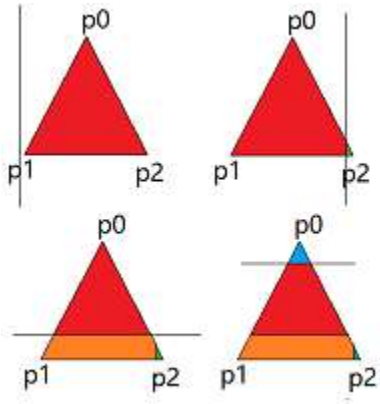
```
list points;//存放裁剪后多边形的顶点
int count=多边形顶点个数
```

```
for 多边形的每个顶点 Pi://存放每条边的终点
{
    if 边<P[i],P[(i+1)%count]>被平凡接受:
    {
        point.push_back(P[(i+1)%count]);//存放此条边的终点
    }
    else if 边<Pi,P[(i+1)%count]>被裁剪:
    {
        if 被裁剪掉 Pi://入点
        {
            points.push_back(新的 Pi 点);//此点和前一个出点组成一条新边
            point.push_back(P[(i+1)%count]);
        }
        else if 被裁剪掉 P[(i+1)%count]://出点
        {
            points.push_back(新的(i+1)%count 点);
        }
    }
    //剩下情况就是边被平凡拒绝，不记录此边任何信息
}
```

4：凸多边形裁剪窗口的裁剪(Sutherland-Hodgman):

多边形的裁剪算法有很多中，下面我只介绍以下适用于凸多边形裁剪窗口的 Sutherland-Hodgman 算法，因为这个算法的思想在后面的章节中我们还会用到。

Sutherland-Hodgman 算法的核心思想是将裁剪窗口分解为一条一条的直线，然后使用这些直线逐条的去裁剪多边形(不管直线使用顺序，只要使用了每一条直线即可)，这样最后剩下的就是该裁剪窗口所裁剪的多边形。



如图所示，这里将裁剪窗口的所有边分解成直线，然后使用这些直线依次的对被裁剪多边形进行裁剪，被裁剪多边形最后剩下的部分即为被该裁剪窗口所裁剪剩下的图形。

Sutherland-Hodgman 只适用于凸多边形的裁剪窗口(被裁剪多边形凹凸都行)。

5：扩充知识 1：判断两个顶点在直线的同一侧：

这里可以用很简单的线性规划知识来判断，假设直线方程为 Ax+By+C=0,两点坐标分别为$(x_1, y_1), (x_2, y_2)$，则有下面的判断条件：

$$\begin{cases} (Ax_1 + By_1 + C) * (Ax_2 + By_2 + C) > 0 & \text{在同一侧} \\ (Ax_1 + By_1 + C) * (Ax_2 + By_2 + C) < 0 & \text{在直线两侧} \\ (Ax_1 + By_1 + C) * (Ax_2 + By_2 + C) = 0 & \text{至少有一点在直线上} \end{cases}$$

6：扩充知识 2：

对于凸多边形的任意一条边，剩下的所有顶点都在这条的同一侧。

下面我将会放两个凸多边形作为裁剪窗口的代码,都使用了 Sutherland-Hodgman 算法裁剪：

1. 任意凸多边形裁剪窗口(chapter3/clip1)：

   首先完成直线对直线的裁剪，函数 clipLL 实现了该功能

   在 clip 函数中调用 clipLL 对多边形进行裁剪，其中参照点选取了不在裁剪边界上的点作为参考点，使用了凸多边形的任意一条边，剩下的所有顶点都在这条的同一侧这个原理。

   将裁剪边界分解成一条一条的直线，然后依次对多边形裁剪，每次的被裁剪多边形是上一次裁剪剩下的多边形。

2. 边界和坐标轴平行的矩形裁剪窗口(chapter3/clip2)：

   这个代码可以用于屏幕裁剪，构造一个裁剪窗口和屏幕的显示范围一致，这样就可以裁剪出只在屏幕范围内的图案了。

   本代码相对于 clip1 更加容易理解，clip 函数基本不变，把 clip1 中的 clipLL 改成了 clipParallelBoundary,这个函数判断边界更加容易，不再需要参考点。使用一个矩形裁剪区域作为裁剪窗口，则对于 left 这条边界来说就是 x<left 则被裁剪，top 裁剪条件为 y>top，right 裁剪条件为 x>right，bottom 裁剪条件为 y<bottom。使用被裁剪线段构造一个直线方程(两点式或者一般式或者其他任意方程都行)，然后把 x=left|right 或者 y=top|bottom 代如被裁剪直线中即可求出交点。