INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SUL-RIO-GRANDENSE - IFSUL, CÂMPUS PASSO FUNDO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Róger Matheus Lasch

# Detecção de Colisões em Simulações Computacionais

Passo Fundo

2023

Róger Matheus Lasch

# Detecção de Colisões em Simulações Computacionais

Projeto de pesquisa submetido como requisito parcial para a aprovação na disciplina de Trabalho de Conclusão I do Curso de Ciência da Computação, do Instituto Federal Sul-Rio-Grandense, Câmpus Passo Fundo.

Orientador: Me. José Antônio Oliveira de Figueiredo

Coorientador: Dr. Jucelino Cortez

Passo Fundo

2023

SUMÁRIO

# 1 TEMA

Simulação Computacional.

## 1.1 Delimitação do tema

Detecção de colisão entre objetos em simulações computacionais.

# 2 PROBLEMA

Estudar e avaliar os mecanismos para detecção de colisão em uma simulação computacional que busca apresentar objetos de forma mais realística possível.

# 3 OBJETIVOS

## 3.1 Objetivo Geral

Explicar como funciona a detecção de colisão em simulações computacionais.

## 3.2 Objetivos Específicos

1. Conhecer as principais etapas de uma simulação física computacional.
2. Avaliar o funcionamento dos algoritmos sort and sweep e octree na fase de detecção de possíveis colisões em uma etapa de simulação.
3. Avaliar o funcionamento de algoritmos para detecção de colisão entre objetos de geometrias específicas.
4. Avaliar o método Impulse para a resolução de colisão entre dois objetos em uma etapa de simulação.

# 4 JUSTIFICATIVA

Segundo Bourg e Bywalec (2013), "a detecção de colisão é um problema de geometria computacional que determina se ocorreu, e onde ocorreu, uma ou mais colisões entre objetos" da simulação. Com base nisto, podemos afirmar que existem diversos algoritmos para detecção de colisão entre objetos de geometrias específicas, e também diferentes técnicas para otimizar esta detecção tornando-a mais eficiente, rápida e robusta.

Ericson (2004) afirma que apesar de hoje em dia os computadores serem extremamente rápidos, a detecção de colisão continua sendo um problema fundamental. Isso ocorre porque jogos e outras simulações são frequentemente construídos em um ambiente 3D, com centenas de milhares de polígonos, exigindo assim, algoritmos e estruturas de dados mais sofisticados para operar, em tempo real, com conjuntos de dados dessa magnitude.

Por ser uma área ainda em expansão, pouco explorada e conhecida, este estudo se justifica pois novos métodos poderão surgir, havendo possibilidades para vários trabalhos de pesquisa e desenvolvimento.

# 5 REFERENCIAL TEÓRICO

Nesta sessão, é apresentada o estudo teórico sobre os principais tópicos envolvendo a detecção de colisão em ambientes simulados.

## 5.1 Simulação Computacional

Segundo Durán (2018) A simulação computacional é uma técnica amplamente utilizada para o estudo de sistemas complexos que por diversos fatores não podem ser facilmente reproduzidos devido a custos elevados, ou pelo tempo necessário para que se realize os experimentos para a obtenção de resultados precisos.

A simulação permite que um pesquisador avalie diversos cenários diferentes para sua pesquisa apenas alterando suas variáveis de controle reduzindo a sim os custos e os riscos de um experimento físico. Conforme Marschner e Shirley (2018), essa técnica pode ser aplicada em diversas áreas como biologia, engenharia, física, química entre outras.

Para construir uma simulação, torna-se necessária a representação matemática ou computacional do comportamento do sistema a ser simulado. Isso inclue mas não se limita a variáveis de controle que podem alterar o ambiente simulado, e os objetos que serão simulados durante algumas etapas, ou por um período de tempo.

A simulação é uma poderosa ferramenta que contribue na descoberta de novas tecnologias, ou no aprimoramento das existentes.

### 5.1.1 Representação Computacional de Formas Geométricas

Na computação, não existe maneira correta ou incorreta de representar a geometria. A forma usada dependerá do nível de detalhamento desejado Pharr, Jakob e Humphreys (2016). É possível fazer esta representação de diversas maneiras, por exemplo: forma vetorial, paramétrica ou de forma poligonal.

Neste estudo será adotada a representação vetorial, devido a sua simplicidade e fácil compreensão. O trabalho será desenvolvido com 3 geometrias principais, que são: esferas, caixas e poliedros.

Para representação dessas formas geométricas, são utilizados pontos de coordenadas cartesianas, definidos em uma estrutura de vetor tridimensional, conforme demonstrado no Código 1

Código 1 – Exemplo de vetor 3d

```
class vector3d{
  float x;
  float y;
  float z;

  vector3d(float x, float y, float z){
    this->x=x;
    this->y=y;
    this->z=z;
  }
};
```

Onde x representa o eixo que vai da esquerda para a direita; y representa o eixo perpendicular ao eixo x; e z é o eixo que indica a profundidade apontando para cima.

Lembrando também que os vetores tem diversas operações relacionadas como soma, subtração, multiplicação, divisão, comprimento ou módulo, produto escalar, produto cruzado e produto triplo escalar que são relevantes para este estudo.

### 5.1.2 Esferas

As esferas são simples de representar computacionalmente Pharr, Jakob e Humphreys (2016), e também incrivelmente fáceis de se computar. Para representarmos uma esfera, tudo o que precisamos é de um ponto que representará o centro da esfera, e um raio, que indica o quanto a esfera se estende em todas as direções a partir do centro. Em Ericson (2004) é possível encontrar uma implementação geral de esferas, e em Bourg e Bywalec (2013) é possível perceber na prática como usar uma bounding sphere.

