Capítulo 7

Recorte de Primitivas 2D

Já vimos que um "pacote gráfico" atua como intermediário entre o aplicativo (e o seu modelo/estrutura de dados interna) e o hardware de visualização. Sua função é aproximar primitivas matemáticas ("ideais"), descritas em termos de vértices num sistema de coordenadas cartesianas, por conjuntos de pixels com a cor ou nível de cinza apropriado. Estes pixels são armazenados num bitmap na memória da CPU, ou num frame buffer (memória de imagem no controlador do dispositivo. Até o momento estudamos, de maneira não extensiva, alguns algoritmos básicos para conversão matricial utilizados por pacotes gráficos. Vamos estudar agora alguns algoritmos para *clipping* [recorte] de primitivas. (O "recorte" é necessário para que a imagem "apareça" dentro do rectângulo de visualização definido para ela.

Existem várias abordagens para o processo de *clippings*. Uma técnica óbvia é recortar a primitiva antes da conversão matricial, calculando analiticamente suas intersecções com o ret8ngulo de recorte/visualização [*clip rectangle*]. Estes pontos de intersecção são então usados para definir os novos vértices para a versão recortada da primitiva. A vantagem, evidente, é que o processo de conversão matricial precisa tratar apenas da versão recortada da primitiva, cuja área pode ser muito menor que a original. Esta é a técnica mais frequentemente utilizada para recortar segmentos de reta, retângulos e polígonos, e os algoritmos que vamos estudar são baseados nesta estratégia.

Outra estratégia seria converter todo o polígono, mas traçar apenas os pixels visíveis no retângulo de visualização [*scissoring*]. lsto pode ser feito checando as coordenadas de cada pixel a ser escrito contra os limites do retângulo. Na prática, existem maneiras de acelerar o processo que evitam o teste de cada pixel individualmente. Se o teste dos limites puder ser feito rapidamente [por hardware especializado, por exemplo], esta abordagem pode ser mais eficiente que a anterior, e tem a vantagem de ser extensível a regiões de recorte arbitrárias.

7.1 Recorte de segmentos de reta

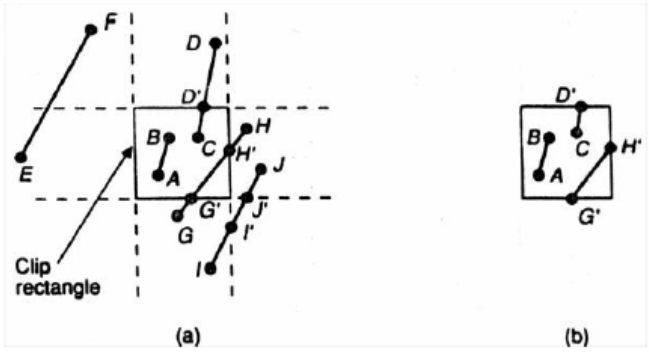

Figura 7.1: Exemplos de recorte de segmentos de reta.

Vamos estudar especificamente o processo de recorte de segmentos de reta contra retângulos. Segmentos que interceptam uma região de recorte retangular, depois de recortados, são sempre transformados num único segmento. Segmentos que estão sobre a fronteira do retângulo de recorte são considerados como dentro dele, e portanto devem ser mostrados (Figura 7.1).

7.1.1 Recorte de Pontos Extremos

Antes de discutir o problema do recorte de segmentos de reta, vamos considerar o problema mais simples de recortar pontos extremos. Se a fronteira do retângulo de recorte tem coordenada x no intervalo xmin e xmax, e coordenada y no intervalo ymin e ymax, então 4 desigualdades precisam ser satisfeitas para que um ponto (x, y) esteja dentro do retângulo de recorte:

$$\begin{aligned} x_{min} \leq x \leq x_{max} \\ y_{min} \leq y \leq y_{max} \end{aligned} \tag{7.1}$$

7.1.2 Algoritmo de Cohen-Sutherland para Recorte de Segmentos de Reta

Para recortar um segmento de reta, precisamos considerar apenas os seus pontos extremos, e não os infinitos pontos no interior. Se ambas as extremidades estão dentro do retângulo de visualização (AB na Figura 7.1), toda a linha está dentro do retângulo e pode ser traçada (neste caso diz-se que a reta foi trivialmente aceita). Se apenas uma das extremidades está dentro (CD na Figura 7.1), a linha intercepta o retângulo de visualização, e o ponto de intersecção precisa ser calculado. Se ambas as extremidades estão fora, a linha pode ou não interceptar o retângulo, e precisamos verificar se as intersecções existem e onde estão.

