DETECÇÃO DE COLISÕES AUXILIADAS POR BOUNDING BOXES

Aluno: Matheus Felipe Ferreira Maciel
Orientador: Thomas Lewiner

Introdução

Detecções de colisões em sistemas gráficos exigem estruturas eficientes para evitar altos custos computacionais. Tais sistemas gráficos podem incluir jogos 3D, modelos computacionais de estruturas como plataformas de petróleo e até mesmo simulações de aerodinâmica de aviões modernos.

Ferramentas como as *Bounding Boxes* são extremamente úteis nesses testes, pois nos permitem que uma malha de triângulos como um grande número de pontos possa ser reduzida a uma estrutura mais simples de ser trabalhada e testada.

Uma *Bounding Box* é basicamente um componente que contém uma malha de triângulos, definido assim o que é exterior e o que interior a malha em questão. Essa definição nos permite realizar testes simples de colisão utilizando pouca capacidade computacional ao custo de precisão.

Esse trabalho mostra uma forma de utilizar estruturas representativas chamadas AABBs e OBBs (*Axis Aligned Bounding Boxes* e *Oriented Aligned Bounding Boxes*, respectivamente) e realizar testes de colisão computacionalmente eficientes utilizando tais estruturas. Sendo assim, sua estrutura se define em duas partes: Definir e mostrar algoritmos para a construção de *Bounding Boxes* e para testes de colisões entre as mesmas, dentro da infinidade de maneiras possíveis de realizar tais tarefas.

Tipos de estruturas utilizadas e métodos de cálculo

- ***Axis Aligned Bounding Box***

Esta estrutura se baseia nos eixos padrões X, Y e Z. Devido a sai definição, os pontos utilizados pela sua demarcação são baseados basicamente nas coordenadas das extremidades da malha de triângulos em questão. Para efeitos de teste, foram criadas algumas maneiras de representação de uma AABB, a notar:

1. De acordo com os pontos da extremidade da malha de triângulos, conforme dito anteriormente;
2. Utilizando o ponto superior direito anterior e inferior esquerdo frontal da malha de triângulos;
3. Calculando-se o centro da malha de triângulos e a distância até as extremidades dos eixos X, Y e Z da malha de triângulos.

Dessas três maneiras, temos uma representação fiel de uma *Axis Aligned Bounding Box* conforme dados colhidos de uma malha de triângulos. Tais informações são vitais para a detecção de colisões entre modelos. Sua definição é rápida e não requer a mesma quantidade de esforço computacional que a próxima estrutura, mas a precisão na representação de modelos é sacrificada em caso que os pontos da malha de triângulos têm uma variância muito elevada.

- ***Oriented Bounding Box***

Esta estrutura visa aumentar a precisão na representação de um modelo, requerendo mais informações e maior esforço computacional, tendo em vista que necessita um maior processamento de dados. O procedimento de cálculo é mostrado a seguir:

Primeiramente, é necessário calcular o ponto médio da amostra de dados (pontos, no caso da estrutura da malha de triângulos), sendo ele o "centro de massa". Sendo nossa base de dados denotada da seguinte forma para R^d,

$$\mathbf{x_1} = \begin{pmatrix} x_1^1 \\ x_1^2 \\ \vdots \\ x_1^d \end{pmatrix}, \ \mathbf{x_2} = \begin{pmatrix} x_2^1 \\ x_2^2 \\ \vdots \\ x_2^d \end{pmatrix}, \ \ldots, \ \mathbf{x_n} = \begin{pmatrix} x_n^1 \\ x_n^2 \\ \vdots \\ x_n^d \end{pmatrix}$$