Código 2 – Representação de esfera

```
class sphere3d{
public:
  vector3d center;
  float radius;

  sphere3d(const vector3d& center, float radius){
    this->center=center;
    this->radius=radius;
  }
};

sphere3d s({0.0f, 0.0f, 0.0f}, 5.0f);
```

Desta forma, temos uma esfera cujo o centro está posicionado exatamente na origem de nosso sistema cartesiano, e que se estende por um raio de 5 em todas as direções.

### 5.1.3 Caixas

As caixas são trivialmente representáveis de diversas formas. Com elas podemos representar formas cuboides facilmente. Ela normalmente é representada por dois vetores, onde um indica o canto inferior esquerdo, e o outro o canto superior direito.

Outras formas populares de se representar caixas são: Ponto central mais meias larguras que se estenderão do centro até as bordas da caixa. Ponto inferior esquerdo, e a extensão de suas arestas no eixo x, y e z.

As três formas fornecem vantagens e desvantagens dependendo do contexto em que forem utilizadas. Em Ericson (2004) é possível encontrar uma explicação mais detalhada de como estas representações funcionam.

O Código 3 apresenta as formas mais comuns para representação de caixas. A classe CaixaMinMax define uma caixa com ponto mínimo e máximo. A classe CaixaMeiaLargura define um objeto com um ponto central e o raio. Já a classe CaixaMedidasCompletas define um objeto com as medidas de lado completas.

Código 3 – Representação de caixa ponto mínimo e máximo

```
class CaixaMinMax{
  vector3d min;
  vector3d max;
};

class CaixaMeiaLargura{
  vector3d center;
  vector3d radius;
};

class CaixaMedidasCompletas{
  vector3d min;
  vector3d measures;
};
```

Todas elas servem para representar a mesma coisa, mas podem ser utilizadas em contextos diferentes dependendo o requisito. A classe CaixaMinMax é melhor aplicada para aabbs porque já tem seus limites bem definidos e pode ser usada mais facilmente para sucessivos testes. Porém tem a desvantagem de precisar fazer uma atualização extra quando precisar ser movimentada.

No entanto, a classe CaixaMeiaLargura tem a vantagem de precisar atualizar apenas o ponto central, porém, quando necessário, terá as operações para computar seus pontos mínimos e máximos. Este tipo de estrutura também é conhecida como esfera de 3 raios.

Em octrees ela é a representação preferida para a representação de nós pois é fácil

de representar e de se computar.

Por fim, a classe CaixaMedidasCompletas lembra muito a criação de uma janela na computação gráfica. Ela tem seu ponto mínimo, e o comprimento de suas arestas até o ponto máximo.

O Código 4 cria três caixas exatamente iguais.

Código 4 – Código de exemplo de criação de 3 caixas

```
1 CaixaMinMax b1({10,10,10}, {20,20,20});
2 CaixaMeiaLargura b2({15,15,15}, {5,5,5});
3 CaixaMedidasCompletas b3({10,10,10}, {10,10,10});
```

Todas as 3 representações terão seus limites entre 10 e 20.

### 5.1.4 Poliedros

Os poliedros nada mais são que uma nuvem de pontos que representam uma geometria específica. Os poliedros certamente são a maneira mais precisa de representar geometria, porém eles apresentam uma dualidade significativa. Segundo Pharr, Jakob e Humphreys (2016), quanto mais detalhado o poliedro, mais caro será para computar colisões, rotações e translações.

Existem dois tipos de poliedros. Poliedros convexos e poliedros côncavos. Poliedros convexos são poliedros que são coplanares ou que possuem suas faces voltadas para fora. Por outro lado, poliedros côncavos possuem pelo menos uma de suas faces voltada para o interior. Neste trabalho não usaremos poliedros pois eles utilizam algoritmos mais sofisticados para operarem sobre nuvens de pontos para detectar colisões. Em Ericson (2004) é discutido com mais detalhes poliedros e seus algoritmos específicos.

## 5.2 Estruturas para delimitação de volumes

Em uma simulação computacional existem diversos objetos e o ambiente precisa testar se alguma colisão ocorreu. Este monitoramento constante acaba tendo um alto custo computacional. Para otimizar esse processo, os simuladores utilizam estruturas chamadas delimitadores de volume, que fazem um encapsulamento da geometria em avaliação.

Existem vários tipos de estruturas delimitadoras de volumes, como esferas, retângulos, e cascas convexas. Devido a simplicidade de implementação, neste trabalho optou-se pela estrutura retangular. As estruturas delimitadoras de volume poderão ser implementadas de duas formas básicas, que são: Aligned Axis Bounding Box (AABB) e Oriented Bounding Box (OBB).

A estrutura AABB, que pode ser implementada como apresentado no Código 5, é uma caixa retangular alinhada aos três eixos do sistema de coordenadas e não pode sofrer

rotações. Caso uma rotação aconteça, o delimitador precisa ser reconstruído, implicando em maior custo computacional. No entanto, segundo Ericson (2004), a verificação de colisão é relativamente mais simples de computar.

Código 5 – Exemplo de AABB

```
class AABB{
public:
  vector3d min;
  vector3d max;
  GeometricShape* shape;
};
```

Código 6 – Exemplo de AABB e Esfera

```
sphere3d* s=new sphere3d({5, 5, 5}, 5);
AABB* aabb=new AABB(s);
```

Como apresentado no Código 6, a bounding box desta esfera será 2 vetores encapsulando completamente em um cubo. Seus vetores respectivamente serão: min=0,0,0 max=10,10,10

Já a estrutura OBB não está presa a nenhum eixo, facilitando a rotação dos objetos contidos nela. A estrutura OBB, apresentada no Código 7, é representada por um ponto central, com meias medidas para os eixos e uma matriz de rotação. Segundo Ericson a estrutura OBBS é computacionalmente custosa e de difícil implementação.