Uma estratégia de força bruta seria checar a linha contra cada aresta do retângulo de visualização, para localizar um eventual ponto de intersecção. Se existe um, a linha corta o retângulo, e está parcialmente dentro dele. Para cada linha e cada aresta do retângulo, tomamos as duas retas (matematicamente infinitas) que as contém, e calculamos a intersecção entre elas. A seguir, testamos se este ponto é interior - ou seja, se está dentro do retângulo de visualização e da linha. Se este é o caso, existe uma intersecção com o retângulo de visualização. Na Figura 7.1, os pontos de intersecção G' e H' são interiores, mas I' e J' não são.

Nesta abordagem, precisamos resolver duas equações simultâneas usando multiplicação e divisão para cada par $< aresta, linha >$. Entretanto, este é um esquema bastante ineficiente, que envolve quantidade considerável de cálculos e testes, e portanto deve ser descartado.

O algoritmo de Cohen-Sutherland é mais eficiente, e executa testes iniciais na linha para determinar se cálculos de intersecção podem ser evitados. Primeiramente, verifica pares de pontos extremos. Se a linha não pode ser trivialmente aceita, são feitas verificações por regiões. Por exemplo, duas comparações simples em x mostram que ambos os pontos extremos da linha EF na Figura 7.1 têm coordenada x menor que xmin, e portanto estão na região a esquerda do retângulo de visualização (ou seja, fora do semi-plano definido pela aresta esquerda). A consequência é que o segmento EF pode ser trivialmente rejeitado, e não precisa ser recortado ou traçado. Da mesma forma, podemos rejeitar trivialmente linhas com ambos os extremos em regiões a direita de x_{max}, abaixo de ymin e acima de y_{max}. Se o segmento não pode ser trivialmente aceito ou rejeitado, ele é subdividido em dois segmentos por uma aresta de recorte, um dos quais pode ser trivialmente rejeitado. Dessa forma, um segmento é recortado interativamente testando-se aceitação ou rejeição trivial, sendo subdividido se nenhum dos dois testes for bem-sucedido, até que o segmento remanescente esteja totalmente contido no retângulo ou totalmente fora dele.

Para executar os testes de aceitação ou rejeição trivial, as arestas do retângulo de visualização são estendidas de forma a dividir o piano do retângulo de visualização em 9 regiões [Figura 7.2]. A cada região é atribuído um código de 4 bits, determinado pela posição da região com relação aos semi-planos externos às arestas do retângulo de recorte. Uma maneira eficiente de calcular os códigos resulta da observação de que o bit 1 é igual ao bit de sinal de $(y_{max} - y)$; o 2 é o bit de sinal de $(y - y_{min})$; o 3 é o bit de sinal de $(x_{max} - x)$, e o 4 e o de $(x - x_{min})$.

- 1° bit: semiplano acima da aresta superior $y > y_{max}$
- 2° bit: semiplano abaixo da aresta inferior $y < y_{min}$
- 3° bit: semiplano a direita da aresta direita $x > x_{max}$
- 4° bit: semiplano a esquerda da aresta esquerda $x < x_{min}$

A cada extremidade do segmento de reta é atribuído o código da região a qual ela pertence. Esses códigos são usados para determinar se o segmento está completamente dentro do retângulo de visualização ou em um semiplano externo a uma aresta. Se os 4 bits dos códigos das extremidades são iguais a zero, então a linha está completamente dentro do retângulo. Se ambas as extremidades estiverem no semi plano externo a uma aresta em particular, como para EF na Figura 7.1, os códigos de ambas as extremidades tem o bit correspondente àquela aresta igual a 1. Para EF, os códigos são 0001 e 1201, respectivamente,