O centro de massa é definido por:

$$\mathbf{m} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i$$

Daí, denotando $y_i = x_i - m$, i = 1, 2, ..., n, temos a matris $S = YY^T$ que corresponde a variância da base de dados e Y corresponde a matriz dxn com y_k representada como as colunas da mesma:

$$S = \begin{pmatrix} y_1^1 & y_2^1 & \cdots & y_n^1 \\ y_1^2 & y_2^2 & & y_n^2 \\ \vdots & \vdots & & \vdots \\ y_1^d & y_2^d & \cdots & y_n^d \end{pmatrix} \begin{pmatrix} y_1^1 & y_1^2 & \cdots & y_1^d \\ y_2^1 & y_2^2 & \cdots & y_2^d \\ \vdots & & & \vdots \\ y_n^1 & y_n^2 & \cdots & y_n^d \end{pmatrix}$$

Observando uma linha L que passa pelo centro de massa e projetar os pontos xi nessa linha, podemos determinar a variância de L como sendo:

$$\mathrm{var}(L) = \frac{1}{n} \sum_{i=1}^{n} \| \mathbf{x}_i' - \mathbf{m} \|^2$$

Dada a direção V e sendo seu módulo igual a um, a projeção de x_i em L = m + VT é:

$$\| \mathbf{x}_i' - \mathbf{m} \| = < \mathbf{v}, \ \mathbf{x}_i - \mathbf{m} > / \| \mathbf{v} \| = < \mathbf{v}, \ \mathbf{y}_i > = \mathbf{v}^T \mathbf{y}_i$$

Sendo assim, temos a equação de deve ser substituída para determinamos a variância dos pontos projetados sobre L:

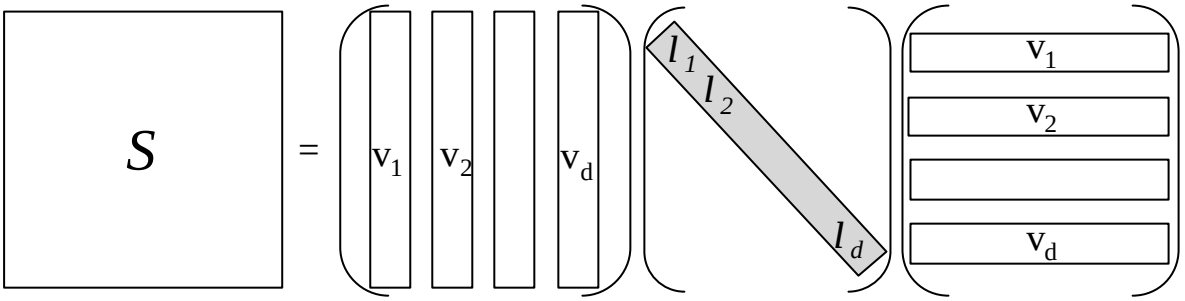
$$\mathrm{var}(L) = \frac{1}{n} \sum_{i=1}^{n} \| \mathbf{x}_i' - \mathbf{m} \|^2 = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{v}^T \mathbf{y}_i)^2 = \frac{1}{n} \| \mathbf{v}^T Y \|^2 =$$

$$= \frac{1}{n} \| Y^T \mathbf{v} \|^2 = \frac{1}{n} < Y^T \mathbf{v}, \ Y^T \mathbf{v} > = \frac{1}{n} \mathbf{v}^T Y Y^T \mathbf{v} = \frac{1}{n} \mathbf{v}^T S \mathbf{v} = \frac{1}{n} < S\mathbf{v}, \ \mathbf{v} >$$

$$\sum_{i=1}^{n}(\mathbf{v}^T\mathbf{y_i})^2=\sum_{i=1}^{n}\left(\begin{pmatrix}v^1 & v^2 & \cdots & v^d\end{pmatrix}\begin{pmatrix}y_i^1\\ y_i^2\\ \vdots\\ y_i^d\end{pmatrix}\right)^2=\left\|\begin{pmatrix}v^1 & v^2 & \cdots & v^d\end{pmatrix}\begin{pmatrix}y_1^1 & y_2^1 & \cdots & y_n^1\\ y_1^2 & y_2^2 & & y_n^2\\ \vdots & \vdots & & \vdots\\ y_1^d & y_2^d & \cdots & y_n^d\end{pmatrix}\right\|^2=\|\mathbf{v^T}Y|$$

