Física - Detecção de Colisões

Introdução

Um dos aspectos mais importantes de um sistema de física realista é a capacidade de detectar e resolver as colisões que ocorrem entre os objetos na simulação - um jogo de sinuca não é muito divertido se não pudermos arremessar as bolas acertando a bola branca com a nossa deixa!

Os dois tutoriais a seguir mostrarão os princípios básicos por trás do processo de resolução de colisões, o que aumentará muito o realismo de sua simulação física. Neste tutorial, veremos como usar algoritmos de detecção de colisão para determinar se dois objetos estão colidindo de alguma forma - ou seja, se suas formas estão se tocando ou se sobrepondo devido ao seu movimento ao longo do tempo. Detectar isso e determinar a quantidade de sobreposição e as posições no espaço em que os objetos se chocam são os primeiros passos para criar uma resolução de colisão realista. O próximo tutorial tratará da resposta à colisão. Como a interseção de objetos entre si não é fisicamente possível, temos que manter o realismo da simulação resolvendo essas colisões, separando-as para que não se sobreponham e determinando se alguma energia cinética deve ser transferida de um objeto para outro - como quando acertamos aquela bola branca, fazendo-a voar pela mesa.

Volumes de colisão

Vimos no tutorial anterior sobre raycasting que há um conjunto de volumes de colisão padrão usados para representar a maioria dos objetos habilitados pela física em nossas simulações de jogo. Assim como acontece com os raios, o processo de detecção de colisões entre cada combinação dessas formas é um pouco diferente, então teremos que definir uma série de funções para lidar com cada colisão potencial que pode ocorrer em cada quadro de simulação.
Para tanto, neste tutorial investigaremos as colisões mais comuns que precisamos detectar: esfera versus esfera, AABB versus AABB e AABB versus esfera.

Detecção de colisão

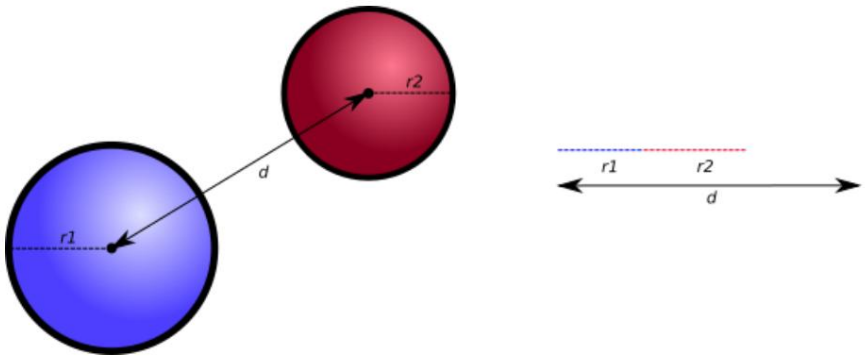
É claro que existem volumes de colisão mais complexos do que caixas simples, e veremos em um tutorial posterior como aprimorar a detecção de colisão para suportar qualquer formato convexo. Não descarte a utilidade de formas mais simples, como caixas e esferas! Os cálculos para determinar colisões entre eles são geralmente bastante rápidos, permitindo que muitos deles sejam calculados por quadro. Além disso, às vezes simplesmente não precisamos da fidelidade adicional que uma detecção de colisão mais complexa traria. Considere o seguinte exemplo de um personagem que pega um powerup andando sobre ele:

Realmente importa se descobrirmos exatamente qual polígono do powerup foi tocado por qual polígono dos pés do personagem? Provavelmente não, então podemos apenas modelar essa interação como um simples teste de interseção caixa/caixa.

