

Técnicas de Detecção de Colisão para Jogos



por [Leandro Silva](#)

Técnicas de Detecção de Colisão para Jogos

por: Gustavo Russo Zanardo

Esse artigo visa mostrar as principais técnicas para detecção de colisão em jogos 2D. Nele são mostradas as vantagens e desvantagens de cada uma e em qual situação devemos usar cada uma.

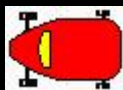
Serão mostradas aqui 4 tipos de testes de colisão, que são:

- BoundingBox
- PixelPerfect
- Colisão entre círculos
- Colisão entre polígonos

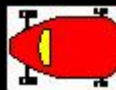
Colisão por Bounding-Box

Bounding-Box – É um dos algoritmos mais rápidos de detecção de colisão que existe, entretanto em alguns casos (como em figuras arredondadas) ele pode ser muito impreciso e o efeito acaba não sendo o desejado.

Basicamente a idéia do Bounding Box é criar um “quadrado imaginário” em torno das figuras que se deseja testar a colisão, e assim tratar as colisões desses quadrados imaginários:



->

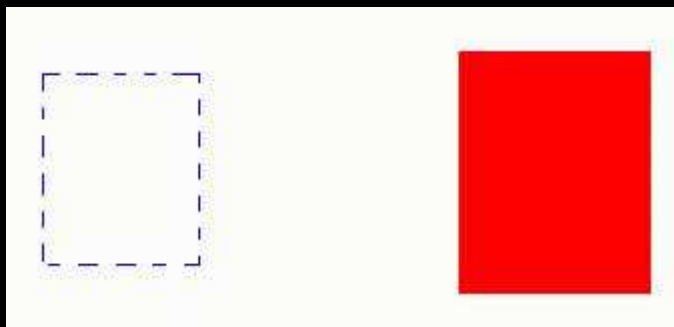


Por exemplo, se você quiser checar a colisão de um ponto com um quadrado, você pode fazer assim:

```
/*-----*/  
//( código em C )  
  
struct QUADRADO{  
    int x;  
    int y;  
    int largura;  
    int altura  
};  
  
struct PONTO{  
    int x;  
    int y;  
};  
  
int checaColisao( struct PONTO *ponto, struct QUADRADO *quadrado ){  
    if( ponto->x >= quadrado->x && ponto->x <= (quadrado->x + quadrado->largura) ){  
        if( ponto->y >= quadrado->y && ponto->y <= (quadrado->y + quadrado->altura) ){  
            //COLISÃO  
            return 1;  
        }  
    }  
    return 0; //não colisão  
}  
/*-----*/
```

Para testar a colisão entre dois quadrados, o mais fácil é testar a não-colisão do que a colisão, ou seja, você coloca **ifs** para testar a não-colisão, e se nenhuma das condições for verdadeira, então está ocorrendo a colisão.

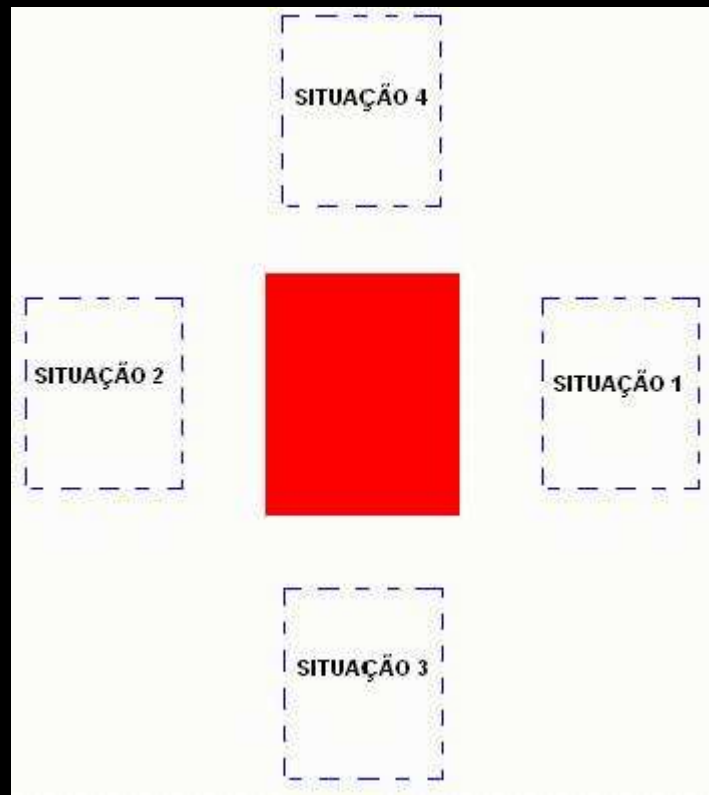
Por exemplo, testar a colisão entre esses dois quadrados (azul e vermelho):



Repare que se o lado esquerdo do quadrado azul estiver com a coordenada x maior que o lado direito do quadrado vermelho, eles nunca estarão colidindo, independente das coordenadas y que os quadrados estiverem (Situação 1). Da mesma forma, se o lado direito do quadrado azul estiver com a coordenada x menor que o lado esquerdo do quadrado vermelho, eles também nunca estarão se colidindo, independente das coordenadas y (Situação 2). Se o lado de cima do quadrado azul estiver com a coordenada y maior que o lado de baixo do quadrado vermelho, eles também nunca estarão colidindo (independente das coordenadas x dos dois quadrados). E finalizando, se o lado de baixo do quadrado azul estiver com a coordenada y menor que o lado de cima do quadrado vermelho eles

também nunca estarão colidindo.

Assim, se uma dessas quatro condições for verdadeira, o quadrado não estará colidindo, do contrário os dois quadrados estarão colidindo.



Agora para testarmos a colisão dos dois quadrados podemos fazer o seguinte.

Considerando esta estrutura de um quadrado:

```
/*-----*/  
struct QUADRADO{  
    int x;  
    int y;  
    int largura;  
    int altura  
};  
/*-----*/
```

Primeiro vamos encontrar os lados dos quadrados.

Sendo quadrado1 o azul e quadrado2 o vermelho, temos os lados do quadrado azul dessa forma::

```
/*-----*/  
esquerdaAzul = quadrado1.x;  
cimaAzul = quadrado1.y  
direitaAzul = quadrado1.x + quadrado1.largura;  
baixoAzul = quadrado1.y + quadrado1.altura;  
/*-----*/
```

da mesma forma encontramos os lados do quadrado vermelho:

```
/*-----*/
esquerdaVermelho = quadrado2.x;
cimaVermelho = quadrado2.y
direitaVermelho = quadrado2.x + quadrado2.largura;
baixoVermelho = quadrado2.y + quadrado2.altura;
/*-----*/
```



Encontrado os lados dos quadrados: azul e vermelho, basta verificar as quatro situações de não-colisão:

```
/*-----*/
If( esquerdaAzul > direitaVermelho ){
    /*não colisão*/
}
If( direitaAzul < esquerdaVermelho ){
    /*não colisão*/
}
If( cimaAzul > baixoVermelho ){
    /*não colisão*/
}
If( baixoAzul < cimaVermelho ){
    /*não colisão*/
}
/*-----*/
```

Se nenhuma dessas condições for verdadeira, então os dois quadrados estão colidindo.

Dessa forma, agora podemos montar uma função que receba dois quadrados e verifique se eles estão colidindo ou não.

O código ficaria assim:

```
/*-----*/
struct QUADRADO{
    int x;
    int y;
    int largura;
    int altura;
};
```

```

int checaColisao( struct QUADRADO *quadrado1, struct QUADRADO *quadrado2 ){
    int esquerda1, direita1, cima1, baixo1;
    int esquerda2, direita2, cima2, baixo2;

    esquerda1 = quadrado1->x;
    direita1 = quadrado1->x + quadrado1->largura;
    cima1 = quadrado1->y;
    baixo1 = quadrado1->y + quadrado1->altura;

    esquerda2 = quadrado2->x;
    direita2 = quadrado2->x + quadrado2->largura;
    cima2 = quadrado2->y;
    baixo2 = quadrado2->y + quadrado2->altura;

    if( esquerda1 > direita2 )
        return 0;          //retorna 0 (não colisão)
    if( direita1 < esquerda2 )
        return 0;          // retorna 0 (não colisão)
    if( cima1 > baixo2 )
        return 0;          // retorna 0 (não colisão)
    if( baixo1 < cima2 )
        return 0;          // retorna 0 (não colisão)

    /*do contrário retorna 1 (COLISÃO)*/
    return 1;
}
/*-----*/

```

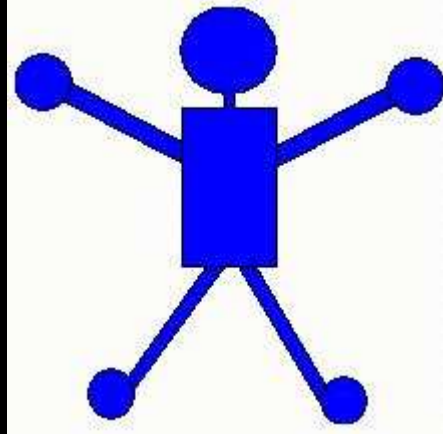
Esta função que recebe dois quadrados, retorna 0 se os quadrados não estiverem colidindo e retorna 1 se eles estiverem colidindo.

O código fonte com um exemplo de colisão por BoundingBox pode ser encontrado aqui:

<http://filexoom.com/uploaded/2007/9/12/85979/BoundingBox.zip>

O Bounding Box é uma técnica simples com uma performance muito boa devido ao baixo número de contas e de verificações que o algoritmo executa. Entretanto, como ela testa colisões entre quadrados, em figuras muito curvas e disformes ela pode não apresentar um resultado satisfatório.

Numa figura como esta por exemplo o resultado não seria satisfatório.



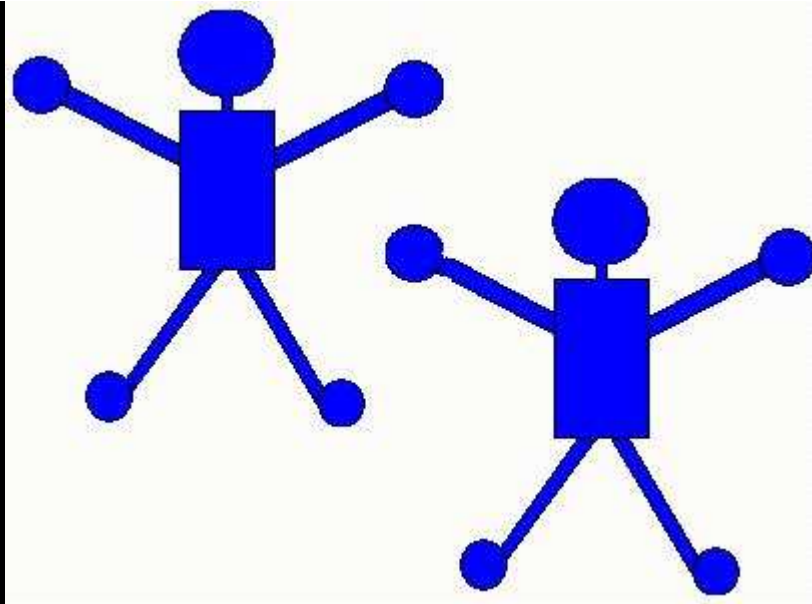
Bem, agora para realizarmos o Bounding Box, vamos imaginar um quadrado em volta dessa figura assim:



Note que o tratamento de colisão por Bounding Box será feito pelo quadrado em volta da figura, ou seja, mesmo as áreas que estão em amarelo (que não deveriam ser consideradas na colisão) serão tratadas como áreas de colisão.

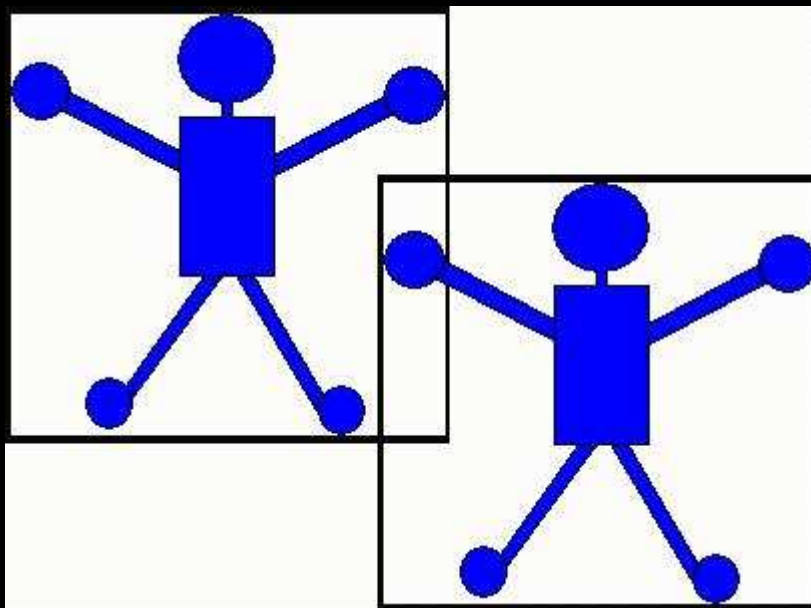
Então em colisões entre duas imagens dessas, poderia haver situações em que mesmo elas não estarem se colidindo, o algoritmo detectaria que elas estariam em colisão.