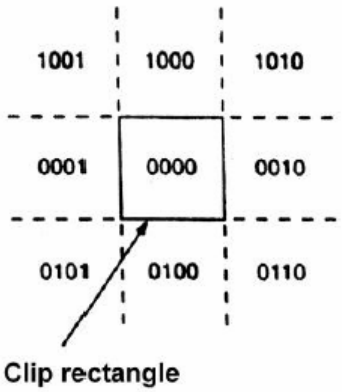

Figura 7.2: Códigos das regiões.

o que mostra com o 4° bit que o segmento está no semiplano externo à aresta esquerda. Portanto, se um and lógico dos códigos das extremidades for diferente de zero, o segmento pode ser trivialmente rejeitado.

Se a linha não puder ser trivialmente aceita ou rejeitada, devemos subdividi-la em dois segmentos de forma que um deles possa ser rejeitado. A subdivisão é feita usando uma aresta que seja interceptada pela linha para subdividi-la em dois segmentos: a seção que fica no semiplano externo à aresta é descartada. 0 teste de quais arestas interceptam a linha segue a mesma ordem dos bits nos códigos: de cima-para-baixo, da direita-para-a-esquerda. Uma propriedade chave dos códigos é que os bits iguais a um correspondem a arestas interceptadas: se uma extremidade está no semiplano externo a uma aresta, e o segmento falha nos testes de rejeição trivial, então a outra extremidade deve estar no semiplano interno àquela aresta, que é portanto interceptada pelo segmento. Assim, o algoritmo sempre escolhe um ponto que esteja fora, e usa os bits do código iguais a um) para determinar uma aresta de recorte. A aresta escolhida é a primeira encontrada na ordem pré-definida, ou seja, a correspondente ao primeiro bit igual a 1 no código.

0 algoritmo funciona da seguinte maneira: Calcula primeiramente os códigos de ambas as extremidades, e checa aceitação ou rejeição trivial. Se nenhum dos testes for bem-sucedido, tomamos uma extremidade externa (pelo menos uma delas é externa), e testamos seu código para obter uma aresta que seja interceptada e determinar o ponto de intersecção. Podemos então descartar o segmento que vai do extremo externo ao ponto de intersecção, substituindo o ponto externo pela intersecção. 0 novo código para o segmento correspondente é calculado, preparando os dados para a próxima iteração.

Considere por exemplo o segmento AD na Figura 7.4. 0 código do ponto A é 0000, e o do ponto D é 1001 . Q segmento não pode ser trivialmente aceito ou rejeitado. 0 algoritmo escolhe D como o ponto externo, e o seu código mostra que a linha cruza as arestas superior e esquerda. De acordo com a ordem de teste, usamos primeiro a aresta superior para recortar AD, gerando AB. O código de B é 0000 e na próxima interação o segmento será trivialmente aceito e traçado.

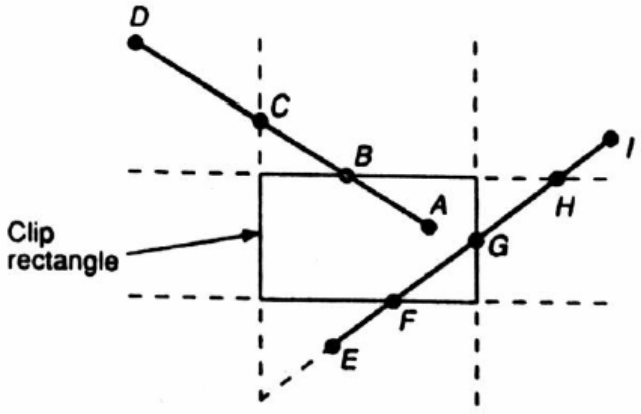

Figura 7.3: Funcionamento do algoritmo de Cohen-Sutherland para recorte de segmentos.

A linha EI requer mais iterações. O primeiro extremo, E, tem código 0100 e é escolhido pelo algoritmo

como ponto externo. O código é testado para obter a primeira aresta que a linha cruza, no caso a aresta inferior, e EI é recortado para FI. Na segunda iteração, FI não pode ser trivialmente aceito ou rejeitado. O código da primeira extremidade, F, é 0000, de forma que o algoritmo escolhe o ponto externo I, cujo código é 1010. A primeira aresta interceptada é a superior, resultando no segmento FH. O código de H é 0010, e a terceira iteração resulta num corte pela aresta direita. O segmento resultante é trivialmente aceito e traçado na quarta iteração. O Algoritmo 7.1 e 7.2 ilustram este procedimento.