Essa definição nos leva ao seguinte teorema:

Sendo f : {v ∈ Rd | ||v|| = 1} → R, f (v) = <Sv, v> (tendo S como uma matriz simétrica), podemos concluir que os autovetores de S são as direções da variâncias máximas e mínimas. Ainda trabalhando com S, sua diagonalização é realizada por:



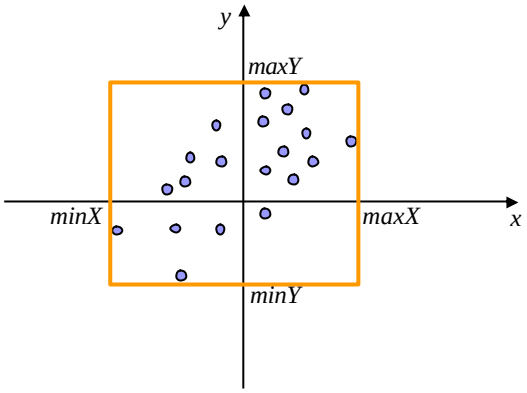
V1, V2, ..., VD correspondem aos autovetores da base ortogonal. Analisando esses autovetores, podemos concluir que aqueles que correspondem aos maiores autovalores são as direções em que a quantidade de informações (pontos da malha) são mais presentes. Com isso, caso os autovalores tenham valores similares, não existe uma direção preferencial para os dados.

Essa base nos permite calcular uma *Oriented Bounding Box* utilizando os eixos definidos pelos autovetores e utilizando a origem como sendo o centro de massa do sistema. Para realizar o calculo efetivamente, é necessário projetar os pontos conhecidos sobre um subespaço definido pelos autovetores. Assim, temos uma representação dos dados que nos permitem montar *Bounding Boxes* com precisão.

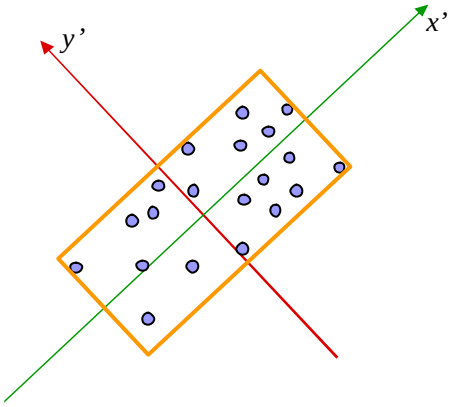
Diferenças entre *Axis Aligned Bounding Boxes* (AABBs) e *Oriented Bounding Boxes* (OBBs)

Uma AABB segue os eixos padrões X, Y e Z, sendo sujeito imprecisões na representação de um conjunto de dados. Essa imprecisão é compensada com o número reduzido de informações que são necessárias ser armazenadas pra representá-las. Já no caso da OBB, é necessário guardar uma quantidade maior de dados, já que a representação da mesma se baseia em vetores e operações vetoriais. Esse maior esforço computacional para o cálculo dessas é compensado pela maior precisão oferecida pelo caixa formada. Um exemplo de como essas duas estruturas pode ser visto abaixo:

Axis Aligned Bounding Box representando um conjunto de dados:



Oriented Bounding Box representando um mesmo conjunto de dados:



Com isso, podemos observar que encontrando eixos melhores a partir da PCA (*Principal Component Analysis*)[1], conseguimos montar OBBs que contém conjuntos de dados com maior precisão.