Exemplo:



Nessa situação as figuras não estão se colidindo, mas pelo algoritmo de Bounding Box elas estariam se colidindo, pois os "quadrados imaginários" em volta das figuras estariam se colidindo.

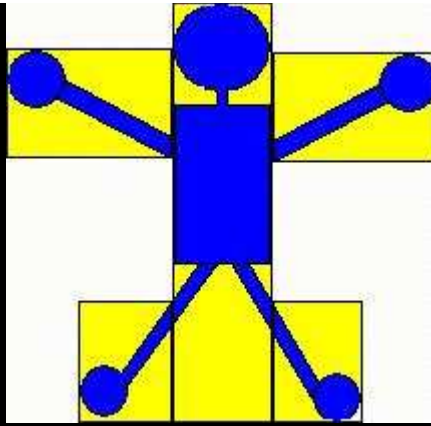
Veja na figura abaixo:



O Bounding Box apesar de em certas ocasiões não ser muito preciso, como no caso anterior, ele deve ser usado sempre que possível nos casos de figuras mais simples, pois tem uma performance muito melhor do que a maioria das técnicas de colisão.

Uma técnica muito usada também é, em vez de criar um retângulo em volta de toda a figura, são criados vários retângulos imaginários menores, aí é o teste é feito com todos esses retângulos.

Veja um exemplo

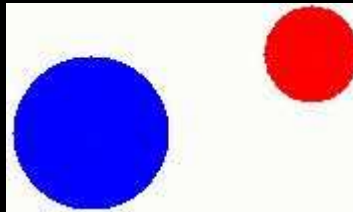


As áreas em amarelo na figura são as áreas dos retângulos imaginários em volta da figura. Na hora de testar a colisão dessa figura com outra, são testados a colisão da outra figura com todos esses retângulos. Isso, apesar de consumir mais processamento, tem um melhor efeito e torna o teste de colisão mais preciso.

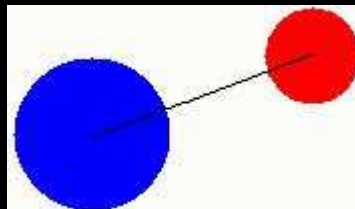
Colisão entre dois círculos

Bem, como vimos anteriormente, a técnica de colisão por Bounding Box pode não ser muito eficiente para testar colisões entre figuras arredondadas

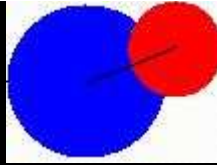
No tratamento de colisão entre dois círculos, por exemplo, o Bounding Box não seria muito eficiente.



Nesse caso, uma boa forma de tratamento de colisão é checar a distância entre os centros dos dois círculos. Se por acaso essa distância for maior que a soma dos raios desses círculos, então esses círculos não estão se colidindo.



Do contrário, se a distância entre os centros dos círculos for menor que a soma dos raios significa que eles estão se colidindo.

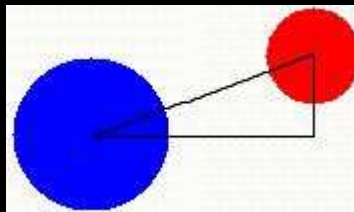


Bem, então para implementar esse algoritmo a primeira coisa que devemos fazer é calcular a distância entre os centros dos dois círculos. Isso pode ser feito através da fórmula de Pitágoras que diz que a soma dos quadrados dos catetos é igual ao quadrado da hipotenusa:

h - hipotenusa
 c1 - cateto1
 c2 - cateto2

$$h^2 = c1^2 + c2^2$$

A distância entre esses dois pontos (os centros das circunferências) é a hipotenusa



Então, passando o quadrado da hipotenusa p/ o outro lado como raiz, ficaria assim:

$$h = \sqrt{(c1^2) + (c2^2)}$$

Os cateto1 pode ser a diferença entre as coordenadas x do centro dos círculos.
 O cateto2 pode ser a diferença entre as coordenadas y do centro dos círculos.

Então dado:

```
/*-----*/
struct CIRCULO{
    int xCentro;
    int yCentro;
    int raio;
};
/*-----*/
```

Ficaria assim:

```
/*-----*/
cateto1 = circulo2.xCentro - circulo1.xCentro;
cateto2 = circulo2.yCentro - circulo1.yCentro;
/*-----*/
```

Para a hipotenusa é só aplicar a fórmula $h = \sqrt{(c1^2) + (c2^2)}$

```

/*-----*/
hipotenusa = sqrt( cateto1*cateto1 + cateto2*cateto2 );
/*-----*/

```

Obs: em C existe uma biblioteca chamada "math.h" que tem uma função que tira a raiz quadrada chamada "sqrt".

Agora basta checar se essa distância (hipotenusa) é menor que a soma dos raios dos círculos:

```

/*-----*/
int distancia = hipotenusa;
if( distancia < (circulo1.raio + circulo2.raio) )
    /* Colisão */
/*-----*/

```

Agora podemos montar uma função que dado dois círculos (struct CIRCULO) diga se eles estão ou não se colidindo:

```

/*-----*/
int checaColisao( struct CIRCULO *circulo1, struct CIRCULO *circulo2 ){
    int cateto1, cateto2, distancia;

    cateto1 = circulo2.xCentro - circulo1.xCentro;
    cateto2 = circulo2.yCentro - circulo1.yCentro;

    distancia = sqrt( cateto1 * cateto1 + cateto2 * cateto2 );

    if( distancia < (circulo1.raio + circulo2.raio) ){
        return 1; //colisão
    }

    /*caso contrário*/
    return 0;
}
/*-----*/

```

Essa função que recebe dois círculos como parâmetro retorna 1 caso os dois círculos estejam se colidindo e retorna 0 caso contrário.

O código fonte com um exemplo de colisão entre círculos pode ser encontrado aqui:

<http://filexoom.com/uploaded/2007/9/12/85979/ColisaoCirculos.zip>

Esse tipo de colisão é uma das melhores maneiras de testar a colisão entre dois círculos pois é muito preciso sendo igual a técnica de PixelPerfect nesse caso, mas com muito mais performance que o PixelPerfect, além de ter uma implementação muito mais simples. Entretanto, ele só terá um efeito bom caso as figuras sejam círculos.

Colisão por PixelPerfect

PixelPerfect é uma técnica de colisão que consiste em analisar cada pixel de cada imagem para verificar se a parte do desenho da imagem se colide com a outra. É uma técnica que em todos os casos a colisão é detectada perfeitamente sem erros. Apesar disso é uma das técnicas com menos performance e por isso só deve ser utilizada quanto realmente houver necessidade de um teste de colisão muito preciso. Ela geralmente é usada em

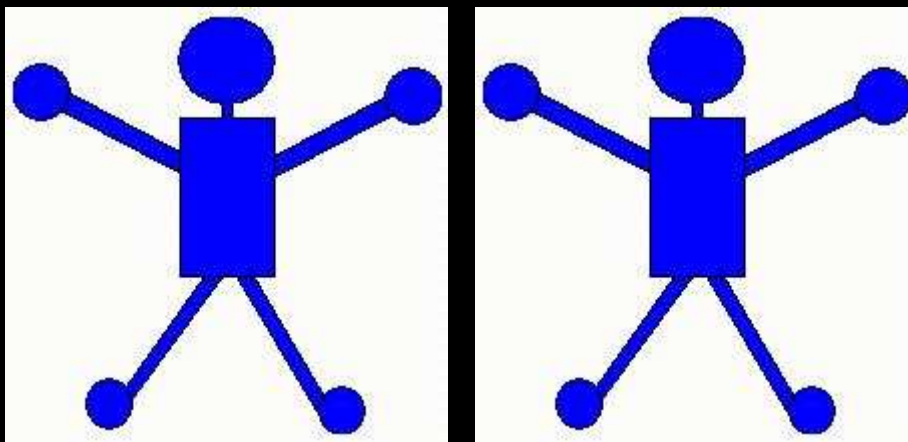
testes com figuras de formas mais complexas.

Vou descrever aqui como implementar um teste de colisão por PixelPerfect.no *Allegro*.

Para colisões por PixelPerfect existe uma biblioteca chamada PMask que faz todo o trabalho para você. Vou mostrar aqui a implementação com a biblioteca PMask e também vou mostrar como fazer essa implementação de PixelPerfect sem a utilização de nenhuma biblioteca, o que requer um pouco mais de trabalho (nada muito complicado também).

1 – Sem a utilização de bibliotecas de detecção de colisão

Vamos supor que temos as seguintes imagens e queremos checar a colisão entre elas:



Bem, temos então essas duas imagens e queremos testar a colisão entre elas. Como queremos precisão na checagem de colisão, vamos utilizar a técnica de PixelPerfect.

Bem a primeira coisa a fazer é realizar a checagem de colisão por BoundingBox nessas imagens. Isso é feito para que só sejam verificados os pixels das imagens caso o BoundingBox esteja checando a colisão, ou seja, caso os "quadrados imaginários" em volta da imagem estejam colidindo. Caso contrário, já sabemos que elas não colidem então não precisamos gastar processamento checando cada pixel.

Então a primeira coisa é realizar o teste BoundingBox nessas figuras.



Vamos primeiramente criar uma estrutura para representar cada figura.

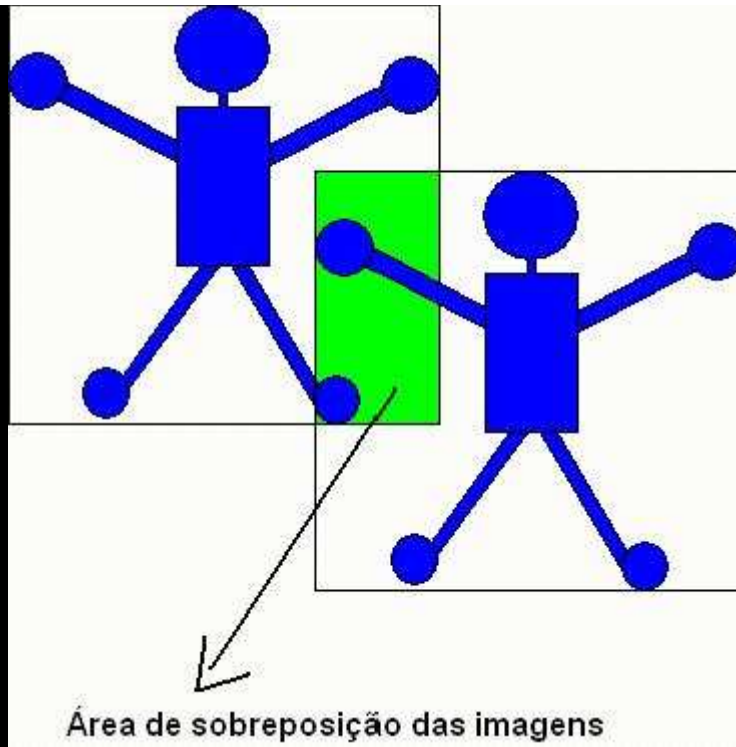
```
/*-----*/  
typedef struct FIGURA{  
    int x;  
    int y;  
    int largura;  
    int altura;  
    BITMAP *imagem;  
}Figura;  
/*-----*/
```

Feito isso, vamos agora criar uma função de checagem por BoundingBox, que receba as duas figuras e retorne 1 caso elas estejam colidindo e 0 caso contrário.

```
/*-----*/  
//checa colisão por BoundingBox  
int checaBoundingBox( Figura *figura1, Figura *figura2 ){  
    int esquerda1, direita1, cima1, baixo1;  
    int esquerda2, direita2, cima2, baixo2;  
  
    esquerda1 = figura1.x;  
    direita1 = figura1.x + figura1.largura;  
    cima1 = figura1.y;  
    baixo1 = figura1.y + figura1.altura;  
  
    esquerda2 = figura2.x;  
    direita2 = figura2.x + figura2.largura;  
    cima2 = figura2.y;  
    baixo2 = figura2.y + figura2.altura;  
  
    if( esquerda1 > direita2 )  
        return 0;  
    if( direita1 < esquerda2 )  
        return 0;  
    if( cima1 > baixo2 )  
        return 0;  
    if( baixo1 < cima2 )  
        return 0;  
  
    return 1;  
}  
/*-----*/
```

Agora já podemos checar se as figuras estão se colidindo por BoundingBox.

Caso a função "checaBoundingBox" retorne 1 (colisão) então devemos obter o "quadrado comum das imagens", ou seja, a área em que as coordenadas são comuns para as duas imagens (a área de sobreposição das imagens).