Código 7 – Código de exemplo de OBB

```
class obb{
public:
  vector3d center;
  vector3d measures;
  matrix3x3 orientation;
};
```

Neste trabalho optou-se pela utilização da estrutura AABB, devido à relativa facilidade de implementação em relação a estrutura OBB

### 5.2.1 Corpo Rígido

Segundo Bourg e Bywalec (2013) Um corpo rígido é um conjunto de partículas que permanecem a distâncias fixas umas das outras sem nenhum tipo de translação ou rotação entre elas. Em outras palavras, um corpo rígido não mudará de forma enquanto se move, ou sua deformação é tão insignificante que pode ser desprezada.

Um corpo rígido é frequentemente representado como tendo uma posição, uma orientação e dimensões para representar seu volume. Estes são requisitos mínimos para

a sua definição, porém, dependendo do nível de precisão desejada, pode-se adicionar mais atributos como tensores de inércia, velocidades lineares e angulares, coeficiente de restituição, e o que mais for necessário para que se atinja os objetivos desejados.

Existem 3 classes principais de corpos que podem ser representadas em uma simulação: Corpos rígidos, que não se deformam enquanto se movem, ou caso se deformem, são capazes de voltar a sua forma original;

Corpos macios, que podem deformar enquanto se movem ou por ação de forças ou eventos externos;

Ou corpos estáticos, que nada mais é que uma sub-classificação de um corpo rígido ou um corpo macio, mas com a ausência de massa, ficando a sim, estacionário na posição em que se encontra.

### 5.2.2 Representação de um Corpo Rígido

Para fins de exemplos, neste trabalho será adotada representação de corpo rígido do Código 8.

Código 8 – Exemplo de corpo rígido

```
class RigidBody{
public:
  float mass;
  float restitution;
  vector3d lastPosition;
  vector3d position;
  vector3d velocity;
  vector3d forces;
  AABB* aabb;
};
```

## 5.3 Detecção de Colisão

Segundo Bourg e Bywalec (2013), a detecção de colisão é um problema de geometria computacional que determina se e onde, dois ou mais objetos colidiram. Uma colisão pode ser detectada de diversas maneiras dependendo da natureza dos objetos envolvidos como esferas e caixas.

Para algumas situações, conforme Foley (1996) uma colisão pode ser detectada se a distância entre os dois objetos está abaixo de uma determinada tolerância. Em outros casos, os objetos podem estar se sobrepondo em um ou mais pontos.

Após uma colisão ser detectada, diversos dados precisam ser coletados para o processamento e resolução da colisão, que é feito posteriormente. Os dados coletados dependerão da natureza do simulador. Para um simulador de física, por exemplo, pode ser

necessário o ponto ou pontos de contato onde os objetos colidiram, bem como a velocidade relativa destes objetos no instante da colisão.

Outros dados relevantes para coleta na detecção de colisão são: o vetor normal, que é um vetor unitário que indica a direção em que a colisão aconteceu; e a profundidade de sobreposição ou de penetração dos objetos, que estabelece o limite para que os objetos se toquem e não se sobreponham.

## 5.4 Fases da Detecção de Colisão

A detecção de colisão é dividida em duas áreas principais que são fase ampla e fase estreita.

A fase ampla, segundo Ericson (2004), é responsável por determinar rapidamente pares de objetos que possivelmente estejam colidindo, descartando aqueles objetos que não estão. Nesta fase, os delimitadores AABBS das formas geométricas são testados pois possuem algoritmos simplificados e eficientes para esta tarefa. Quando uma colisão entre dois AABBS é detectada, a sua geometria contida pode não estar colidindo com a geometria do segundo AABB. Então estes dois objetos são agrupados para uma análise mais detalhada em uma fase posterior.

Já a fase estreita, também segundo Ericson (2004), é responsável pelo trabalho mais detalhado e preciso da detecção de colisão. É nesta fase onde os algoritmos e técnicas específicas são aplicadas para determinar se de fato os pares de objetos indicados pela fase anterior estão colidindo. Caso estejam colidindo, informações como vetor normal, ponto de contato e profundidade de penetração devem ser calculados para uso posterior.

## 5.5 Algoritmos para Fase Ampla

Em um ambiente simulado podem existir muitos objetos, alguns próximos outros distantes. Ao selecionar um objeto qualquer deste conjunto de objetos queremos saber com quais outros objetos existe a chance de colisão. Em uma situação desta natureza, a complexidade dos testes normalmente é de $O(n^2)$, onde n é o número de objetos a serem testados. Apesar de ser um problema polinomial, esta complexidade ainda é muito dispendiosa, porque os objetos mais distantes espacialmente não precisariam ser testados.

Para acelerar este processo, segundo Ericson (2004), existem diversas estruturas de dados e técnicas de compartimentação espacial. A lógica destas técnicas é agrupar os objetos que estão próximos uns dos outros e interromper a busca quando o restante dos objetos estão fora de alcance.

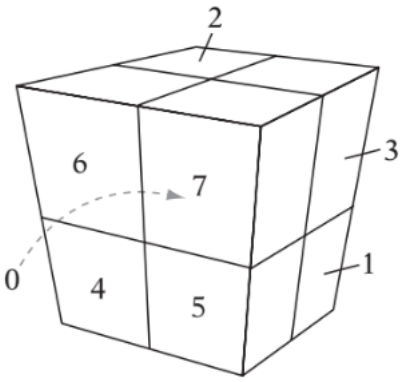
Algumas das estruturas mais populares atualmente são as árvores octrees, grades uniformes e a divisão espacial por hash.

### 5.5.1 Octrees

Uma octree é uma estrutura de dados hierárquica que organiza um espaço tridimensional em células menores recursivamente. Dado um nó raiz, este nó é dividido em 8 filhos, que também podem ser subdivididos em mais 8 filhos até que um critério de parada seja satisfeito. Este critério pode ser uma profundidade máxima para a árvore, ou o tamanho do nó que será dividido atingiu um limite mínimo.