Este algoritmo não á o mais eficiente - como os testes e recorte são executados numa ordem fixa, eventualmente serão feitos recortes desnecessários. Por exemplo, quando a intersecção com uma aresta do retângulo é uma "intersecção externa", que não está na fronteira do retângulo de visualização (ponto H na linha EI, Figura 7.4). Existem algoritmos mais eficientes (v. Foley, 3.12), mas este ainda é o mais utilizado, por ser simples e bastante conhecido.

```
typedef enum {LEFT = 0, RIGHT = 1, BOTTOM = 2, TOP = 3} aresta;
typedef enum {TRUE, FALSE} boolean;
typedef boolean outcode;

/* Retorna TRUE se todos os outcodes em codigo sao FALSE, e
 * FALSE c.c. */
boolean Vazio(outcode codigo[]);

/* Retorna TRUE se a interseccao logica entre codigo1 e codigo
 * 2 e vazia, e FALSE c.c. */
boolean InterseccaoVazia(outcode codigo0[], outcode codigo1 []);

/* Retorna TRUE se codigo0 e codigo 1 sao iguais, e FALSE c.c. */
boolean Igual(outcode codigo0[], outcode codigo1[]);

/* Calcula outcode para ponto (x,y) */
void CalculaOutCode(float x, float y, outcode codigo[]) {
   int i;

   for (i - LEFT; i < = TOP; i++)
      codigo[i] = FALSE;
   /* Fim for */
   if (y > YMAX)
      codigo[TOP] = TRUE;
   else if (y < YMIN)
      codigo[BOTTOM] = TRUE;
   /* Fim se */
   if (x > xmax)
      codigo[RIGHT] = TRUE;
   else if (x < xmax)
      codigo[BOTTOM] = TRUE;
   /* Fim Se */
}/* End CalculaOutCode */
```

Algoritmo 7.1: Calculando os códigos do algoritmo de Cohen-Sutherland.

7.2 Recorte de Circunferências

Para recortar uma circunferência contra um retângulo, podemos primeiro executar um teste de aceitação/rejeição trivial interceptando a extensão da circunferência (um quadrado do tamanho do diâmetro da circunferência) com o retângulo de visualização usando o algoritmo para recorte de polígonos que será visto a seguir. Se a circunferência intercepta o retângulo, ela é subdivida em quadrantes, e testes de aceitação/rejeição trivial são feitos para cada um. Estes testes podem levar a testes por octantes. É possível então calcular a intersecção da circunferência e da aresta analiticamente, resolvendo suas equações simultaneamente, e a seguir fazer a conversão matricial dos arcos resultantes. Se a conversão matricial é rápida, ou se a circunferência não é muito grande, seria provavelmente mais eficiente usar a técnica de scissoring, testando cada pixel na circunferência contra os limites do retângulo antes de traça-lo.