Vamos então criar uma estrutura para representar esse quadrado da área de sobreposição das figuras:

```

/*-----*/
typedef struct QUADRADO{
int esquerda; //left
int direita; //right
int cima; //top
int baixo; //bottom
}Quadrado;
/*-----*/

```

Os campos representam os lados do quadrado e são:

esquerda - coordenada x da figura do quadrado

direita - coordenada x da figura do quadrado mais a largura do quadrado

cima - coordenada y da figura do quadrado

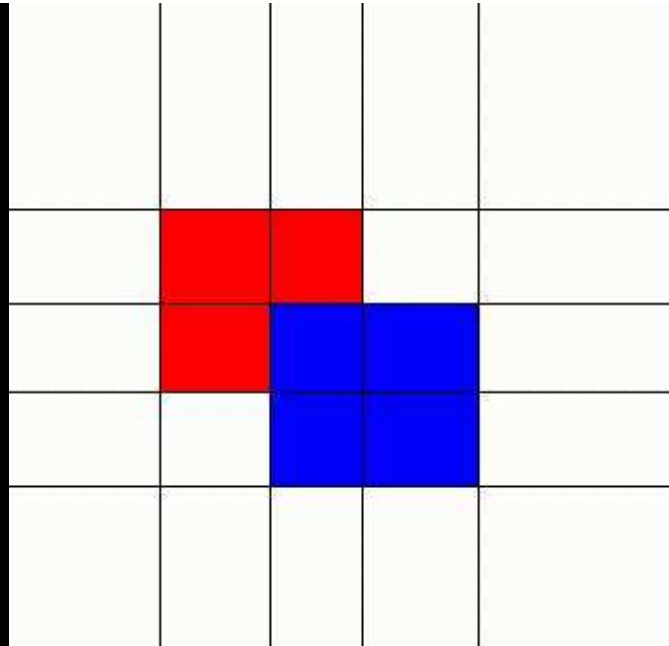
baixo - coordenada y da figura do quadrado mais a altura do quadrado

Para obtermos esse quadrado de sobreposição, vamos fazer o seguinte:

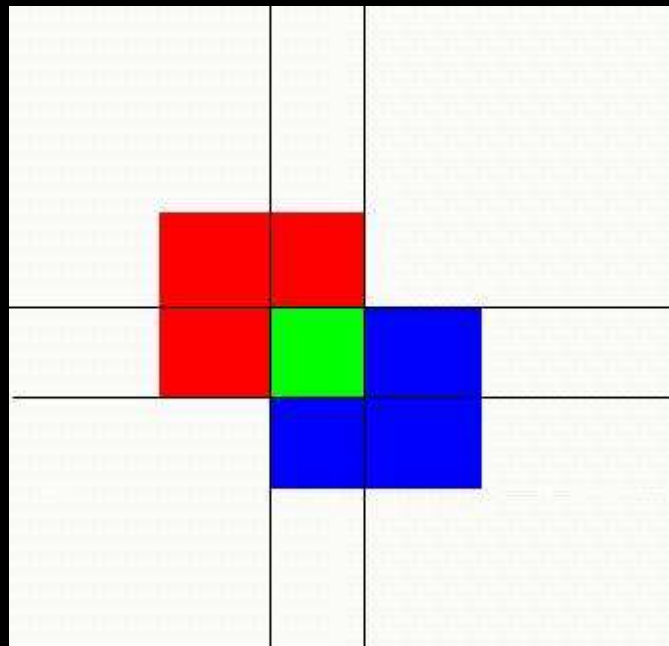
Para obter os lados verticais do quadrado (esquerda e direita), temos que pegar a segunda e terceira linha que aparece na tela, ou seja, o segundo e terceiro lado dos quadrados (considerando os dois quadrados).

Se você não entendeu faça o seguinte, imagine dois quadrados se colidindo e em cada lado de cada quadrado está desenhado uma linha.

Veja na figura abaixo:



Agora pegue a segunda e terceira linha da horizontal e a segunda e terceira linha da vertical. Essas linhas são exatamente os lados do quadrado de sobreposição.



Dessas linhas que sobraram, a primeira da vertical é o lado esquerdo desse quadrado e a segunda da vertical é o lado direito desse quadrado. A primeira linha da horizontal é o lado de cima do quadrado e a segunda da horizontal é o lado de baixo do quadrado ("quadrado comum").

Agora para fazer o código disso e achar os lados do 'quadrado comum', faça:

O lado esquerdo desse quadrado (left) - é o segundo lado esquerdo que aparece na tela (considerando os 2 quadrados), ou seja, o maior dos lados esquerdos.

O lado direito desse quadrado (right) - é o primeiro lado direito que aparece na tela (considerando os 2 quadrados), ou seja, o menor dos lados direitos.

O lado de cima desse quadrado (top) - é o segundo lado de cima que aparece na tela (considerando os 2 quadrados), ou seja, o maior dos lados de cima.

O lado de baixo desse quadrado (bottom) - é o primeiro lado de baixo que aparece na tela (considerando os 2 quadrados), ou seja, o menor dos lados de baixo.

Para facilitar na codificação, pode-se fazer funções que retornam o maior valor e o menor valor entre dois números:

```
/*-----*/
#define maior ( a, b ) ( a > b ) ? a : b //retorna maior valor
#define menor( a, b ) ( a < b ) ? a : b //retorna menor valor
/*-----*/
```

Codificando tudo isso para criar uma função que ache esse quadrado de sobreposição:

```
/*-----*/
#define maior ( a, b ) ( a > b ) ? a : b //retorna maior valor
#define menor( a, b ) ( a < b ) ? a : b //retorna menor valor

void achaQuadradoSobreposto( Figura *figura1, Figura *figura2, Quadrado *quadrado ){
    int esquerda1, direita1, cima1, baixo1;
    int esquerda2, direita2, cima2, baixo2;

    esquerda1 = figura1->x;
    direita1 = figura1->x + figura1->largura;
    cima1 = figura1->y;
    baixo1 = figura1->y + figura1->altura;

    esquerda2 = figura2->x;
    direita2 = figura2->x + figura2->largura;
    cima2 = figura2->y;
    baixo2 = figura2->y + figura2->altura;

    quadrado->esquerda = maior( esquerda1, esquerda2 );
    quadrado->direita = menor( direita1, direita2 );
    quadrado->cima = maior( cima1, cima2 );
    quadrado->baixo = menor( baixo1, baixo2 );
}
/*-----*/
```

Essa função então vai colocar na estrutura quadrado passada no terceiro parâmetro, os lados do quadrado de sobreposição das figuras.

Esse quadrado representa a área comum entre as duas figuras.

Agora já com esse quadrado, vamos checar essa área que acabamos de achar (quadrado de sobreposição) em ambas as figuras para ver se há colisão ou não.

Iremos ir pegando pixel a pixel dessa área do quadrado comum de ambas as figuras, e se em um mesmo pixel de cada figura conter uma cor que não é transparente, então retornaremos na função o código de colisão (1), caso contrário, se mesmo após verificar todos os pixels dessa área não contiver um pixel não-transparente na mesma área de cada figura, então retornaremos na função o código de não-colisão (0).

```
/*-----*/
```

```

#define COR_TRANSPARENTE makecol( 255, 0, 255 )

int checaColisaoPixel( Figura *figura1, Figura *figura2, Quadrado *quadrado ){
    int x, y, i, j;
    int pixel1, pixel2;

    for( i = quadrado->esquerda; i < quadrado->direita; i++ ){
        for( j = quadrado->cima; j < quadrado->baixo; j++ ){

            x = i - figura1->x; //acha coordenada x do quadrado comum na figura 1
            y = j - figura1->y; //acha coordenada y do quadrado comum na figura 1
            pixel1 = getpixel( figura1->imagem, x, y ); //pega pixel na figura 1

            x = i - figura2->x; //acha coordenada x do quadrado comum na figura 2
            y = j - figura2->y; //acha coordenada y do quadrado comum na figura 2
            pixel2 = getpixel( figura2->imagem, x, y ); //pega pixel na figura 2

            if( ( pixel1 != COR_TRANSPARENTE ) && ( pixel2 != COR_TRANSPARENTE ) ){
                return 1; //Colisão!!!
            }
        }
    }
    return 0; //não-colisão
}
/*-----*/

```

Observe que eu defini a cor transparente como sendo "makecol(255, 0, 255)" que é o magenta, porque no Allegro em profundidade de cores de 32 bits essa é a cor transparente, ou seja, se vc colocar uma figura com essa cor e desenhar ela no Allegro, os pixels que forem dessa cor não irão aparecer.

Basicamente o que essa função fez foi pegar os pixels da área de sobreposição (quadrado comum) em cada figura e verificar se em um mesmo pixel em cada figura está ou não transparente.

Obs: Quando eu digo quadrado de sobreposição note que não precisa ser necessariamente um quadrado mas sim um retângulo.

Agora podemos criar uma função que quando chamada ela já chama todas essas funções que criamos:

```

/*-----*/
int realizaPixelPerfect( Figura *figura1, Figura *figura2 ){
    Quadrado *quadrado;

    if( checaBoundingBox( figura1, figura2 ) == 1 ){
        achaQuadradoSobreposto( figura1, figura2, &quadrado );
        return checaColisaoPixel( figura1, figura2, &quadrado );
    }
    return 0;
}
/*-----*/

```

```

/*-----*/
#define maior ( a, b ) ( a > b ) ? a : b //retorna maior valor

```

```

#define menor( a, b ) ( a < b ) ? a : b //retorna menor valor

#define COR_TRANSPARENTE makecol( 255, 0, 255 )

typedef struct FIGURA{
    int x;
    int y;
    int largura;
    int altura;
    BITMAP *imagem;
}Figura;

typedef struct QUADRADO{ //retângulo da área dobreposta
    int esquerda;
    int direita;
    int cima;
    int baixo;
}Quadrado;

//checa colisão por BoundingBox
int checaBoundingBox( Figura *figura1, Figura *figura2 ){
    int esquerda1, direita1, cima1, baixo1;
    int esquerda2, direita2, cima2, baixo2;

    esquerda1 = figura1->x;
    direita1 = figura1->x + figura1->largura;
    cima1 = figura1->y;
    baixo1 = figura1->y + figura1->altura;

    esquerda2 = figura2->x;
    direita2 = figura2->x + figura2->largura;
    cima2 = figura2->y;
    baixo2 = figura2->y + figura2->altura;

    if( esquerda1 > direita2 )
        return 0;
    if( direita1 < esquerda2 )
        return 0;
    if( cima1 > baixo2 )
        return 0;
    if( baixo1 < cima2 )
        return 0;

    return 1;
}

void achaQuadradoSobreposto( Figura *figura1, Figura *figura2, Quadrado *quadrado ){
    int esquerda1, direita1, cima1, baixo1;
    int esquerda2, direita2, cima2, baixo2;

    esquerda1 = figura1->x;
    direita1 = figura1->x + figura1->largura;
    cima1 = figura1->y;
    baixo1 = figura1->y + figura1->altura;

```

```

esquerda2 = figura2->x;
direita2 = figura2->x + figura2->largura;
cima2 = figura2->y;
baixo2 = figura2->y + figura2->altura;

```

```

quadrado->esquerda = maior( esquerda1, esquerda2 );
quadrado->direita = menor( direita1, direita2 );
quadrado->cima = maior( cima1, cima2 );
quadrado->baixo = menor( baixo1, baixo2 );
}

```

```

int checaColisaoPixel( Figura *figura1, Figura *figura2, Quadrado *quadrado ){
    int x, y, i, j;
    int pixel1, pixel2;

    for( i = quadrado->esquerda; i < quadrado->direita; i++ ){
        for( j = quadrado->cima; j < quadrado->baixo; j++ ){

            x = i - figura1->x; //acha coordenada x do quadrado comum na figura 1
            y = j - figura1->y; //acha coordenada y do quadrado comum na figura 1
            pixel1 = getpixel( figura1->imagem, x, y ); //pega pixel na figura 1

            x = i - figura2->x; //acha coordenada x do quadrado comum na figura 2
            y = j - figura2->y; //acha coordenada y do quadrado comum na figura 2
            pixel2 = getpixel( figura2->imagem, x, y ); //pega pixel na figura 2

            if( ( pixel1 != COR_TRANSPARENTE ) && ( pixel2 != COR_TRANSPARENTE) ){
                return 1; //Colisão!!!
            }
        }
    }
    return 0; //não-colisão
}

```

```

int realizaPixelPerfect( Figura *figura1, Figura *figura2 ){
    Quadrado quadrado;

    if( checaBoundingBox( figura1, figura2 ) == 1 ){
        achaQuadradoSobreposto( figura1, figura2, &quadrado );
        return checaColisaoPixel( figura1, figura2, &quadrado );
    }
    return 0;
}
/*-----*/

```

Basta agora chamar a função "realizaPixelPerfect" passando as duas estruturas das figuras que queremos testar a colisão, e será retornado 1 caso elas estejam colidindo e 0 caso contrário.