Uma estrutura possível para a representação de uma AABB e uma OBB (em C++) pode ter o seguinte formato (Sem incluir métodos possíveis para o mesmo):

```
class AABB {
AABB () ;   // Class Constructor
~AABB () {}   // Class Destructor

  double _posx1, _posy1, _posz1;   //          Bounding box position number 1
  double _posx2, _posy2, _posz2;   //          Bounding box position number 2
  double _dimx, _dimy, _dimz;   // Bounding box dimension
  double _centerX, _centerY, _centerZ;   // Bounding box center
} ;

class OrientedBoundingBox {
BoundingBox () ;   // Class Constructor
```

```
~BoundingBox () {}   // Class Destructor

  double _posx1, _posy1, _posz1    ;   // Bounding box position
  double _posx2, _posy2, _posz2    ;   // Bounding box position
  double _dimx, _dimy, _dimz       ; // Bounding box dimension
  double _directionMatrix[ 3 ][ 3 ] ;   // Direction matrix for bounding box
} ;
```

Comparando as estruturas, a OBB utiliza mais seis tipos de dado Double, o que significa que temos uma diferença de 48 bytes entre as duas estruturas. Essa eficácia na capacidade de armazenamento da AABB ainda não a faz uma boa escolha na maioria dos sistemas gráficos, tendo em vista que a precisão e a redução de "alarmes falsos" sobre colisões proporcionadas pelas OBBs reduzem o tempo de processamento em outros segmentos da aplicação como a renderização, apesar do seu cálculo exigir mais esforço computacional.

Testando colisões entre AABBs

O principal intuito de se usar estruturas auxiliares que contenham estruturas mais complicadas é o fato de que existe uma facilitação de teste que realizados de outra forma podem se mostrar muito complexos. As AABBs são uma maneira simples e eficaz de realizar um teste de colisão entre duas malhas de triângulos. Para tanto, são feitos testes para confirmar a interseção de duas caixas. Esse teste é realizado verificando se para cada ponto de referencia de uma dada caixa está contido na outra. Dessa forma, é efetuado um teste simples e eficiente para se testar uma colisão. O código em C++ usado para tal teste é descrito a seguir:

```
/* Detect cube intersection */
bool AABB::detectColision ( AABB &box1, AABB &box2 ) {
        return
                containsPoint ( box1, box2.getPosX1 (), box2.getPosY1 (),
box2.getPosZ1 () ) ||
                containsPoint ( box1, box2.getPosX2 (), box2.getPosY1 (),
box2.getPosZ1 () ) ||
                containsPoint ( box1, box2.getPosX1 (), box2.getPosY2 (),
box2.getPosZ1 () ) ||
                containsPoint ( box1, box2.getPosX1 (), box2.getPosY1 (),
box2.getPosZ2 () ) ||
                containsPoint ( box1, box2.getPosX2 (), box2.getPosY2 (),
box2.getPosZ1 () ) ||
                containsPoint ( box1, box2.getPosX1 (), box2.getPosY2 (),
box2.getPosZ2 () ) ||
                containsPoint ( box1, box2.getPosX2 (), box2.getPosY1 (),
box2.getPosZ2 () ) ||
                containsPoint ( box1, box2.getPosX2 (), box2.getPosY2 (),
box2.getPosZ2 () ) ;
}

bool AABB::containsPoint ( AABB &box, double x, double y, double z ) {
return
        ( x <= box.getCenterX () + box.getDimX () / 2 && x >= box.getCenterX () -
box.getDimX () / 2 ) &&
```

```
            ( y <= box.getCenterY () + box.getDimY () / 2 && y >= box.getCenterY () -
box.getDimY () / 2 ) &&
            ( z <= box.getCenterZ () + box.getDimZ () / 2 && z >= box.getCenterZ () -
box.getDimZ () / 2 ) ;
    }
```

Testando colisões entre OBBs

Realizar testes de colisão entre OBBs é uma tarefa mais complexa que o caso das AABBs. Para realizar essa tarefa, precisamos criar uma árvore [2] que contenha a malha de triângulos e, utilizando informações da malha, subdividi - lá em duas sub malhas. Caso a sub malha contenha só um triângulo, nenhuma OBB é construída e o triangulo é considerado uma folha da árvore. Esses nós podem conter informações para auxiliar os testes de colisão: Uma boa idéia de organização de dados é armazenar um ponteiro para o objeto, uma OBB, ponteiros para os filhos e um identificador do triângulo em questão. Esse tipo de construção também controla o número de triângulos a serem testados na aplicação, com o devido número de modificações, aumentando assim sua eficiência principalmente no que diz respeito ao uso de malhas muito complexas.