Um algoritmo para recorte de polígonos precisa tratar de muitos casos distintos, como mostra a Figura 7.5. O caso (a) é particularmente interessante porque um polígono côncavo é recortado em dois polígonos

```
/* algoritmo de recorte de Cohen-Sutherland para linha P0(x0,y0) a
 * P1(x1,y1), e retangulo de recorte com diagonal de (xmin,ymin) a
 * (xmax,ymax). */
void RecorteLinhaCohenSutherland(float x0,float y0,float x1,float y1,int valor){
   boolean aceito, pronto;
   outcode outcode0[4], outcode1 [4], *outcodeOut; /* outcodes para P0, P1 e
                                                      quaisquer outros pontos que
                                                      estao fora do retangulo de
                                                      recorte */
   float x, y;
   aceito = FALSE; pronto = FALSE;

   CalculaOutCode(x0,y0,outcode0); CalcuiaOutCode(xl ,y1 ,outcode1);
   do {
      if (vazio(outcode0) && vazio(outcode1)){
         /* aceitacao trivial e sai */
         aceito = TRUE;
         pronto = TRUE;
      }else if (interseccao_vazia(outcode0,outcode1))
         /* rejeicao trivial e sai */
         pronto = TRUE;
      else {
         /* ambos os testes falharam, entao calcula o segmento de
         reta a recortar: a partir de um ponto externo a uma
         interseccao com a aresta de recorte */
         /* pelo menos um ponto esta fora do retangulo de recorte;
         seleciona-o */

         outcodeOut= vazio(outcode0) ? outcode1 : outcode0;
         /* acha agora o ponto de interseccao, usando as formulas:
            y = y0 + inclinacao * (x-x0), x = x0 + ( 1 /inclinacao) * (y-y0) */
         if (outcodeOut(TOP)) {
            /* divide a linha no topo do retangulo de recorte */
            x= x0 + (x1-x0) * (ymax-y0)/(y1-y0); y= ymax;
         }else if (outcodeOut[BOTTOM]) {
            /* divide a linha na base do retangulo de recorte */
            x = x0 + (x1-x0) + (ymin-y0)/(y1-y0); y = ymin;
         }else if (outcodeOut[RIGHT]) {
            /* divide a linha na aresta direita do retangulo de recorte */
            y = y0 + (y1 -y0) + (xmax-x0)/(x1-x01; x = xmax;
         }else if (outcodeOut[LEFT]) {
            /* divide a linha na aresta esquerda do retangulo de recorte */
            y = y0 + (y 1 -y0) + (xmin-x0)/(x1-x0); x = xmin;
         } /* Fim se */
         /* agora move o ponto externo para o ponto de interseccao, para
            recortar e preparar para o proximo passo */
         if (igual(outcodeOut,outcode0)) {
            x0 = x; y0 = y; CalculaOutCode(x0,y0,outcode0);
         }else{
            x1 = x; y1 = y; CalculaOutCode(x1,y1,outcode1);
         }
      }/* fim do else da subdivisao */
   } while (pronto);

   if (aceito)
      MeioPontoLinhaReal(x0,y0,x1,y1),valor); /* versao do algoritmo para coordenadas reais */
} /* fim do algoritmo de recorte e tracado */
```

Algoritmo 7.2: Desenhando a linha com o algoritmo de Cohen-Sutherland. Algumas das funções usadas estão ilustradas no Algoritmo 7.2.

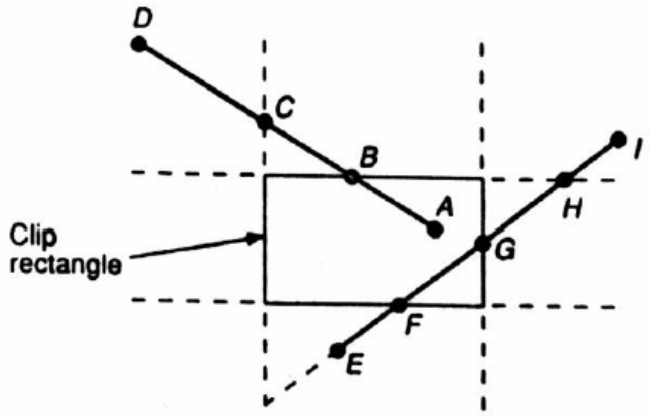

Figura 7.4: Algoritmo de Sutherland-Hodgman para Recorte de Polígonos.

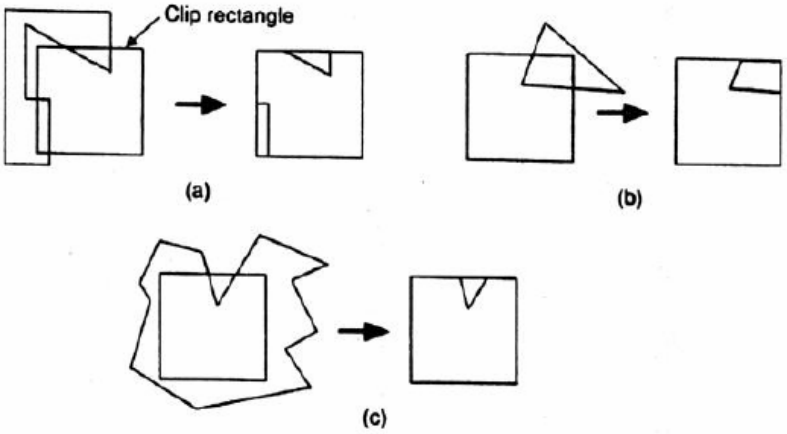

Figura 7.5: Exemptos de recorte de polígonos. (a) Múltiplos componentes. (b) Caso convexo (c) Caso côncavo com muitas arestas exteriores.