O código fonte de um exemplo de colisão por PixelPerfect pode ser encontrado aqui:

Versão 1: http://filexoom.com/uploaded/2007/9/12/85979/PixelPerfect_%28V%201%29.zip - Exemplo de colisão por PixelPerfect

Versão 2: http://filexoom.com/uploaded/2007/9/12/85979/PixelPerfect_%28V%202%29.zip
 - O mesmo que o de cima só que no exemplo quadrados são desenhados em volta das

figuras e o quadrado de sobreposição também é pintado na tela para um entendimento mais fácil.

Bem então vimos como implementar um algoritmo para teste de colisão utilizando a técnica PixelPerfect. Agora vou mostrar como realizar testes de colisão com PixelPerfect utilizando a biblioteca PMask que já possui funções para teste de colisão por Pixel.

2 – Utilizando a biblioteca PMask

Apesar de mostrar como implementar o código de colisão por Pixel anteriormente, o recomendado (quando estamos programando em Allegro) é utilizar a biblioteca PMask.

Ela é o mais recomendado nesse caso, pois ela já possui funções de detecção de colisão por pixel que são otimizadas além do que utilizando ela você ainda tem muito menos trabalho (já que não precisará implementar o algoritmo PixelPerfect, mas apenas chamar as funções já prontas da biblioteca).

Para utilizar a PMask é muito fácil. Bem primeiro adicione ao projeto os arquivos "pmask.h" e "pmask.c". Feito isto, é só dar um "#include "pmask.h" que você já pode utilizar as funções da Pmask.

A Pmask possui várias funções mais vou falar aqui apenas as mais básicas.

Primeiro vamos criar uma estrutura p/ representar a figura (apenas p/ facilitar a leitura do nosso código).

```
/*-----*/
typedef struct FIGURA{
    int x;
    int y;
    BITMAP *imagem;
    struct PMASK *mask;
}Figura;
/*-----*/
```

Bem, a única novidade até aqui é a variável ponteiro "mask" da estrutura do tipo "struct PMASK". Essa estrutura chamada "struct PMASK" é uma estrutura que a PMask utiliza p/ efetuar o teste de colisão. Cada estrutura dessa é vinculada a uma imagem (BITMAP).

Nesse exemplo vamos supor que temos duas estruturas do tipo FIGURA que são "figura1" e "figura2". Depois de inicializar os campos das estruturas, crie uma estrutura mask utilizando a função "create_allegro_pmask".

```
/*-----*/
figura1.mask = create_allegro_pmask( figura1.imagem );
figura2.mask = create_allegro_pmask( figura2.imagem );
/*-----*/
```

Protótipo da função: *struct PMASK* create_allegro_pmask(BITMAP *bitmap);*

Agora com tudo isso já feito, é só chamar a função "check_pmask_collision" para checar a colisão.

```
/*-----*/
check_pmask_collision( figura1.mask, figura2.mask, figura1.x, figura1.y, figura2.x, figura2.y );
/*-----*/
```

Protótipo da função: *int check_pmask_collision(struct PMASK *mask1, struct PMASK *mask2, int x1, int y1, int*

```
x2, int y2 );
```

O primeiro argumento dessa função é a estrutura pmask da primeira imagem, o segundo argumento é a estrutura pmask da segunda imagem, o terceiro argumento é a coordenada x da primeira imagem, o quarto argumento é a coordenada y da primeira imagem., o quinto argumento é a coordenada x da segunda imagem e o sexto argumento é a coordenada y da segunda imagem.

Essa função irá retornar 1 caso ocorra colisão entre essas duas imagens e irá retornar 0 caso contrário. Então geralmente essa função irá aparecer dentro de um if:

```
/*-----*/  
if( check_pmask_collision( figura1.mask, figura2.mask, figura1.x, figura1.y, figura2.x, figura2.y )  
{  
    /* COLISÃO*/  
}  
else{  
    /* NÃO-COLISÃO*/  
}  
/*-----*/
```

No final do código essas estruturas PMask (struct PMASK) que foram criadas com a função "create_allegro_pmask" devem ser destruídas com a função "destroy_pmask".

```
/*-----*/  
destroy_pmask( figura1.mask );  
destroy_pmask( figura2.mask );  
/*-----*/
```

Protótipo da função: `void destroy_pmask(struct PMASK *mask);`

Pronto. Basicamente é assim que se trata a colisão por PixelPerfect utilizando a biblioteca Pmask. Lembrando que para as partes na imagem em que não se quer que seja detectada colisão, deve-se colocar na cor transparente (makecol(255, 0, 255) – magenta).

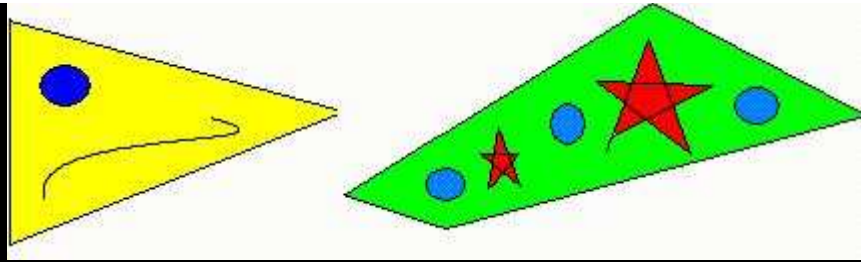
O código fonte de um exemplo de colisão por PixelPerfect utilizando a biblioteca Pmask pode ser encontrado aqui: http://filexoom.com/uploaded/2007/9/12/85979/PixelPerfect_PMask.zip

Colisão entre Polígonos

A colisão PixelPerfect possui um efeito muito bom em todas as situações, entretanto ela pode ser muito lenta. Existem situações em que testes de colisão feitos por BoundingBox não ficam com um bom efeito e testes por PixelPerfect podem causar uma baixa performance no jogo.

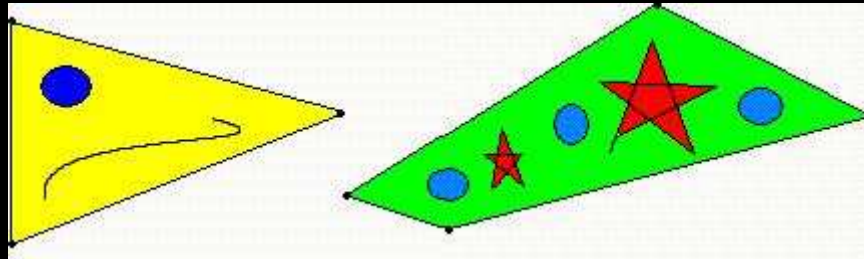
Há casos em que as figuras para os testes de colisão não são muito complexas e por isso não se justifica o uso da técnica de PixelPerfect mas também testes por BoundingBox não tem um bom efeito.

Um exemplo de figuras como essa:



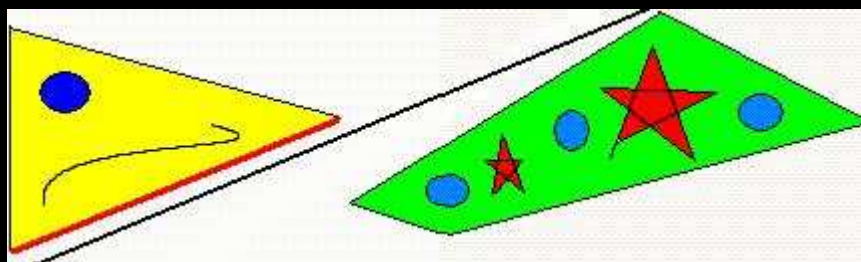
Apesar de essas duas figuras não serem formas complexas com partes arredondadas, um tratamento de colisão entre elas por BoundingBox não teria um bom efeito. Um teste feito por PixelPerfect provavelmente causaria uma performance ruim. Nesse caso, existe uma outra técnica de colisão que é muito mais precisa que a BoundingBox e que também possui uma performance superior aos testes feitos por PixelPerfect. Essa técnica é o teste de colisão por Polígonos.

Essa técnica consiste em pegar uma figura e criar um polígono em volta dessas figuras, ou seja, definir pontos em volta dessas figuras que ligados formam um polígono.



Cada figura agora possui um conjunto de pontos. Nesse caso, a primeira figura (amarela) possui um polígono com 3 pontos e a segunda figura (verde) possui um com 4 pontos. Com as coordenadas desses pontos, agora se pode testar a colisão entre essas figuras.

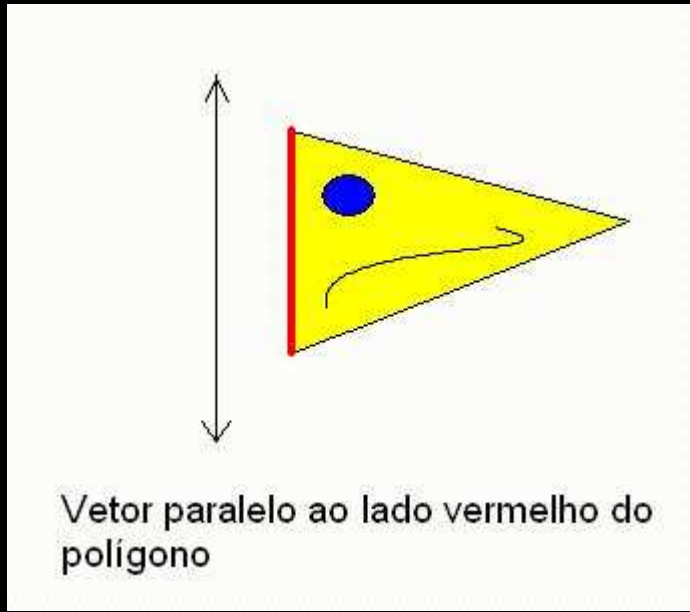
Bem, a técnica que vou explicar para o tratamento de colisões entre polígonos é a seguinte. Sendo os dois polígonos convexos (todos os seus ângulos não ultrapassam 180 graus) e se eles não estiverem se colidindo, então existe uma linha que separa os dois e essa linha é paralela a um dos lados de pelo menos um dos polígonos.



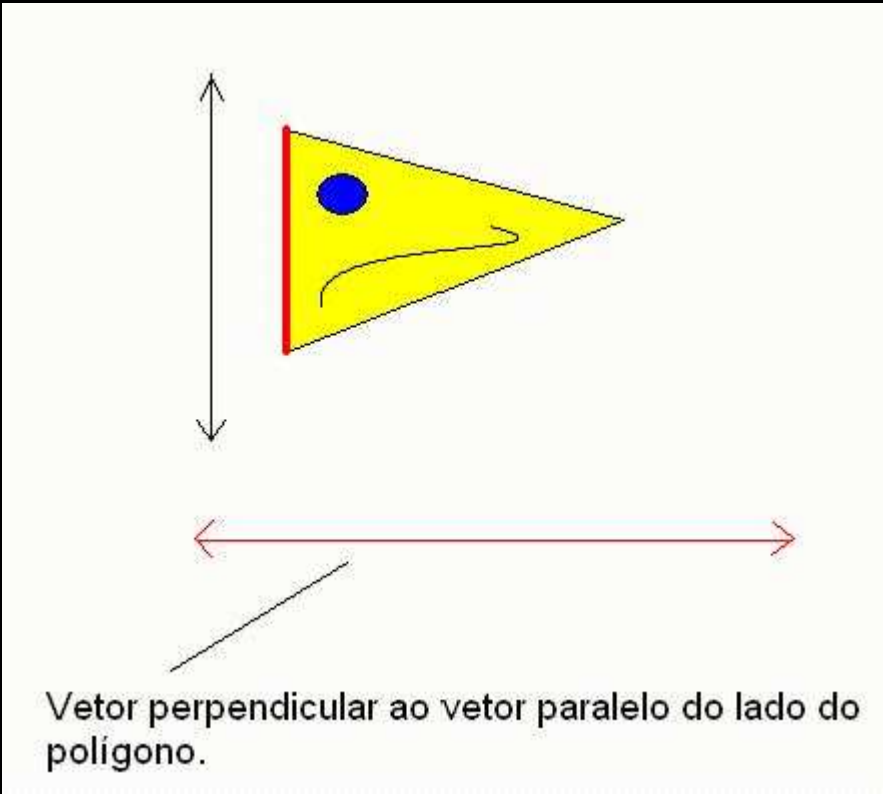
Veja que essa linha preta que separa os dois polígonos é paralela a linha vermelha do polígono amarelo. Então o algoritmo consiste em verificar se essa linha existe, se ela não existe então os polígonos estão se colidindo.