Entrando na colisão em si, podemos usar essas árvores para criarmos uma dupla recursão em duas árvores OBB e comparar suas respectivas OBBs para colisões. Efetivamente falando, uma OBB de uma árvore é testada com uma OBB de outra árvore. No caso de interseção, então o filho da segunda OBB é comparada com a OBB corrente da primeira árvore. Com isso, podemos encontrar os pontos de colisão, conforme o pseudocódigo apresentado:

```
bool FindCollision (float dt, ObbTree node0, int depth0, ObbTree node1, int depth1) {
    if ( !TestIntersection(dt,node0,node1) )
    return true;
    if ( HasChildren(node0,depth0) ) {
        if ( !FindCollision(dt,node0.Lchild,depth0-1,node1,depth1) )
        return false;
        if ( !FindCollision(dt,node0.Rchild,depth0-1,node1,depth1) )
        return false;
        if ( HasChildren(node1,depth1) ) {
            if ( !FindCollision(dt,node0,depth0,node1.Lchild,depth1-1) )
            return false;
            if ( !FindCollision(dt,node0,depth0,node1.Rchild,depth1-1) )
            return false;
        }
    return true;
    }
    if ( HasChildren(node1,depth1) ) {
    if ( !FindCollision(dt,node0,depth0,node1.Lchild,depth1-1) )
        return false;
    if ( !FindCollision(dt,node0,depth0,node1.Rchild,depth1-1) )
        return false;
    return true;
    }
return FindIntersection(dt,pkTree1);
}
```

Dessa forma, cobrimos uma das possíveis maneiras de detectar colisões entre OBBs, sendo essa a parte mais complexa de todo o tema abordado. Essa afirmativa decorre das muitas possíveis maneiras de se realizar a tarefa[3].

Conclusões e possíveis trabalhos futuros

Detecção de colisões é um aspecto importante de sistemas gráficos complexos como jogos de computador e programas de simulações variados. Modelos rígidos tratados nesse trabalho podem ser tratados com os algoritmos aqui apresentados de forma eficiente. Para se trabalhar com modelos deformáveis, é necessário modificar a aplicação profundamente, tendo em vista que os modelos são modificados em tempo de execução do programa, demandando assim um grande esforço computacional quando comparado com modelos não deformáveis.

O uso de linguagens de programação (C, C++) e outras ferramentas auxiliares (OpenGL, GSL) permitiram a visualização de tais algoritmos em funcionamento, sendo partes essenciais da pesquisa realizada.

A computação gráfica tem uma base profunda baseada na matemática e parte dela foi explorada nesse projeto. Conforme foi visto aqui e no curso de Elementos Matemáticos e Computação Gráfica, a computação gráfica atual se baseia nos avanços da computação e da matemática, tendo em vista que ambas se completam nesse assunto.

Referências

1 – LEE, Joon Lae. **SVD (Singular Value Decomposition) and Its Applications**. Out. 2006. Disponível em <www.mathnet.or.kr/real/2006/1/1101_lee.ppt>. Acesso em: 20 de outubro, 2007.

2 – H. M. DEITEL & P. J. DEITEL. **C++ Como Programar**. 3° Ed. Porto Alegre: Bookman, 2001. 1098 p.

3 - Eberly, David. Magic Software. **Dynamic Collision Detection using Oriented Bounding.** Disponível em <www.Boxeswww.geometrictools.com/Documentation/**DynamicCollisionDetection**.pdf>. Acesso em: 12 de Janeiro, 2008.