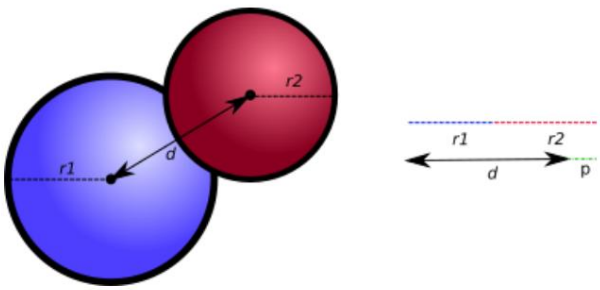
Além de saber se dois objetos estão colidindo, também precisamos saber detalhes sobre exatamente como eles estão colidindo. Precisamos saber aproximadamente quais pedaços dos objetos estão se cruzando, para que possam ser separados, e aproximadamente quanto eles estão se cruzando (às vezes conhecido como distância de penetração), para calcular as forças que os separarão novamente. Para ajudar com isso, as funções de detecção de colisão geralmente não retornam apenas uma resposta verdadeira ou falsa, mas também algumas informações adicionais de colisão - em nossos exemplos determinaremos um ponto de colisão (ou ponto de contato) e uma distância de interseção e colisão normal.

Colisões Esfera/Esfera

Determinar se duas esferas se sobrepõem é bastante fácil. Se a distância entre duas esferas for maior que a soma dos seus raios, elas não podem estar colidindo:



A partir disso, é fácil dizer que para quaisquer duas esferas delimitadoras cuja distância d entre elas seja menor que a soma de seus raios, devem estar colidindo:

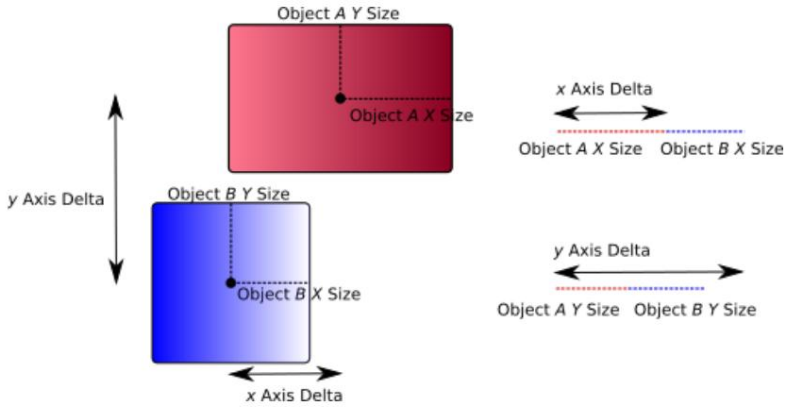


Determinar a colisão normal n é igualmente fácil - é apenas o vetor de direção entre os objetos. É bom estabelecer uma direção normal de colisão consistente, então afirmaremos que se os objetos A e B estiverem colidindo, a normal aponta para longe de A e em direção a B (a direção normal exata dependerá da forma da colisão e da natureza da interseção, como veremos com os outros tipos de colisão). A distância de intersecção p é simplesmente a diferença entre os comprimentos das esferas e a soma dos seus raios. Por último, devemos considerar um ponto de colisão - o ponto no espaço a partir do qual estamos assumindo que quaisquer forças necessárias para resolver a colisão serão aplicadas. Por uma série de razões, geralmente armazenamos dois pontos de colisão para um par de objetos em colisão, cada um relativo a uma das origens do objeto. Para uma esfera, podemos determinar o ponto de colisão de cada objeto viajando para frente ou para trás ao longo da colisão normal pelo raio de cada esfera.