Para achar essa linha a idéia é a seguinte:

Pega-se um lado de um polígono e acha o vetor dessa linha:



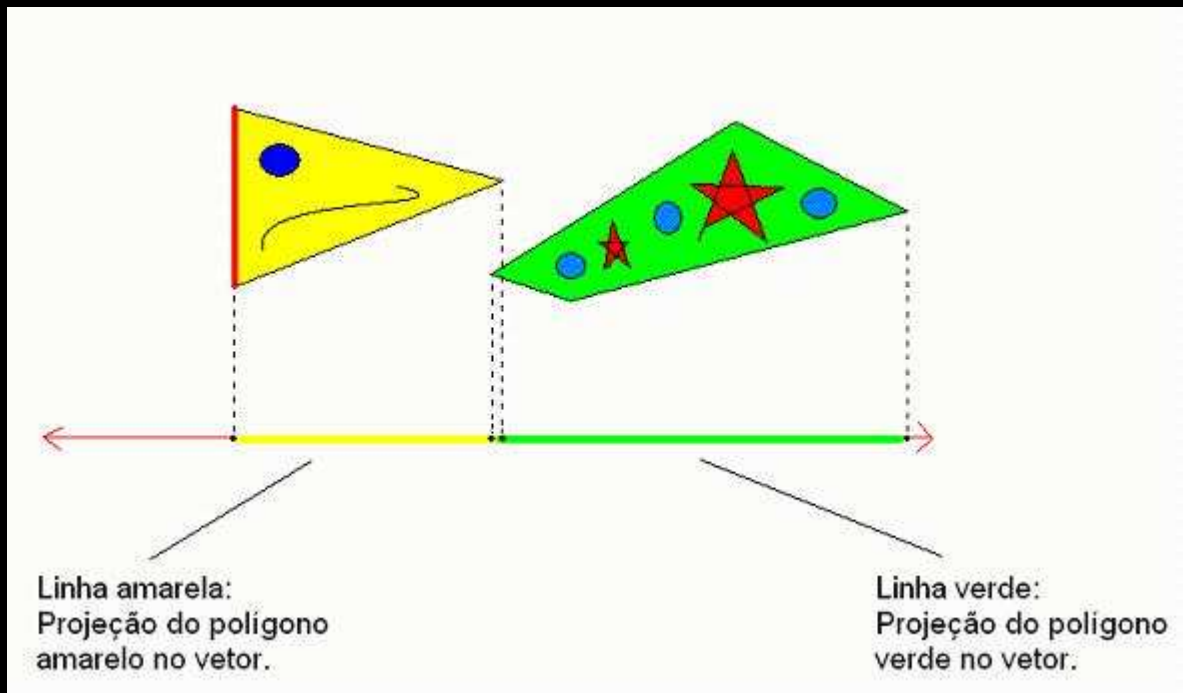
Depois acha-se o vetor perpendicular a esse vetor, ou seja, o vetor que forma um ângulo de 90 graus com esse vetor:



Na figura anterior pode-se ver que o vetor vermelho forma um ângulo de 90 graus com o vetor paralelo ao lado vermelho do polígono.

Feito isso, projeta-se os dois polígonos nesse vetor, ou seja, para cada ponto do polígono acha-se o ponto correspondente no vetor perpendicular e então desenha-se uma linha entre esses pontos.

Veja na figura abaixo:

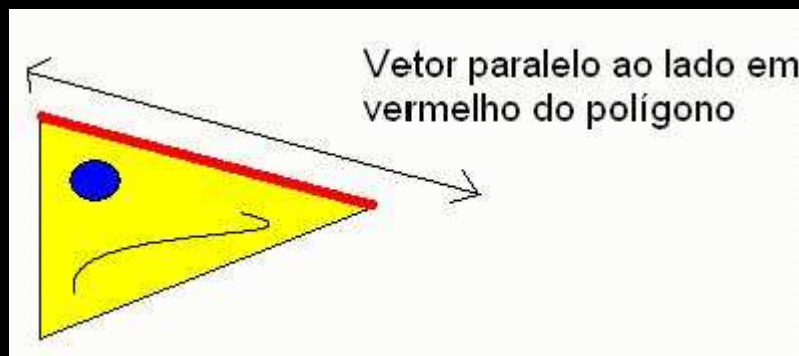


Note que para se achar a projeção do polígono, primeiro projeta-se todos os pontos do polígono no vetor e acha o menor ponto do polígono que foi projetado no vetor e o maior no vetor. Depois, aí desenha-se uma linha entre eles. Essa linha é a projeção do polígono.

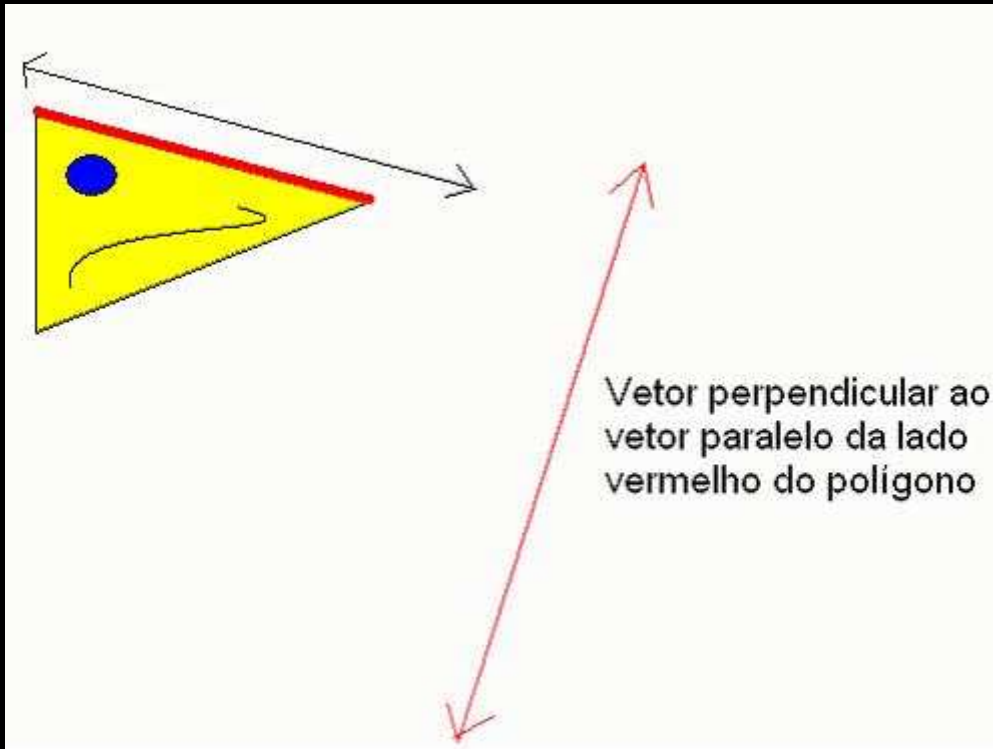
Feito isso, deve-se verificar se as projeções (as linhas projetadas no vetor), estão sobrepostas (se colidindo). Se elas não estão sobrepostas, então já achamos a linha que separa os dois polígonos e já sabemos que eles não se colidem. Caso elas estejam sobrepostas (como no caso acima), devemos repetir o processo para todos as linhas do polígono até que se encontre a linha ou até que se tenha efetuado o processo em todos os lados do polígono. Isso deve ser feito em ambos os polígonos.

No caso acima, pode-se notar que as linhas dos polígonos projetadas no vetor estão se sobrepondo, então vamos pegar outro lado do polígono para realizar o teste.

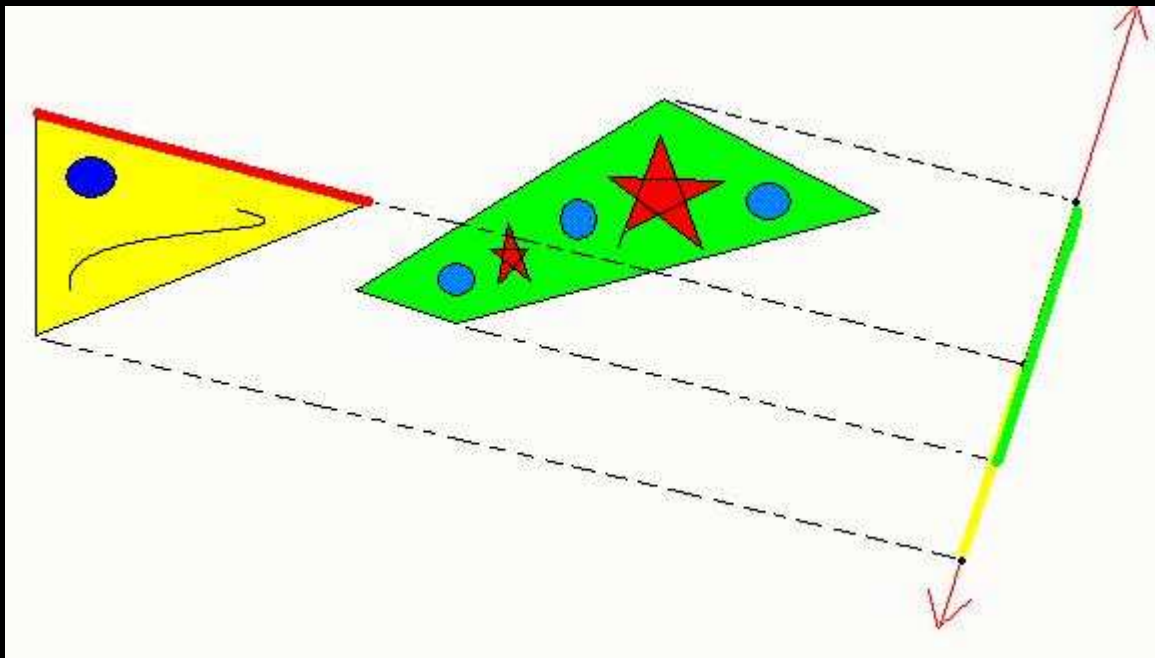
Então vamos pegar um novo lado do polígono e achar o vetor paralelo a esse lado:



Veja agora que pegamos o lado vermelho do polígono e encontramos o vetor paralelo a esse lado. Agora, vamos encontrar o vetor perpendicular a esse vetor que encontramos:

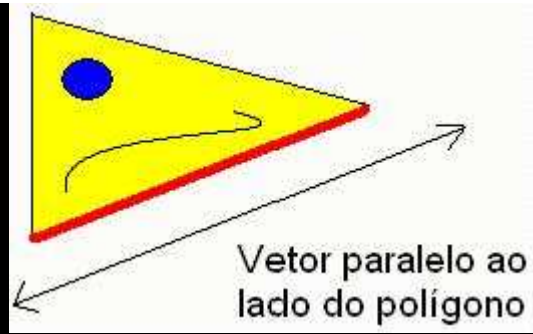


Esse vetor vermelho na figura acima é o vetor perpendicular ao vetor paralelo ao lado vermelho do polígono. Agora vamos projetar os dois polígonos nesse vetor perpendicular encontrado:

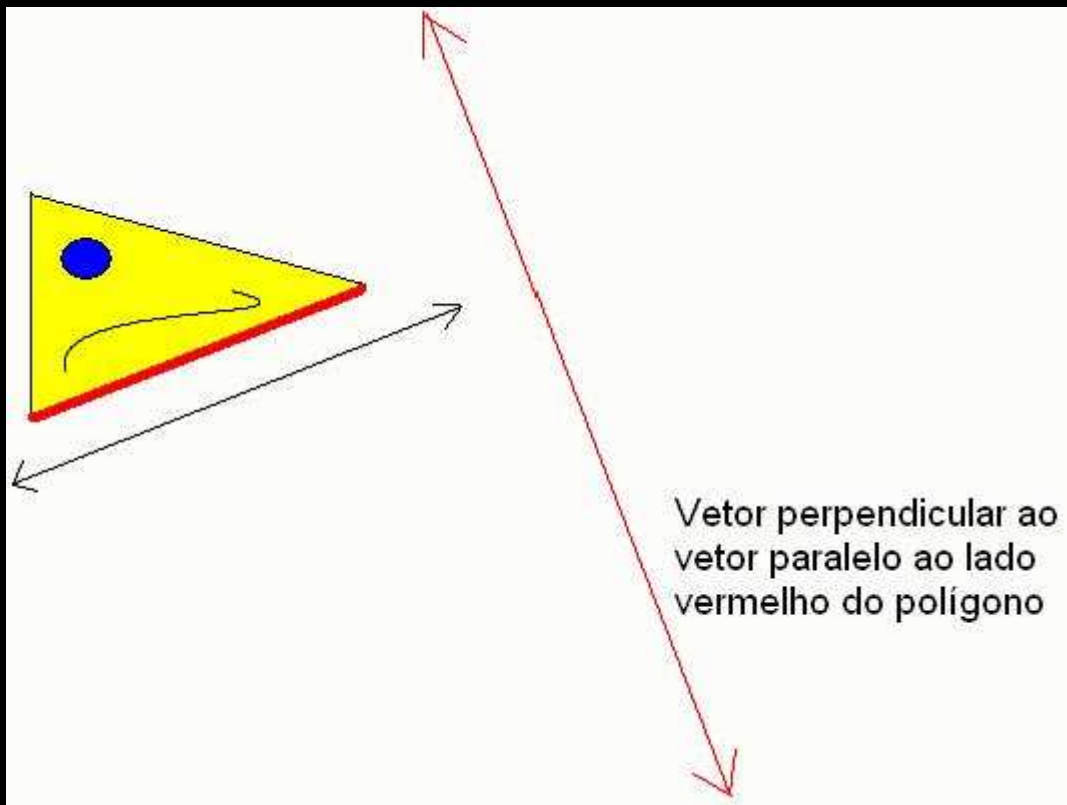


Novamente as projeções estão sobrepostas, então não foi encontrada a linha que divide os dois polígonos. Vamos então repetir o processo com a última linha do polígono amarelo que ainda não testamos:

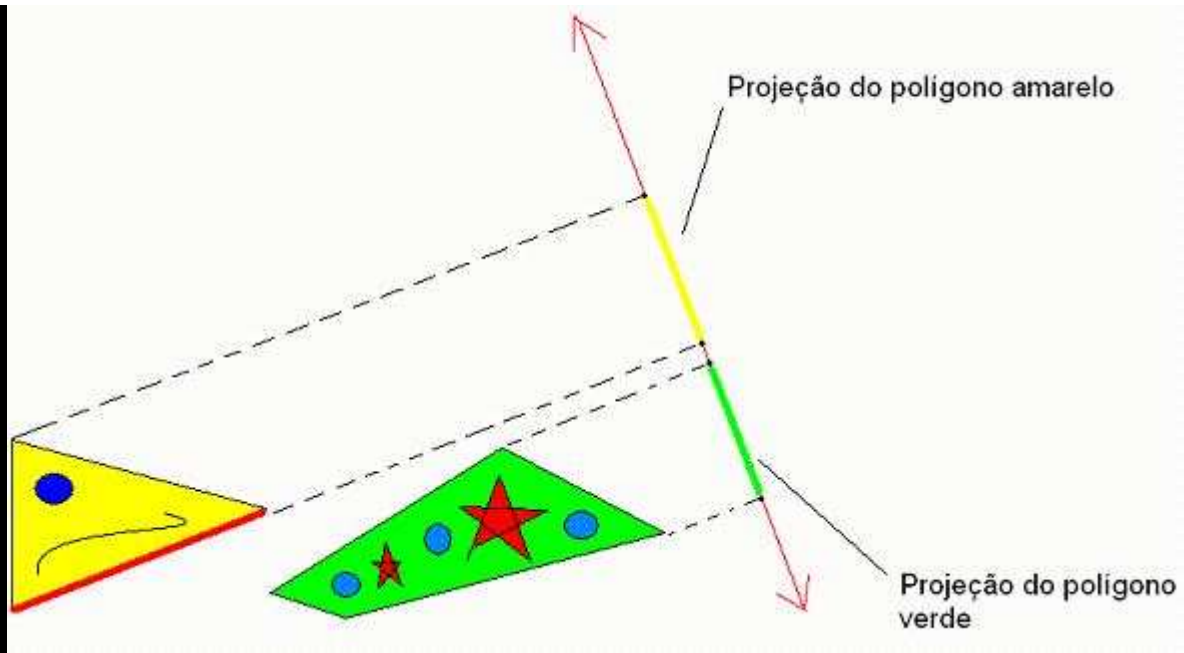
Primeiro então vamos encontrar o vetor paralelo ao lado:



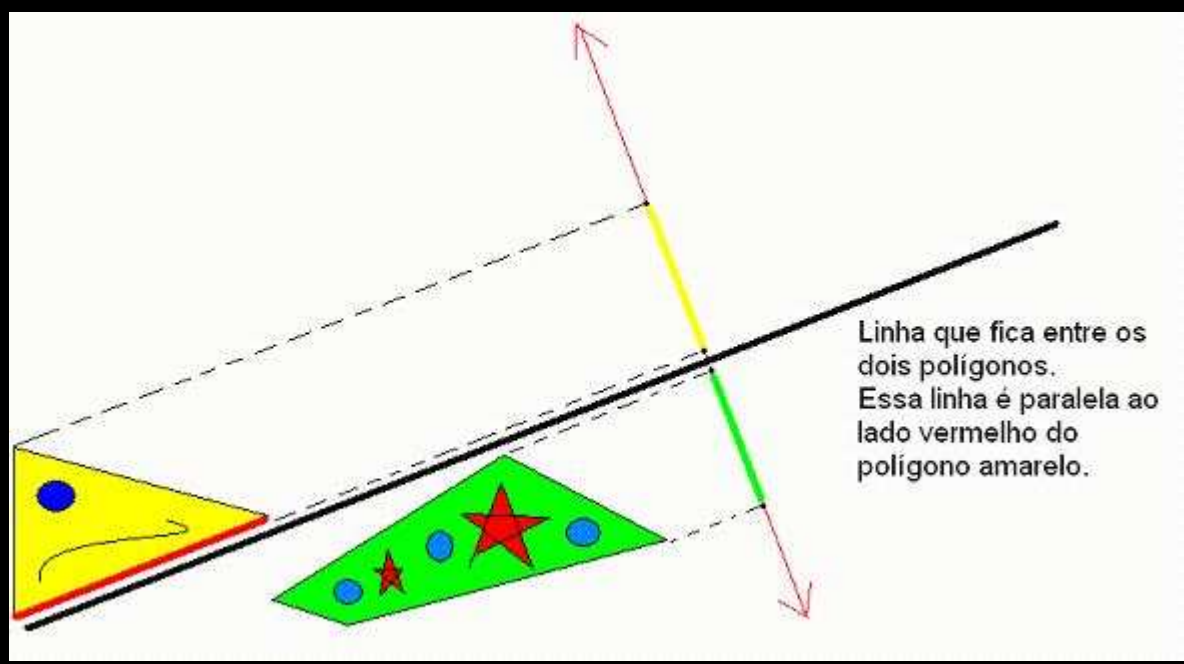
Agora, vamos encontrar o vetor perpendicular a esse vetor:



Vamos então projetar os dois polígonos nesse vetor perpendicular encontrado:



Note que agora as projeções não estão sobrepostas, isso significa que os polígonos não estão se colidindo, pois achamos uma linha que fica entre os dois polígonos:



Como encontramos uma linha que separa os dois polígonos, então isso quer dizer que os polígonos não estão se colidindo. Se nessa etapa não fosse encontrado essa linha, então iríamos repetir o processo para os lados do outro polígono. Se então ainda sim não achássemos essa linha que separa os dois polígonos, então os dois polígonos estariam se colidindo.

Agora que já sabemos a idéia, vamos ver como codificamos tudo isso e fazemos um algoritmo que implemente essa técnica.

Bem, a primeira coisa que vamos fazer é criar uma estrutura que represente um ponto e uma outra que represente um polígono para facilitar o entendimento do código:

```
/*-----*/  
typedef struct PONTO{  
    double x;  
    double y;  
}Ponto;  
  
typedef struct POLIGONO{  
int numeroPontos;  
    Ponto *pontos;  
}Poligono;  
/*-----*/
```

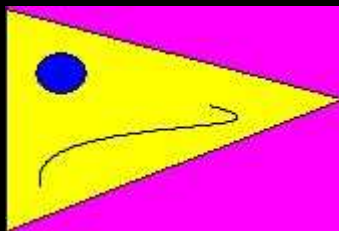
Agora, vamos criar uma estrutura para representar cada figura:

```
/*-----*/  
typedef struct FIGURA{  
    BITMAP *imagem;  
    double x;  
    double y;  
    Poligono poligono;  
}Figura;  
/*-----*/
```

Note que cada estrutura da figura tem um bitmap que é a imagem a ser mostrada na tela, uma coordenada x, uma coordenada y e uma estrutura do tipo polígono. Esse polígono é justamente o que vamos usar para testar a colisão entre essas figuras.

Então primeiramente você encontrar todos os pontos do polígono e preenche nessa estrutura. Para isso você pode abrir a figura no Paint por exemplo e ir vendo e anotando as coordenadas de cada ponto do polígono da figura.

Observe a figura a seguir:



Nessa figura, que é um triângulo, podemos verificar que existem três pontos. Então se formos no Paint e verificarmos as coordenadas dos pontos vamos encontrar o seguinte:

```
Ponto1 - ( 0, 0 )  
Ponto2 - ( 0, 112 )  
Ponto3 - ( 167, 46 )
```

Agora, vamos então criar nossa figura e preencher os valores do polígono dela com esses pontos:

```
/*-----*/
```

```
Figura figura1;
```

```
figura1.poligono.numeroPontos = 3;  
figura1.poligono.pontos = (Ponto*)malloc( sizeof(Ponto) * figura1.poligono.numeroPontos);
```

```
figura1.poligono.pontos[0].x = 0;  
figura1.poligono.pontos[0].y = 0;
```

```
figura1.poligono.pontos[1].x = 0;  
figura1.poligono.pontos[1].y = 112;
```

```
figura1.poligono.pontos[2].x = 167;  
figura1.poligono.pontos[2].y = 46;  
/*-----*/
```

Note que como temos um ponteiro do tipo "Ponto" dentro da estrutura "Polígono", temos que alocar memória para um vetor nesse ponteiro, por isso utilizamos a função "malloc" p/ alocar espaço para 3 estruturas do tipo "Ponto". Não se esqueça também de carregar a imagem e definir as coordenadas iniciais da figura:

```
/*-----*/  
figura1->imagem = load_bitmap( 'arquivoimagem.bmp', NULL );  
figura1.x = 200;  
figura1.y = 200;  
/*-----*/
```

Aqui eu defini as coordenadas iniciais como (200, 200). Agora, como nossa figura vai aparecer na coordenada (200, 200), temos que somar a cada ponto do polígono 200 para as coordenadas x e 200 para as coordenadas y:

```
/*-----*/  
int i;  
  
for( i = 0; i < figura1.poligono.numeroPontos; i++ ){  
    figura1.poligono.pontos[i].x += figura1.x;  
    figura1.poligono.pontos[i].y += figura1.y;  
}  
/*-----*/
```

Se você não quiser fazer isso, você pode somar apenas na hora de tratar a colisão.

Bem, agora vamos criar nossa função de tratamento de colisão. Esse nosso método receberá duas estruturas figuras e retornará TRUE (1) caso elas estejam se colidindo e FALSE (0) caso contrário.

Nossa função deverá fazer o seguinte: projetar os dois polígonos nos vetores encontrados em cada lado do polígono da figura1 e verificar se a reta que separa os dois polígonos existe. Ela também deverá fazer o mesmo com o polígono da figura2, ou seja, projetar os dois polígonos nos vetores encontrados em cada lado do polígono da figura2 e verificar se a reta que separa os dois polígonos existe.

Caso essa reta que separa os dois polígonos seja encontrada, isso significa que os dois polígonos estão se colidindo e por isso nossa função irá retornar FALSE (não-colisão), caso não seja encontrada, isso significa que ela não existe e portanto nossa função irá retornar TRUE (colisão):

P/ facilitar a escrita do retorno da função, vamos definir TRUE como sendo 1 e FALSE como sendo 0:

```
/*-----*/  
#define TRUE    1  
#define FALSE   0
```

```
/*-----*/
```

Agora vamos escrever nossa função de teste de colisão:

```
/*-----*/
```

```
int checaColisao( Figura *figura1, Figura *figura2 ){
    Poligono *poligono1, *poligono2;
    int achou1, achou2;

    poligono1 = &(amp;figura1->poligono);
    poligono2 = &(amp;figura2->poligono);

    achou1 = achouLinhaParalela( poligono1, poligono2 );
    achou2 = achouLinhaParalela( poligono2, poligono1 );

    if( (achou1 == TRUE) || (achou2 == TRUE) ) //se reta que separa 2 polígonos existe
        return FALSE; //não colisão
    return TRUE; //colisão
}
/*-----*/
```

A função "achouLinhaQueSepara" recebe dois polígono como parâmetro. Os dois polígonos passados serão projetados nos vetores do primeiro polígono passado, e ela retorna TRUE caso a linha seja encontrada e FALSE caso contrário. Como os polígonos devem ser projetados nos vetores de ambos os polígonos, essa função é chamada duas vezes. Na primeira vez, ela projeta os dois polígonos nos vetores do polígono1. Na segunda vez, ela projeta os dois polígonos nos vetores do polígono2.

Caso em alguma projeção em algum dos dois polígonos a reta que separa eles seja encontrada, então nossa função retorna FALSE (não-colisão).