Os objetos indexados pela estrutura serão adicionados no menor nó que seja capaz de conter todo o volume do objeto. Com esta compartimentação espacial, os objetos só precisam verificar a possibilidade de colisão com seus irmãos, e com os nós mais profundos acelerando assim o processo. A Figura 1 mostra a representação de uma árvore octree e seus filhos.

Figura 1 – Figura mostrando a representação de uma octree



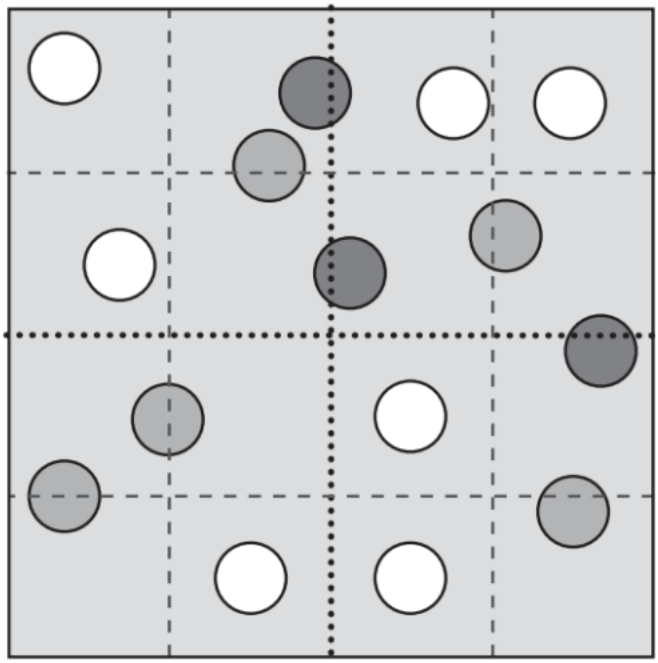

Fonte: (ERICSON, 2004).

Descrição da Imagem: A Figura 1 representa um cubo, tridimensional, em perspectiva (de forma que vemos três faces e um canto). Este cubo (que representa um nó na estrutura octree) é dividido em 8 subcubos - com 1/8 vezes o tamanho do cubo maior. Em outras palavras, o cubo é dividido na metade nos eixos x,y e z.

Na imagem vemos 7 subcubos numerados de 0 até 8. Na parte inferior estão os cubos 0 (que não é visível por estar escondido atrás dos outros nessa perspectiva); Da direita para esquerda estão os cubos 1,5 e 4. Na parte superior estão os cubos 3,7,6 e 2. O cubo 2 está sobre o cubo 0; o cubo 3 está sobre o cubo 1; o cubo 7 está sobre o cubo 7 e, por fim, o cubo 6 está sobre o cubo 4.

A Figura 2 mostra uma árvore quadtree, que é a versão bidimensional de uma árvore octree. Nesta figura é ilustrado como os objetos adicionados ficam indexados nos nós.

Figura 2 – Figura ilustrando uma árvore octree acomodando objetos



Fonte: (ERICSON, 2004).

Descrição da Imagem: A Figura 2: mostra um quadrado subdividido em quadrados menores, com 15 bolinhas distribuídas aleatoriamente, que representam objetos da simulação. Nenhuma bolinha toca outra, portanto não existe colisão entre os objetos da simulação. O quadrado maior (1º nível na quadtree) é então dividido ao meio (verticalmente e horizontalmente), formando 4 subquadrados menores (2º nível na quadtree). Na imagem, esta divisão é representada por linhas pontilhadas. A seguir, uma nova divisão ocorre e cada subquadrado é dividido novamente (verticalmente e horizontalmente) em 4 novos subquadrados (3º nível na quadtree). Na imagem, esta divisão é representada por linhas tracejadas.

Ao final das divisões, cada quadrado foi dividido em 4 subquadrados. As bolinhas dentro do quadrado inicial são agora classificados em função de sua posição estão nestas divisões dos quadrados. Se alguma bolinha foi "cortada" por uma das linhas divisórias, então pertencem ao nível superior, caso contrário pertencem ao nível inferior.

A primeira divisão (linha pontilhada) "corta" 3 bolinhas (cor cinza escuro). Estas bolinhas pertencem ao quadrado maior (1º nível); As linhas tracejadas "cortam" 5 bolinhas (cor cinza claro). Estas bolinhas pertencem aos quadrados da primeira divisão (2º nível); As bolinhas que não estão sendo cortadas por nenhuma linha (cor branco), ficaram exatamente dentro dos quadrados menores (3º nível).

### 5.5.2 Classificar e Varrer

Na detecção de colisão, existe uma categoria de algoritmos denominada de classificação e varredura. Estes algoritmos tem como objetivo determinar grupos de objetos que

estão colidindo uns com os outros. Esta categoria de algoritmos é dividida em duas etapas principais:

Classificação: Classifica os objetos seguindo um critério específico. Este critério pode ser ordenar os objetos ao longo de um determinado eixo.

Varredura: Varre os objetos classificados detectando interseções entre os os objetos que estão na projeção do intervalo especificado.