separados. O problema é complexo porque cada aresta do polígono precisa ser testada contra cada aresta do ret9ngulo de visualização; novas arestas precisam ser acrescentadas, arestas existentes precisam ser descartadas, mantidas ou subdivididas. Vários polígonos podem resultar do recorte de um único, e é necessário uma abordagem organizada para tratar todos os caso possíveis.

O algoritmo para recorte de polígonos de Sutherland-Hodgman usa a estratégia "dividir para conquistar"; resolve uma série de problemas simples e idênticos cujas soluções, quando combinadas, resolvem o problema todo. O problema básico consiste em recortar o polígono contra uma única aresta de recorte infinita. Quatro arestas, cada qual definindo um lado do retângulo, recortam o polígono sucessivamente contra o retângulo de visualização.

Note a diferença entre esta estratégia para um polígono e o algoritmo de Cohen-Sutherland para recorte de linhas: O algoritmo para polígonos recorta quatro arestas sucessivamente, enquanto que o algoritmo para linhas testa o código para verificar qual aresta é interceptada, e recorta apenas quando necessário. O algoritmo de Sutherland-Hodgman é na verdade mais geral: um polígono (côncavo ou convexo) pode ser recortado contra qualquer polígono de recorte convexo. Em 3D, polígonos podem ser recortados contra volumes poliedrais convexos definidos por planos. O algoritmo aceita como entrada uma série de vértices v1, v2, ..., vn que, em 2D, definem as arestas de um polígono. A seguir, recorta o polígono contra uma única aresta de recorte infinita, e retorna outra série de vértices que definem o polígono recortado. Num segundo passo, o polígono parcialmente recortado é recortado novamente contra a segunda aresta, e assim por diante.

O algoritmo movimenta-se em torno do polígono, indo do vértice vn ao vértice v1, e de volta a vn, examinando a cada passo a relação entre um vértice e a aresta de recorte. Em cada passo, zero, um ou dois vértices são adicionados à lista de saída dos vértices que definem o polígono recortado, dependendo

da posição do vértice sendo analisado em relação à aresta de recorte.

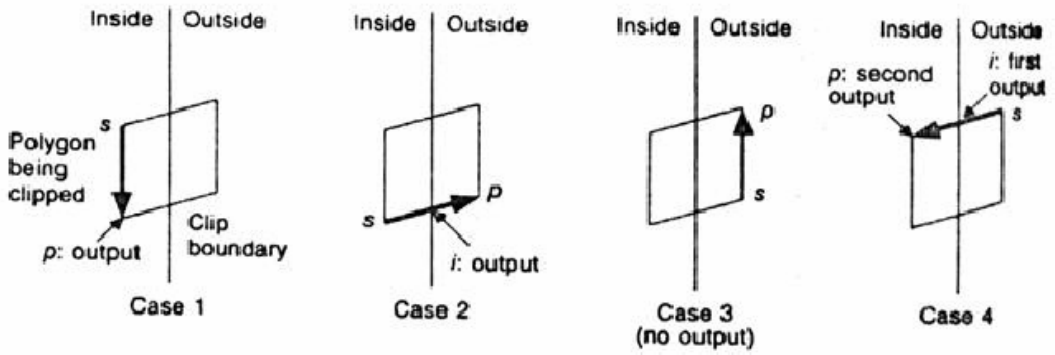

Figura 7.6: Quatro situações possíveis no recorte de polígonos.