Vamos agora escrever nossa função mais importante, a função "achouLinhaQueSepara":

Ela deverá fazer o seguinte, ir pegando dois pontos consecutivos do polígono. Esses pontos são os pontos dos lados do polígono:

```
/*-----*/
```

```
int achouLinhaQueSepara( Poligono *poligono1, Poligono *poligono2 ){
    Ponto *ponto1, *ponto2; //pontos do lado atual do polígono1
    int i;
    int indicePonto1, indicePonto2;

    for( i = 0; i < poligono1->numeroPontos; i++ ){

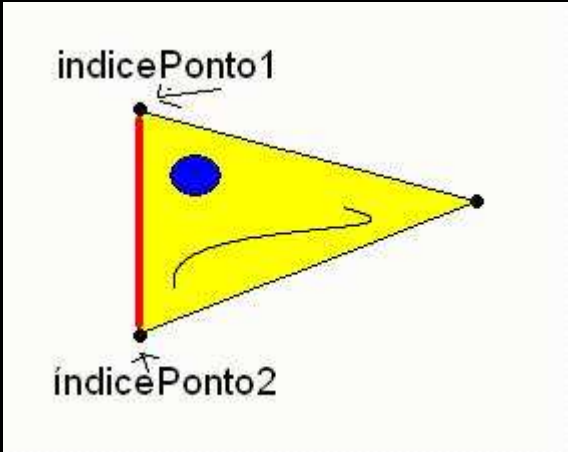
        indicePonto1 = i;
        indicePonto2 = i + 1;
        if( índice >= poligono1->numeroPontos )
            indicePonto2 = 0;

        /* aqui deverá ser encontrado o vetor perpendicular ao vetor paralelo a
           esse lado, e então achar as projeções dos dois polígonos nesse vetor */

        /*verifica aqui se as projeções encontradas se sobrepõem, caso elas não
           se sobreponham, então devemos retornar 0 (não colisão), pois isso significa
           que encontramos a linha que separa os dois polígono.*/
    }
    /* se linha que separa os dois polígono não for encontrada devemos retornar 1 */
    /* por isso, aqui deve ter um return 1; */
}
}
/*-----*/
```

```
/*-----*/
```

Bem, no código acima temos um "for" que vai pegando de dois em dois pontos do polígono (indicePonto1 e indicePonto2) para pegar um dos seus lados. Note que indicePonto1 é o índice do ponto de uma das extremidades desse lado do polígono e indicePonto2 o índice do ponto da outra extremidade da reta do lado do polígono:



Considerando a reta vermelha (que é um dos lados do polígono) "indicePonto1" é o índice do ponto da extremidade de cima da reta vermelha e "indicePonto2" é o índice do ponto da extremidade de baixo dessa reta vermelha.

Temos então esses dois pontos dessa reta. Agora precisamos achar o vetor dessa reta. Para isso vamos fazer a seguinte:

Primeiro vamos criar uma estrutura para esse vetor, como um vetor no R2 (duas dimensões) possui dois campos (x, y), já possuímos uma estrutura como essa que é a estrutura "Ponto" (struct Ponto) criada anteriormente. Vamos então criar um vetor:

```
/*-----*/  
Ponto vetor;  
/*-----*/
```

Agora vamos encontrar esse vetor através dos pontos que temos dessa reta (que é um dos lados do polígono):

```
/*-----*/  
vetor.x = ponto2->x - ponto1->x;  
vetor.y = ponto2->y - ponto1->y;  
/*-----*/
```

Exemplo, vamos considerar os pontos:

Ponto1 (0, 0)
Ponto2 (0, 112)

Nesse caso, vamos encontrar o vetor (0, 112).

Bem, já possuímos nosso vetor da reta do lado do polígono, agora precisamos encontrar o vetor perpendicular a ele. Para achar esse vetor é muito fácil, basta apenas fazer (-y, x).
Ou seja:

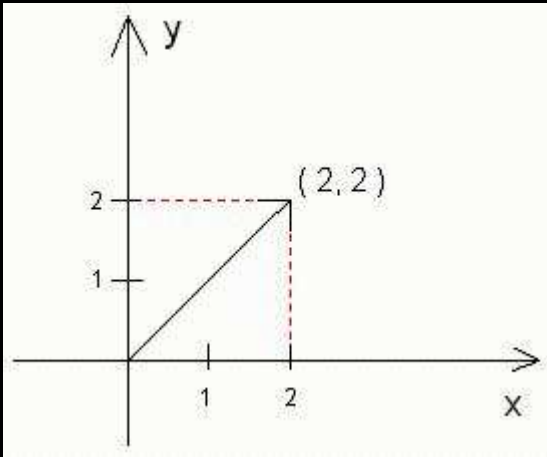
VetorPerpendicular (x, y) = vetor (-y, x)

Então vamos criar mais um vetor para guardar esse vetor perpendicular e vamos encontrá-lo:

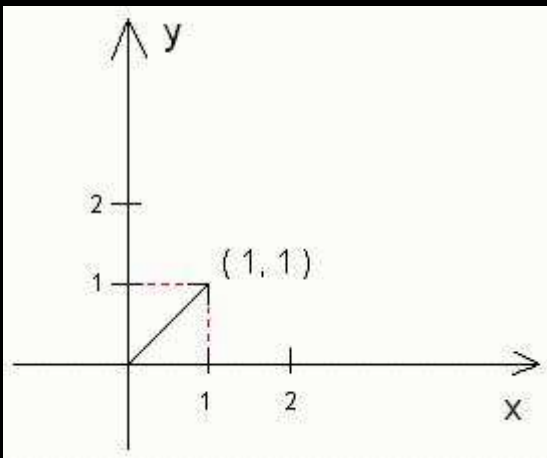
```
/*-----*/  
Ponto vetorPerpendicular;  
vetorPerpendicular.x = -vetor.y;  
vetorPerpendicular.y = vetor.x;  
/*-----*/
```

Pronto, já encontramos o vetor perpendicular. Temos agora que normalizar ele. Isso quer dizer que precisamos reduzir esse vetor para comprimento igual a 1.

Exemplo, considere o vetor (2, 2):



Esse vetor pode ser simplificado ficando assim (1, 1):



Note que o vetor é o mesmo, apenas está com um comprimento diferente (1).

Para normalizar um vetor (achar o mesmo vetor com comprimento 1) temos que achar o comprimento atual dele

e então dividimos o x dele pelo comprimento e o y também, obtendo assim o vetor na forma normalizada. Para achar o comprimento atual, podemos utilizar a seguinte fórmula:

$$\text{Comprimento} = \sqrt{x^2 + y^2}:$$

Vamos então escrever uma função para normalizar um vetor:

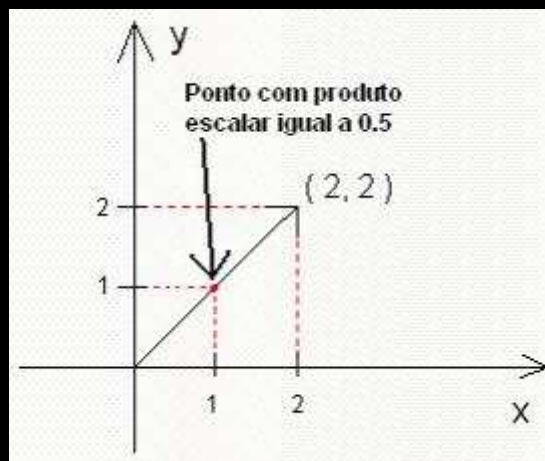
```
/*-----*/  
void normalizaVetor( Ponto *vetor ){  
    double comprimento;  
  
    comprimento = sqrt( (vetor->x * vetor->x) + (vetor->y * vetor->y) );  
    vetor->x = vetor->x / comprimento;  
    vetor->y = vetor->y / comprimento;  
}  
/*-----*/
```

Pronto, então agora vamos normalizar nosso vetor perpendicular que achamos antes com essa função que criamos:

```
/*-----*/  
normalizaVetor( vetorPerpendicular );  
/*-----*/
```

Já com o vetor normalizado, precisamos agora projetar os dois polígonos nesse vetor perpendicular. Para projetar um ponto em um determinado vetor, pode-se utilizar o produto escalar. O produto escalar é uma grandeza do vetor e não valor de coordenadas x e y. Como o próprio nome já diz é uma medida de escala.

Exemplo, um ponto no vetor (2, 2) com produto escalar 0.5:



Veja que o produto escalar é 0.5, mas suas coordenadas x e y não são 0.5. Isso porque o valor 0.5 é uma medida de escala, ou seja, 0.5 é o ponto que fica na metade dessa reta.

Isso é em escala, ou seja, o ponto que fica no final da reta tem produto escalar 1. O que fica na metade da reta tem produto escalar 0.5.

Se tivermos o produto escalar e quisermos obter as coordenadas podemos fazer o seguinte:

$$\text{Coordenada X} = \text{OrigemX} + (\text{ProdutoEscalar} * \text{VetorX})$$

$$\text{Coordenada Y} = \text{OrigemY} + (\text{ProdutoEscalar} * \text{VetorY})$$

Então, no caso da figura acima, as coordenadas do ponto com produto escalar 0.5 seriam:

$$\text{Coordenada X} = 0 + (0.5 * 2)$$

$$\text{Coordenada Y} = 0 + (0.5 * 2)$$

Logo, obtemos:

$$\text{Coordenada X} = 1$$

$$\text{Coordenada Y} = 1$$

Então, para acharmos um ponto projetado nesse vetor, devemos achar seu produto escalar. Para isso, tudo que temos que fazer é multiplicar as coordenadas desse ponto por esse vetor:

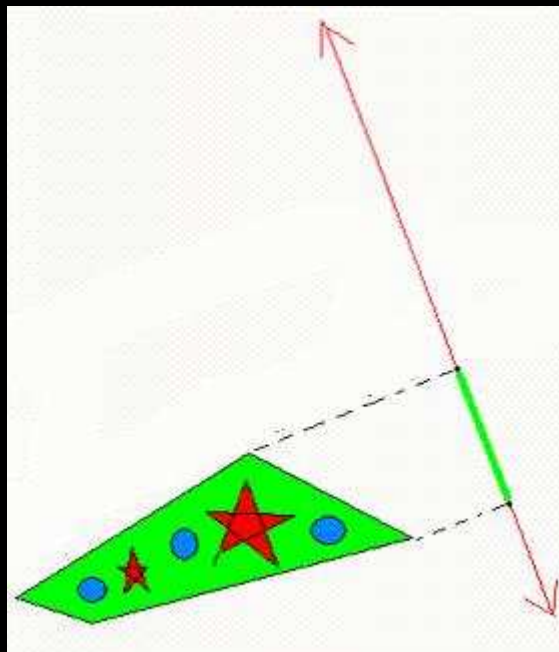
Exemplo:

$$\text{Ponto}(a, b) * \text{Vetor}(x, y) = a * x + b * y = \text{ProdutoEscalar}$$

Tudo que precisamos nesse caso p/ a implementação do algoritmo é dos produtos escalares.

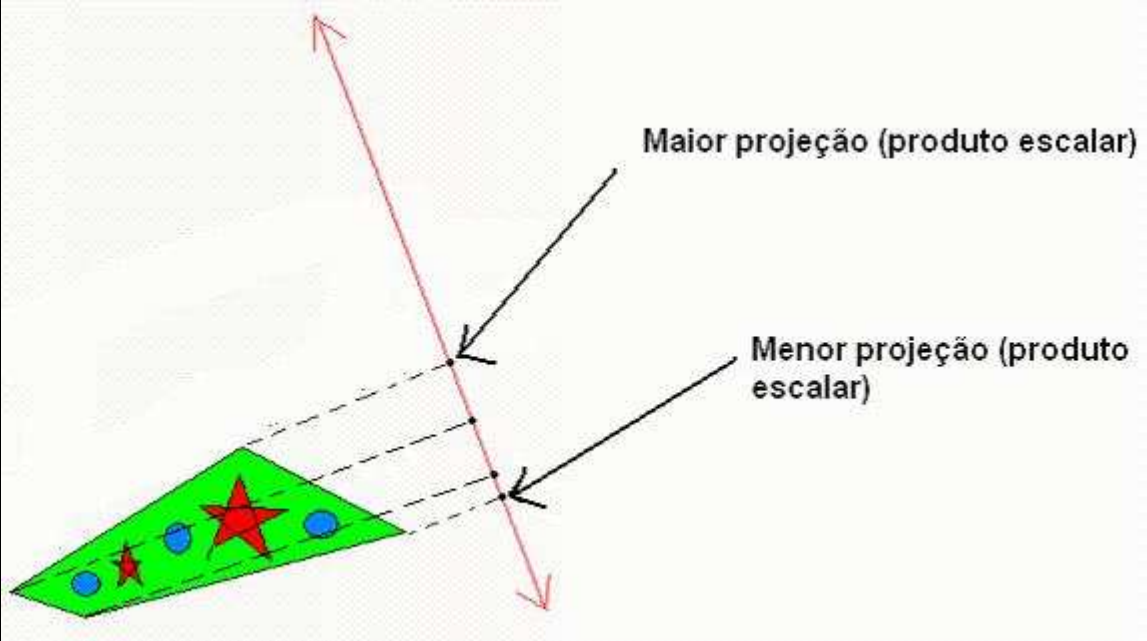
Na verdade não precisamos das coordenadas das projeções, pois os produtos escalares já nos dizem onde o ponto da projeção se localiza na reta e por isso já é possível verificar se uma projeção se sobrepõe a outra.