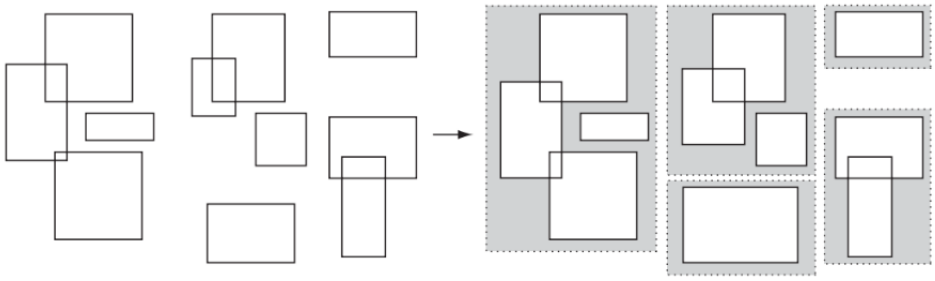
Segundo Ericson (2004) o algoritmo começa ordenando o vetor de entrada por um dos eixos X, Y ou Z., em seguida, o vetor é percorrido, e em cada iteração os objetos são testados quanto a colisão. Caso o limite inferior do Código 5 seja maior do que o limite superior do AABB testado, o laço interno pode parar com segurança pois não existe mais risco de colisão com os próximos objetos. Em seguida, a variância dos eixos são calculados, e o eixo com a maior variância é selecionado para uma execução futura.

Código 9 – Exemplo de ordenar e varrer

```
void SortAndSweepAABBArray(){
qsort(gAABBArray, MAX_OBJECTS, sizeof(AABB *), cmpAABBs);
float s[3]={0.0f, 0.0f, 0.0f}, s2[3]={0.0f, 0.0f, 0.0f},v[3];

for (int i = 0; i < MAX_OBJECTS; i++){
  Point p = 0.5f * (gAABBArray[i]->min + gAABBArray[i]->max);
  for (int c = 0; c < 3; c++){
    s[c] += p[c];
    s2[c] += p[c] * p[c];
  }

for (int j = i + 1; j < MAX_OBJECTS; j++){
  if (gAABBArray[j]->min[gSortAxis] > gAABBArray[i]->max[gSortAxis])
  break;
  if (AABBOverlap(gAABBArray[i], gAABBArray[j]))
  TestCollision(gAABBArray[i], gAABBArray[j]);
  }
}

for (int c = 0; c < 3; c++)
v[c] = s2[c] - s[c] * s[c] / MAX_OBJECTS;
gSortAxis = 0;
if (v[1] > v[0])
gSortAxis = 1;
if (v[2] > v[gSortAxis])
gSortAxis = 2;
}
```

A Figura 3 ilustra um grupo de objetos antes de passar pela fase ampla, e os objetos após a passagem pela fase ampla.

Figura 3 – Ilustração de objetos antes e depois de passarem pela fase ampla



Fonte: Ericson (2004).

Descrição da Imagem: A Figura 3 apresenta o resultado do processamento na fase ampla. A imagem tem duas partes: a esquerda, que mostra os objetos antes do processamento da fase ampla; a direita, que mostra os agrupamentos feitos pelo processamento da fase ampla. Antes do processamento, a imagem mostra 11 objetos retangulares dispostos de forma aleatória, mas que podem ser agrupados por proximidade (alguns estão mais próximos do que outros). Pode-se dizer que os objetos poderiam ser separados em 3 grupos verticais, como colunas.

Na primeira coluna há 4 objetos: 3 são maiores e estão em colisão; 1 menor, está sozinho bem no meio. Na segunda coluna, também há 4 objetos: 2 na parte superior, em colisão; 2 objetos separados na parte de baixo da coluna, um mais próximo desta colisão (meio da coluna) e outro mais distante (bem na parte inferior da coluna). Na terceira coluna há 3 objetos: 1 está sozinho (na parte superior), 2 estão em colisão (parte inferior da coluna).

Na parte direita da imagem estão os mesmos blocos, na mesma organização, mas agora agrupados em função do processamento da fase ampla. Os agrupamentos são representados por uma área sombreada em cinza em torno dos objetos. Os 11 objetos foram agrupados em 5 subgrupos menores. Destes, 3 subgrupos possuem objetos que colidem e 2 possuem objetos isolados. A separação ficou da seguinte forma:

A primeira coluna tornou-se um subgrupo completo com os mesmos 4 objetos. A segunda coluna ficou com um subgrupo com os 3 objetos superiores (2 em colisão e 1 bem próximo). O objeto mais distante (bem na parte inferior da coluna) ficou sozinho em um subgrupo. A terceira ficou com 2 subgrupos: O objeto que estava na parte superior ficou em um subgrupo, também sozinho. Já os objetos da parte inferior formaram um subgrupo com os dois objetos de baixo.

Após a fase ampla estes 11 objetos foram separados em 5 subgrupos. Destes 5 subgrupos, 3 possuem objetos que colidem e 2 possuem objetos isolados (sozinhos)

## 5.6 Algoritmos de Fase Estreita

Após os testes da fase ampla retornarem pares de objetos que possivelmente estejam colidindo, é a vez da fase estreita ser executada.

Nesta fase, os algoritmos mais custosos e complexos computacionalmente são executados para determinar com certeza se os objetos estão em colisão. Caso estejam, pode-

se coletar as informações sobre a colisão como normal, ponto de contato e profundidade de penetração para resolver a colisão mais tarde.

### 5.6.1 Esfera contra Esfera

Para que duas esferas estejam em colisão, a soma de seus raios deve ser menor ou igual a distância de seus centros segundo Ericson (2004). A representação da esfera descrita no Código 2 será utilizada.

Código 10 – Colisão entre esferas

```
bool sphereSphere(sphere3d* s1, sphere3d* s2){
vector3d v=s1->center-s2->center;
float dist=((v.x*v.x)+(v.y*v.y)+(v.z*v.z));
float sqRadius=s1->radius+s2->radius;
sqRadius=sqRadius*sqRadius;

if(dist<=sqRadius){
  return true;
  }
return false;
}
```

Neste exemplo, a soma dos raios foram elevados ao quadrado para realizar o teste contra a distância também ao quadrado entre os centros das esferas. Esta é uma prática recomendada pois o cálculo de raiz quadrada pode ser muito lento para ser realizada como descrito em Ericson (2004) e Bourg e Bywalec (2013).