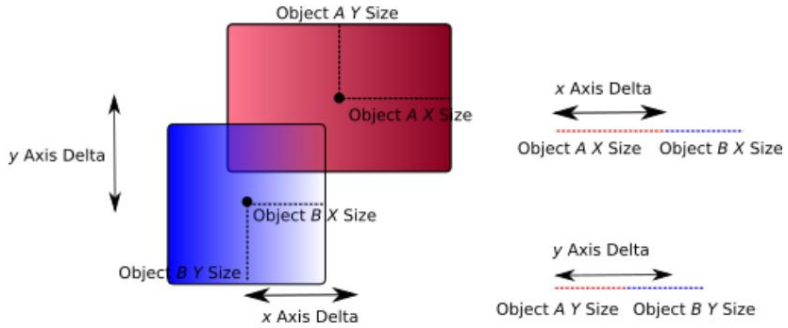
Colisões AABB / AABB Calcular colisões

entre caixas delimitadoras alinhadas aos eixos é um pouco mais complicado do que com esferas delimitadoras. Para determinar se eles estão colidindo, nós os tratamos de forma bastante semelhante às esferas - calculamos a soma das dimensões da caixa delimitadora em cada eixo e, em seguida, determinamos se a diferença absoluta na posição em cada eixo é menor que a soma da caixa delimitadora. Se todos os 3 eixos forem menores que seus

soma da caixa delimitadora, então as caixas devem estar colidindo. Aqui estão alguns exemplos para ver isso em ação:

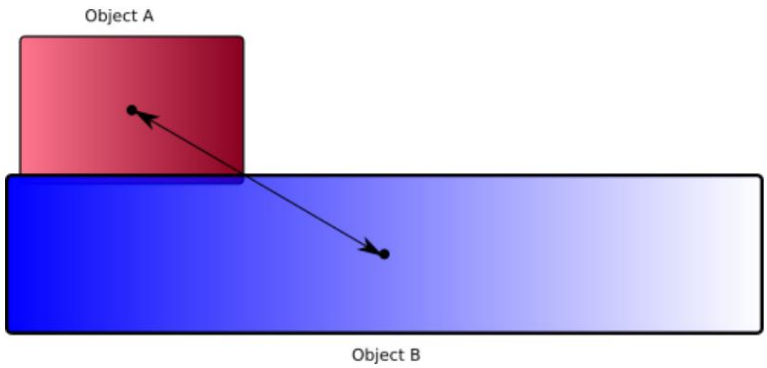


Neste primeiro caso, podemos ver que a soma dos tamanhos das caixas do eixo x é menor que a diferença de posição no eixo x dos dois objetos, indicando que talvez haja uma colisão, aqui. No entanto, a diferença de posição do eixo y é maior que os tamanhos do eixo y da caixa, portanto, na verdade, as caixas não estão colidindo. Agora, para o caso em que os objetos estão colidindo:



Agora a soma dos tamanhos AABB em cada eixo é maior que a posição relativa entre os dois objetos, indicando que os objetos devem estar em contato. Embora esses exemplos mostrem apenas 2 dimensões, a mesma regra também vale para 3D: a diferença de posição em todos os eixos deve ser menor que a soma dos tamanhos das caixas nesse eixo, ou os volumes não colidirão.

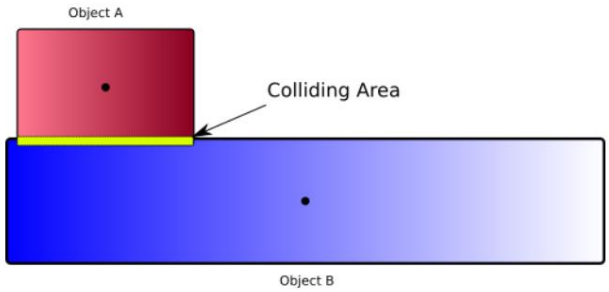
Embora este teste nos diga se as caixas estão colidindo ou não, ele realmente não nos dá as outras peças do quebra-cabeça: a normal de colisão, a distância de intersecção ou os pontos de intersecção. Para obtê-los, precisamos trabalhar um pouco mais. Você pode inicialmente pensar que o vetor de direção entre as caixas seria adequado para uma colisão normal, como acontece com esferas delimitadoras, mas considere o seguinte exemplo:



A caixa a deveria ser empurrada ligeiramente para cima para resolver sua colisão com a caixa b, mas o vetor de direção entre suas origens na verdade aponta para a esquerda, então a caixa a seria empurrada ligeiramente para a esquerda, onde colidirá novamente no próximo quadro, se a simulação tiver a gravidade puxando-a de volta para b - ela terminaria em um ciclo de deslocamento constante para a esquerda, ganhando energia aparentemente do nada.