São quatro situações possíveis (Figura 7.6). Consideremos uma aresta do polígono que vai do vértice s ao vértice p. Suponha que o vértice s foi analisado na iteração anterior, e a interação corrente este analisando p. No caso 1 , a aresta do polígono está completamente dentro da fronteira do retângulo de visualização, e o vértice p é adicionado à lista de saída. No caso 2, o ponto i, que define a intersecção entre a aresta do polígono e a aresta do retângulo, é colocado como um vértice na lista de saída. No caso 3, ambos os vértices estão fora do retângulo, e nenhum é colocado na lista de saída. No caso 4, ambos os pontos de intersecção, i e p, são colocados na lista de saída.

O procedimento, mostrado no Algoritmo 7.3, aceita um vetor de vértices, inVertexArray, e cria outro vetor outVertexArray. Para manter o código simples, não incluímos verificação dos erros nos limites dos vetores, e usamos o procedimento Output() para incluir um vértice em outVertexArray. O procedimento Intersect() calcula a intersecção de uma aresta s-¿p do polígono com a aresta clipBoundary, definida por dois vértices da fronteira do polígono de recorte. A função Inside() retorna true se o vértice está no interior da fronteira de recorte. Assume-se que o vértice está no interior da fronteira se estiver "a esquerda da aresta de recorte, quando olha-se do primeiro para o segundo vértice da aresta". Esta orientação corresponde a uma enumeração das arestas no sentido anti-horário. Para calcular se um ponto está fora da fronteira de recorte, podemos testar o sinal do produto vetorial do vetor normal à fronteira de recorte e da aresta do polígono.

0 algoritmo é bastante geral, e pode ser estruturado para chamar a si próprio recursivamente de uma forma que o torna bastante eficiente para implementação em hardware.

```
/* Algoritmo de Sutherland-Hodgman para recorte do polígonos */
#define MAX 20

typedef struct {float x,y;} ponto;
typedef ponto vertice;
typedef vertice aresta[2];
typedef vertice VetorVertices[MAX];
typedef enum {TRUE = 1, FALSE = 0} boolean;

/* Acrescenta um novo vertice a vetorSaida, e atualiza tamSaida */
void saida(vertice novoVertice, int *tamSaida, VetorVertices vetorSaida);

/* Checa se o vertice esta dentro ou fora da aresta de recorte */
boolean dentro(vertice testeVertice, aresta arestaRecorte);


/* recorta a aresta (prim,seg) do poligono contra arestaRecorte,
 * retorna o novo ponto */
vertice intercepta(vertice prim, vertice seg, aresta arestaRecorte);

void RecortePoligonosSutherlandHodgman(
      VetorVertices vetorEntrada, /* vetor de vertices de entrada */
      VetorVertices vetorSaida, /* vetor de vertices de saida */
      int tamEntrada, /* numero de entradas em vetorEntrada */
      int tamSaida, /* numero de vertices em vetorSaida */
      aresta arestaRecorte /* aresta do poligono de recorte */){
   vertice s, p;  /* pontos inicial e final da aresta de recorte corrente */
   vertice i;     /* ponto de interseccao com uma aresta de recorte */
   int j;         /* contador para o loop de vertices */
   *tamSaida = 0;

   s = vetorEntrada[tamEntrada]; / * comeca com o ultimo vertice em vetorEntrada */
   for (j = 1 ;j < = tamEntrada;j + + ) {
      p= vetorEntrada[j]; /* agora s e p correspondem aos vertices na Fig. */
      if (dentro(s,arestaRecorte)) {
         /* casos 1 e 4 */
         if (dentro(s,arestaRecorte))
            saida(p,tamSaida,vetorSaida);
         else {
            i = intercepta(s,p,arestaRecorte);
            saida(i,tamSaida,vetorSaida);
            saida(p,tamSaida,vetorSaida);
         }/* Fim Se */
      }else{
         /* casos 2 e 3 */
         if (dentro(s,arestaRecorte)) {
            i = intercepta(s,p,arestaRecorte);
            saida(i,tamSaida,vetorSaida);
         }/* Fim Se */
      }/*Fim Se */
      s = p;
   }/* fim do for */
}/* fim do algoritmo de recorte */
```

Algoritmo 7.3: Algoritmo de Cohen-Sutherland.