### 5.6.2 Caixa contra Caixa

Para que duas caixas estejam em colisão, precisa existir sobreposição nos três eixos testados. Neste exemplo, a definição de caixa pela classe CaixaMinMax, do Código 3 será preferida.

Código 11 – Colisão entre caixas

```
bool boxBox(box3d* b1, box3d* b2){
if((b1->min.x>b2->max.x)||(b2->min.x>b1->max.x))
  return false;

if((b1->min.y>b2->max.y)||(b2->min.y>b1->max.y))
  return false;

if((b1->min.z>b2->max.z)||(b2->min.z>b1->max.z))
  return false;
return true;
}
```

Existem outras variações para este mesmo algoritmo descritos com mais detalhes em Ericson (2004).

### 5.6.3 Esfera contra Caixa

O teste de esfera e caixa pode ser realizado em duas etapas principais: Primeiro: encontrar o ponto mais próximo da caixa em relação a esfera. Segundo: Calcular a distância entre o ponto mais próximo e o centro da esfera. Caso a distância seja menor que o raio da esfera, então existe uma colisão.

Código 12 – Colisão entre esfera e caixa

```
bool spherebox(sphere3d *s, box3d *b){
vector3d closestPoint;
for (int i = 0; i < 3; i++){
  float v = s->center[i];
  if (v < b->min[i])
  v = b->min[i];
  if (v > b->max[i])
    v = b->max[i];
  closestPoint[i] = v;
  }

float sqdist=((closestPoint - s->center) * (closestPoint - s->center));
float sqradius = (s->radius * s->radius);

if(sqdist <= sqradius){
  return true;
  }
return false;
}
```

Implementações mais sofisticadas podem ser encontradas em Ericson (2004).

### 5.6.4 Informações sobre a Colisão

Se o objetivo for determinar se os objetos colidem, os testes demonstrados são suficientes. No entanto, se o objetivo for resolver a colisão informações adicionais devem ser coletadas durante o processo de detecção de colisão. O processo de coleta de informações é um problema difícil segundo Ericson (2004). Quais informações coletar depende muito dos objetivos do sistema.

Porém, três informações são fundamentais: Ponto de contato: Retorna um ou mais pontos de contato indicando onde os objetos colidiram. Normal da colisão: Este é um vetor unitário apontando a direção em que os objetos devem ser movidos para que eles parem de colidir. Profundidade de penetração: É o quanto os objetos estão se sobrepondo.

Esta informação é utilizada juntamente com a normal de colisão para calcular o vetor de translação mínima (VTM) ou (MTV) em inglês, necessário para separar os dois objetos.

Cuidados Adicionais devem ser tomados ao se determinar a normal da colisão como por exemplo, se uma esfera colidir contra uma caixa, e seu centro estiver dentro da caixa, o vetor normal será um vetor nulo, o que poderá resultar em comportamentos indesejados na hora de resolver a colisão.

## 5.7 Resolução de Colisão

Após detectar uma colisão entre dois objetos torna-se necessário resolver a mesma. Em simulações que não requerem um realismo muito grande, apenas a correção das posições dos objetos torna-se necessária.

Em simulações mais avançadas, como simulações de física, outros métodos podem ser utilizados como a resolução de colisão por impulso. Neste método, características físicas dos objetos são levados em consideração para que os objetos reajam de forma adequada e mais natural.

Outro cuidado deve-se ser levado em conta nesta etapa. Os objetos em colisão podem ser um objeto estático e um objeto dinâmico, ou dois objetos dinâmicos. A colisão não deve ser resolvida caso os dois objetos em colisão sejam estáticos. Caso um dos objetos seja dinâmico e o outro estático, a resolução da colisão deve atuar apenas no objeto dinâmico. Caso os dois objetos sejam dinâmicos, a resolução deve atuar sobre os dois objetos em direções opostas. O Código 13 exemplifica este processo.

Código 13 – Exemplo de resolução de colisão

```
1  void solveCollision(object* b1, object* b2, vector3d contactPoint,
       vector3d normal, vector3d depth){
2
3  if(b1->mass<=0.0f){
4    float dir=dotProduct(r2->velocity, normal);
5    if(dir>0) return;
6    if(depth>0){
7      vector3d mtv=normal*depth;
8      b2->Translate(mtv);
9    }
10
11   float j=-(b2->velocity*normal) * (b2->restitution+1)*b2->mass;
12   vector3d impulse=(j*info.normal);
13   r1->velocity+=(1 / r1->mass) * impulse;
14   }
15 else{
16 }
17 }
```

Segundo Bourg e Bywalec (2013) este é apenas um exemplo básico e simplificado. Em simulações complexas muitos Outros atributos devem ser levados em consideração como tensores de inércia, velocidade tangencial, velocidade angular e torque.

# 6 METODOLOGIA

A proposta de desenvolvimento deste projeto é um programa gráfico que executará simulações de movimento em ambientes 3d, para analisar os mecanismos de detecção de colisão.

## 6.1 Recursos preliminares do Simulador

O simulador desenvolvido deverá implementar algumas leis básicas da física, para que os objetos simulados reajam de forma mais realista, tornando assim, a simulação mais dinâmica. O programa será organizado em 3 partes principais:

Tela principal: Será onde a simulação será desenhada, mostrando ao usuário o andamento da simulação. A tela principal também terá um menu permitindo o controle de diversos aspectos da simulação.

Configuração do ambiente: Esta tela será responsável por configurar o ambiente de simulação. Propriedades como limites do ambiente, passo de tempo, e se a gravidade deve ou não ser aplicada são exemplos de propriedades configuráveis. Também nesta tela, terá a lista de objetos que poderão ser controlados. Engine de simulação: Esta parte do programa será subdividida em simulação do movimento e detecção de colisão.