Veja a figura abaixo:

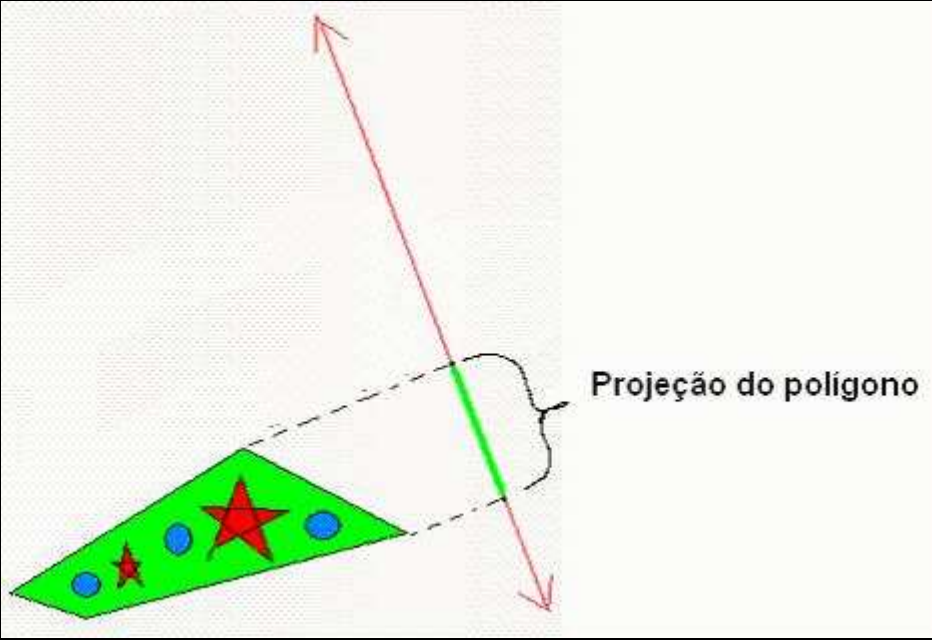


Note que a projeção desse polígono pode ser achada projetando todos os pontos no vetor, achando assim o produto escalar de cada ponto, e depois verificando o menor e o maior produtos escalares (pontos projetados na

reta). Esses pontos são as retas projetadas:



Então, obtendo o menor ponto projetado e o maior, obtemos a projeção do polígono no vetor:



Então vamos fazer uma função que dado um polígono e um vetor, ache essa projeção desse polígono nesse vetor dado.
Devemos então projetar todos os pontos desse polígono nesse vetor (achar os produtos escalares desses pontos) e pegar o menor e o maior. Esses são os dois pontos de nossa projeção.

Vamos criar uma estrutura p/ representar uma projeção. Essa estrutura deve conter dois campos, menor produto escalar e maior produto escalar:

```
/*-----*/
```

```
typedef struct PROJECAO{
    double menor;
    double maior;
}Projecao;
/*-----*/
```

Veja que poderíamos usar a estrutura Ponto criada anteriormente (struct PONTO) pois ela possui dois campos que também são do tipo double, mas apenas para facilitar a leitura do código definimos essa estrutura Projecao.

Como precisamos achar o menor produto escalar e o maior, vamos definir o seguinte para facilitar:

```
/*-----*/
#define MAIOR(a, b) (a > b) ? a : b
#define MENOR(a, b) (a < b) ? a : b
/*-----*/
```

```
/*-----*/
void projetaPoligono( Poligono *poligono, Ponto *vetor, Projecao *projecao ){
    int i;
    double produtoEscalar;
    Ponto *pontoAtual;

    pontoAtual = &(amp;poligono->Pontos[0]);

    produtoEscalar = (vetor->x * pontoAtual->x) + (vetor->y * pontoAtual->y);

    projecao->menor = produtoEscalar;
    projecao->maior = produtoEscalar;

    for( i = 1; i < poligono->numeroPontos; i++ ){
        pontoAtual = &(amp;poligono->Pontos[i]);

        produtoEscalar = (vetor->x * pontoAtual->x) + (vetor->y * pontoAtual->y);

        projecao->maior = MAIOR( projecao->maior, produtoEscalar );
        projecao->menor = MENOR( projecao->menor, produtoEscalar );
    }
}
/*-----*/
```

Note que essa função percorre cada ponto do polígono e acha sua projeção no vetor (seu produto escalar), depois guarda o menor e o maior na estrutura projeção passada.

Devemos então achar as projeções dos dois polígonos no vetor perpendicular ao lado que estamos testando. Anteriormente achamos esse vetor que está na variável "vetorPerpendicular".

Vamos então achar as projeções dos dois polígonos nesse vetor:

```
/*-----*/
Projecao projecao1, projecao2;

projetaPoligono( poligono1, vetorPerpendicular, &projecao1 );
projetaPoligono( poligono2, vetorPerpendicular, &projecao2 );
/*-----*/
```

Temos agora em projecao1 a projeção do polígono1 e em projecao2 a projeção do polígono2. Tudo que precisamos fazer agora é verificar se essas duas projeções (retas) se sobrepõem. Uma função dessas poderia ser escrita da seguinte maneira;

```
/*-----*/
int checaColisaoProjecoes( Linha *projecao1, Linha *projecao2 ){
    if( projecao1->maior < projecao2->menor ) return 0;
    if( projecao1->menor > projecao2->maior ) return 0;

    return 1;
}
/*-----*/
```

Essa função retorna 1 caso as projeções (retas) estejam se colidindo (sobrepostas) e 0 caso elas não estejam.

Com isso, podemos então escrever a função "achaLinhaQueSepara" completa:

```
/*-----*/
int achouLinhaQueSepara( Poligono *poligono1, Poligono *poligono2 ){
    Ponto *ponto1, *ponto2; //pontos do lado atual (lado que está verificando) do polígono1
    Ponto vetorAtual, vetorPerpendicular;
    int i;
    int indicePonto1, indicePonto2;
    Projeção projecao1, projecao2;

    for( i = 0; i < poligono1->numeroPontos; i++ ){
        indicePonto1 = i;
        indicePonto2 = i+1;
        if( indicePonto2 >= poligono1->numeroPontos )
            indicePonto2 = 0;

        ponto1 = &(poligono1->pontos[indicePonto1]);
        ponto2 = &(poligono1->pontos[indicePonto2]);

        vetorAtual.x = ponto2->x - ponto1->x;
        vetorAtual.y = ponto2->y - ponto1->y;

        /****** Acha vetor perpendicular (-y, x) *****/
        vetorPerpendicular.x = -vetorAtual.y;
        vetorPerpendicular.y = vetorAtual.x;

        normalizaVetor( &vetorPerpendicular ); //normaliza vetor
        /******

        /****** Acha projeções dos dois polígonos no vetor perpendicular*****/
        projetaPoligono( poligono1, &vetorPerpendicular, &projecao1 );
        projetaPoligono( poligono2, &vetorPerpendicular, &projecao2 );
        /******

        if( checaColisaoProjecoes( &projecao1, &projecao2 ) == FALSE )
            return 1; //achou linha que divide dis polígonos
    }
}
```

```

    return 0; //não achou linha que divide os dois polígonos
}
/*-----*/

```

Vamos agora juntar tudo que fizemos até aqui p/ implementarmos o algoritmo de teste de colisão:

```

/*-----*/
#define TRUE 1
#define FALSE 0

#define MAIOR(a, b) (a > b) ? a : b
#define MENOR(a, b) (a < b) ? a : b

typedef struct PONTO{
    double x;
    double y;
}Ponto;

typedef struct POLIGONO{
    int numeroPontos;
    Ponto *pontos;
}Poligono;

typedef struct PROJECAO{
    double menor;
    double maior;
}Projeção;

typedef struct FIGURA{
    BITMAP *imagem;
    double x;
    double y;
    Poligono poligono;
}Figura;

/***** Protótipo das funções *****/
int checaColisao( Figura *figura1, Figura *figura2 );

int achouLinhaQueSepara( Poligono *poligono1, Poligono *poligono2 );
void normalizaVetor( Ponto *vetor );
void projetaPoligono( Poligono *poligono, Ponto *vetor, Projecao *projecao );
int checaColisaoProjecoes( Projecao *projecao1, Projecao *projecao2 );
/*****

int checaColisao( Figura *figura1, Figura *figura2 ){
    Poligono *poligono1, *poligono2;
    int achou1, achou2;

    poligono1 = &(figura1->poligono);
    poligono2 = &(figura2->poligono);

```

```

    achou1 = achouLinhaParalela( poligono1, poligono2 );
    achou2 = achouLinhaParalela( poligono2, poligono1 );

    if( (achou1 == TRUE) || (achou2 == TRUE) ) //se reta que separa 2 polígonos existe
        return FALSE; //não colisão
    return TRUE; //colisão
}

```

```

int achouLinhaQueSepara( Poligono *poligono1, Poligono *poligono2 ){
    Ponto *ponto1, *ponto2; //pontos do lado atual (lado que está verificando) do polígono1
    Ponto vetorAtual, vetorPerpendicular;
    int i;
    int indicePonto1, indicePonto2;
    Projecção projecao1, projecao2;

    for( i = 0; i < poligono1->numeroPontos; i++ ){
        indicePonto1 = i;
        indicePonto2 = i+1;
        if( indicePonto2 >= poligono1->numeroPontos )
            indicePonto2 = 0;

        ponto1 = &(poligono1->pontos[indicePonto1]);
        ponto2 = &(poligono1->pontos[indicePonto2]);

        vetorAtual.x = ponto2->x - ponto1->x;
        vetorAtual.y = ponto2->y - ponto1->y;

        /***** Acha vetor perpendicular (-y, x) *****/
        vetorPerpendicular.x = -vetorAtual.y;
        vetorPerpendicular.y = vetorAtual.x;

        normalizaVetor( &vetorPerpendicular ); //normaliza vetor
        /*****/

        /***** Acha projeções dos dois polígonos no vetor perpendicular *****/
        projetaPoligono( poligono1, &vetorPerpendicular, &projecao1 );
        projetaPoligono( poligono2, &vetorPerpendicular, &projecao2 );
        /*****/

        if( checaColisaoProjecoes( &projecao1, &projecao2 ) == FALSE )
            return 1; //achou linha que divide dis polígonos
    }

    return 0; //não achou linha que divide os dois polígonos
}

```

```

void normalizaVetor( Ponto *vetor ){
    double comprimento;

    comprimento = sqrt( (vetor->x * vetor->x) + (vetor->y * vetor->y) );
    vetor->x = vetor->x / comprimento;
    vetor->y = vetor->y / comprimento;
}

```

```

void projetaPoligono( Poligono *poligono, Ponto *vetor, Projecao *projecao ){
    int i;
    double produtoEscalar;
    Ponto *pontoAtual;

    pontoAtual = &(amp;poligono->pontos[0]);

    produtoEscalar = (vetor->x * pontoAtual->x) + (vetor->y * pontoAtual->y);

    projecao->menor = produtoEscalar;
    projecao->maior = produtoEscalar;

    for( i = 1; i < poligono->numeroPontos; i++ ){
        pontoAtual = &(amp;poligono->pontos[i]);

        produtoEscalar = (vetor->x * pontoAtual->x) + (vetor->y * pontoAtual->y);

        projecao->maior = MAIOR( projecao->maior, produtoEscalar );
        projecao->menor = MENOR( projecao->menor, produtoEscalar );
    }
}

int checaColisaoProjecoes( Linha *projecao1, Linha *projecao2 ){
    if( projecao1->maior < projecao2->menor ) return 0;
    if( projecao1->menor > projecao2->maior ) return 0;

    return 1;
}
/*-----*/

```

Essas são as funções da implementação. Não coloquei funções não relacionadas a detecção de colisão como funções de inicialização das estruturas da figura.

Um exemplo com tudo que vimos até aqui sobre teste de colisão entre polígonos pode ser encontrado aqui: <http://filexoom.com/uploaded/2007/9/12/85979/PolygonCollision.zip>

Não vou mostra aqui mas você pode otimizar seu código, por exemplo, antes de realizar o teste de colisão por polígonos, realizar o teste por BoundingBox nesses dois polígonos. Assim, caso o BoundingBox já detectasse não-colisão, nem precisaríamos gastar processamento efetuando o teste de colisão por polígonos.

Mostrei aqui como testar colisão entre dois polígonos convexos. Existem ainda outras formas de detecção de colisão entre polígonos que não vou mencionar aqui.

Aqui tem o link com todos os exemplos deste artigo:

<http://filexoom.com/uploaded/2007/9/12/85979/Tudo.zip>

Aqui estão alguns links sobre teste de colisão entre polígonos:

http://www.gpwiki.org/index.php/Polygon_Collision

<http://www.codeproject.com/cs/media/PolygonCollision.asp>

Links sobre produto escalar e projeção em vetores:

<http://pessoal.sercomtel.com.br/matematica/geometria/vetor2d/vetor2d.htm>

<http://tutorial.math.lamar.edu/AllBrowsers/2414/DotProduct.asp>

E termina aqui o artigo sobre técnicas de detecção de colisão para jogos.

Procurei mostrar aqui as principais técnicas utilizadas para teste de colisão para jogos 2D.

Aconselho a ver bem os exemplos. Todos eles foram feitos no Dev-Cpp, e contém o executável e o código fonte.

Versão deste artigo em .DOC:

<http://filexoom.com/uploaded/2007/9/12/85979/Tecnicas%20de%20colisao%20para%20jogos.doc>

Escrito por: Gustavo Russo Zanardo



[Imprimir](#)

Copyright © UniDev 2005