Simulação do movimento: Esta etapa é responsável pelos cálculos que modelam o movimento dos objetos presentes no ambiente da simulação, utilizando para isto as leis de Newton. No entanto, a única restrição existente é que os objetos estáticos, que representam obstáculos ou a geografia do espaço, não devem ser movimentados.

Detecção de colisão: A detecção de colisão é a etapa responsável por aplicar algoritmos específicos nos objetos a fim de determinar se existe uma sobreposição, ou se a distância entre 2 objetos é inferior a uma determinada tolerância. Esta etapa também é responsável por resolver a colisão caso exista.

Na tela do simulador, o usuário terá a opção de definir alguns parâmetros do ambiente. A lista a seguir apresenta os parâmetros que se pretende implementar:

- Limites espaciais: definem o espaço que os objetos podem percorrer;
- Definição de obstáculos: inserir/remover obstáculos fixos no espaço da simulação;
- Gravidade: Aplicar força gravitacional na simulação;
- Controle de tempo: Definir o passo de tempo da simulação;

Os objetos simulados poderão, por meio de um painel, ter os seguintes parâmetros configuráveis:

- Massa: Quantida de massa do objeto;
- Posição: Local em que o objeto inicia;
- Velocidade: Velocidade inicial objeto simulado;
- Coeficiente de restituição: Quanto de energia é transferida em uma colisão;
- Forma: Entre esfera; caixa ou cubo;

### 6.1.1 Modelagem preliminar do simulador

Nesta sessão é apresentado a modelagem preliminar da engine de simulação. Os principais componentes são comentados.

A classe descrita em Código 14 representará as coordenadas cartesianas tridimensionais no espaço. Existem três componentes fundamentais:

- eixoX: Representa a distância do ponto em relação a uma origem de referência horizontalmente.
- eixoy: Representa verticalmente a distância do ponto até a origem.
- eixoz: É o eixo perpendicular ao eixo x e eixo y. Utilizado para dar profundidade.

Código 14 – Modelagem prévia para vetor3d

```
class vetor3d{
  eixoX
  eixoY
  eixoZ
  magnitude()
  normalizar()
  inverso()
  produto_escalar()
  produto_vetorial()
  produto_triplo_escalar()
};
```

O Código 14 também tem diversas operações associadas como soma, subtração, multiplicação e divisão. Também possui métodos específicos como normalização, magnitude, inverso e seus produtos como produto vetorial e escalar.

A classe de forma geométrica apresentada no Código 15 é uma base para as próximas implementações de esferas e caixas. Nela são declados os padrões que devem ser seguidos pelas classes filhas.

Código 15 – Modelagem prévia para interface de forma geométrica

```
class FormaGeometrica{
  tipo
  recuperar_centroide()
  calcular_suporte()
  transladar(vetor v)
  escalar(s)
  rotacionar(orientacao)
  converter_string()
};
```

A classe Esfera apresentada Código 16 tem 2 atributos principais que são:

- PontoCentral: É um vetor que representa as coordenadas centrais da esfera.
- Raio: Representa o quanto a esfera se estende em todas as direções a partir do centro.

Código 16 – Modelagem prévia para esfera 3d e caixa

```
class Esfera estende FormaGeometrica{
  pontoCentral
  raio
};

class Caixa estende FormaGeometrica{
  minimo
  arestas
};
```

A classe Caixa apresentada em Código 16 tem duas propriedades que também são importantes:

- Mínimo: É um vetor que representa o canto inferior esquerdo da caixa.
- Arestas: É o quanto as arestas se estendem nos 3 eixos a partir do canto inferior esquerdo.

A classe apresentada no Código 17 é responsável por representar um volume delimitador para envolver completamente uma forma geométrica, seja ela esfera, caixa ou poliedro.

A classe possui 3 atributos principais:

- min: É um vetor representando o canto inferior esquerdo do aabb.
- max: É um vetor representando o canto superior direito do aabb.
- FormaGeometrica: É a forma geométrica que o aabb está contendo.
- Métodos: Os métodos da classe são redimensionar e trasladar.

Código 17 – Modelagem prévia para AABB

```
class AABB{
  min
  max
  FormaGeometrica
  translacao(vetor v)
  escala(s)
  escala(vetor v)
  computar_volume_delimitador()
};
```

O Código 18 da classe CollisionInfo é responsável por conter informações sobre a colisão que foi detectada como ponto de colisão, normal, profundidade, e os objetos envolvidos na colisão.

Código 18 – Modelagem prévia para coletor de informações

```
class Colisao_info{
  ponto_de_colisao
  normal_da_colisao
  profundidade
  objeto_1
  objeto_2
};
```

O Código 19 da classe de colisão irá conter os métodos específicos de colisão entre as formas geométricas específicas como esferas x esferas, caixas x caixas e poliedros x poliedros.

Código 19 – Modelagem prévia para classe estática para colisões

```
class Colisao{
  esfera_esfera(s1, s2, info)
  esfera_caixa(s, c, info)
  caixa_caixa(c1, c2, info)
  esfera_poliedro(s, p, info)
  caixa_poliedro(c, p, info)
  poliedro_poliedro(p1, p2, info)
};
```

A interface do Código 20 define alguns atributos básicos que será necessário ao modelar o objeto. Se precisarmos de atributos mais específicos, ou se quisermos expandir nossa simulação futuramente, basta sub-classificarmos esta interface e realizar as mudanças necessárias.

Código 20 – Modelagem prévia para interface de corpo rígido

```
class Corpo_rigido_interface{
  massa
  restituicao
  nome
  aabb
  posicao
  };
```

A interface FaseAmpla do Código 21 representa o nosso algoritmo de broadphase. Como existem diversos algoritmos, optou-se de criar uma interface e realizar a implementação separadamente.

O método scan recebe um vetor de corpos rígidos para verificar, e também uma lista onde será armazenado informações sobre uma possível colisão.

Código 21 – Modelagem prévia para fase ampla e estreita

```
class FaseAmpla{
  escanear(lista_de_objetos, lista_de_possiveis_colisoes)
};

class FaseEstreita{
  detectar_colisoes(lista_de_possiveis_colisoes, lista_de_colisoes)
};
```

A classe FaseEstreita do Código 21 implementa o algoritmo de fase estreita da mesma forma que a interface de fase ampla. Este algoritmo recebe uma lista de possíveis colisões e sua tarefa é aplicar os testes de detecção de colisão entre geometrias específicas e avaliar se de fato, o par testado está colidindo.

O Código 22 irá resolver as colisões detectadas. Ele age separando os corpos se existir sobreposição, e aplicando impulso.

Código 22 – Modelagem prévia para solucionador de colisões

```
class Solucionador_de_colisao{
  resolver(lista_de_colisoes)
  resolver_par(objeto_1, objeto_2, info)
};
```

O Código 23 implementa o algoritmo que fará os objetos se movimentarem na simulação. Para isso serão utilizadas equações das leis de Newton.

Código 23 – Modelagem prévia para integrador numérico

```
class Integrador_numerico{
  mover(lista_de_objetos, tempo)
  mover_objeto(objeto, tempo)
};
```

O Código 24 da octree é utilizada na fase ampla, mas como ela é relativamente cara de se construir, ela é utilizada em objetos estáticos como obstáculos que representam a geografia do ambiente simulado.

A classe tem 2 métodos de fase ampla. O primeiro, recebe uma lista de todos objetos que queremos que sejam testados, e a saída é uma lista com todas as colisões detectadas. O segundo método, recebe um objeto e uma lista de saída de colisões detectadas.

Código 24 – Modelagem prévia para octree

```
class Octree{
  raiz
  adicionar_objeto(objeto)
  remover_objeto(objeto)
  fase_ampla(lista_de_objetos, lista_de_possiveis_colisoes)
  fase_ampla(objeto, lista_de_possiveis_colisoes)
}
```

O Código 25 é responsável pela simulação do ambiente. Para isso recebe e manipula todos os objetos instanciados pelas classes anteriormente descritas. A classe permite adicionar e remover objetos, e também definir os algoritmos utilizados para realizar as tarefas. O método de atualização será chamado sempre que um passo na simulação for avançado. Ele primeiramente movimentará os objetos, e então fará a fase ampla, fase estreita, e resolverá as colisões.

Código 25 – Modelagem prévia para ambiente de simulação

```
class Ambiente_de_simulacao estende AABB{
  tempo_atual
  gravidade
  lista_de_objetos
  fase_ampla
  fase_estreita
  solucionador_de_colisoes
  integrador_numerico
  octree
  adicionar_objeto(objeto)
  remover_objeto(objeto)
  passo_de_simulacao(tempo)
};
```

### 6.1.2 Requisitos do Projeto

A listagem a seguir apresenta os recursos previstos para o desenvolvimento do projeto. Todos os recursos são de código aberto, ou possuem licenças de uso não comercial.

- Compilador compatível: com c++20 ou superior;
- Compilador compatível: com C2017 ou posterior;
- Framework wxwidgets: para a criação das telas e controles como listas, botões e demais controles;
- Biblioteca opengl: para a renderização gráfica;
- Biblioteca wxglade: para construir a interface gráfica do usuário (GUI);
- Biblioteca fmod: para indicações sonoras;
- Biblioteca bass: para indicações sonoras;

## 6.2 Cronograma

A listagem a seguir, apresenta uma distribuição estimada das tarefas a serem realizadas de forma quinzenal.

1. Agosto:
    - Primeira quinzena:
        - Configuração do ambiente de desenvolvimento e modelagem das classes do simulador;
    - Segunda quinzena;
        - Revisão e testes das classes desenvolvidas;
        - Registros e documentação parcial para composição do artigo final;
2. Setembro:
    - Primeira quinzena:
        - Desenvolvimento da janela principal do aplicativo e seus diálogos;
    - Segunda quinzena:
        - Desenvolvimento da animação gráfica dos objetos;
        - Registro sobre o andamento do projeto para a composição do artigo final;
3. Outubro:

- Primeira quinzena:
  - Refinamento de todo o projeto desenvolvido e realizar a integração dos componentes;
- Segunda quinzena:
  - Finalização do protótipo para testes e correções de bugs;
  - Registro do progresso para a composição do artigo final;

4. Novembro:
   - Primeira quinzena:
     - Testes do protótipo;
     - Escrita do artigo final;
   - Segunda quinzena:
     - Término do artigo final;
5. Dezembro:
   - Primeira quinzena:
     - Finalização e entrega do artigo final;
     - Defesa do Trabalho de conclusão de curso;

# REFERÊNCIAS

BOURG, D. M.; BYWALEC, B. *Physics for Game Developers: Science, math, and code for realistic effects*. [S.l.]: "O'Reilly Media, Inc.", 2013. Citado 6 vezes nas páginas 3, 5, 8, 9, 15 e 18.

DURÁN, J. M. *Computer simulations in science and engineering: Concepts-Practices-Perspectives*. [S.l.]: Springer, 2018. Citado na página 4.

ERICSON, C. *Real-time collision detection*. [S.l.]: Crc Press, 2004. Citado 12 vezes nas páginas 3, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15 e 16.

FOLEY, J. D. *Computer graphics: principles and practice*. [S.l.]: Addison-Wesley Professional, 1996. v. 12110. Citado na página 9.

MARSCHNER, S.; SHIRLEY, P. *Fundamentals of computer graphics*. [S.l.]: CRC Press, 2018. Citado na página 4.

PHARR, M.; JAKOB, W.; HUMPHREYS, G. *Physically based rendering: From theory to implementation*. [S.l.]: Morgan Kaufmann, 2016. Citado 3 vezes nas páginas 4, 5 e 7.