Para determinar uma colisão normal para dois AABBs, precisamos pensar na distância de penetração em cada eixo individualmente e descobrir qual eixo se sobrepõe menos. Por que escolher o eixo com menor interseção entre os objetos? Observe novamente o diagrama acima - podemos determinar visualmente que os dois objetos estão se cruzando no eixo y (portanto, o objeto A deveria realmente ser empurrado para cima e o objeto B empurrado para baixo para 'consertar' essa colisão), mas seus volumes estão na verdade se cruzando muito mais no eixo x (o objeto A está de fato completamente sobreposto pelo objeto B neste eixo). À medida que percorremos cada eixo para ver se os objetos se cruzam nesse eixo, podemos acompanhar quanto - se as formas forem determinadas para se sobrepor, o eixo com a menor penetração nos dá a colisão normal, e a penetração em esse eixo nos fornece nossa distância total de penetração.

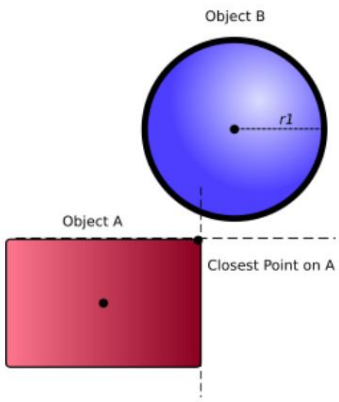
A última coisa que precisamos é definir os nossos pontos de colisão, para sabermos onde aplicar as forças nos nossos objetos para os afastar. Pode-se ver na imagem de exemplo acima que não existe um único ponto no espaço para algumas colisões AABB, ao contrário das colisões esfera/esfera. Um tutorial posterior abordará exatamente como lidar com casos onde há uma área de colisão. O caso de uma colisão AABB nos dá mais informações sobre por que calculamos dois pontos de colisão em nossos algoritmos de detecção de colisão, em vez de apenas um único ponto no mundo. Considere o exemplo das duas caixas colidindo uma com a outra:



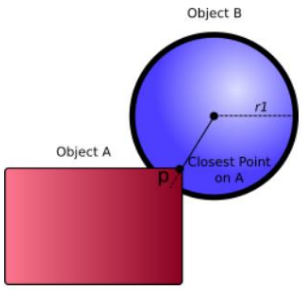
Podemos ver que, assumindo que temos o eixo y como normal de colisão, seria natural assumir que poderíamos escolher qualquer ponto dentro da área de sobreposição mostrada. Entretanto, mais adiante na série de tutoriais veremos como adicionar torque aos objetos e fazê-los girar. Se aplicarmos uma força em qualquer ponto ao longo da área de colisão, então os nossos dois AABBs deverão realmente começar a rodar. Isso é ruim para os AABBs, pois devido à sua propriedade de alinhamento de eixo, qualquer rotação do objeto não alteraria o volume ocupado pelo colisor. Portanto, para AABBs, assumiremos que o ponto de colisão de cada objeto está em sua origem local - quando olharmos para a resolução de colisão, veremos como isso evita que nossos AABBs façam a coisa errada, com a colisão normal e a distância de penetração ainda nos permitindo separar nossos objetos para manter a consistência da simulação.

Colisões Caixa/Esfera

Testar colisões entre caixas e esferas é praticamente o mesmo que testar interseções de caixas, pois encontramos o ponto mais próximo da caixa da esfera, usando a operação de fixação, limitando a posição da esfera para estar entre o intervalo (posição da caixa - caixa tamanho, posição bos + tamanho) em cada eixo. Se esse ponto for maior que o raio do centro da esfera, então os objetos não podem estar colidindo!
O diagrama abaixo deve deixar isso mais claro:

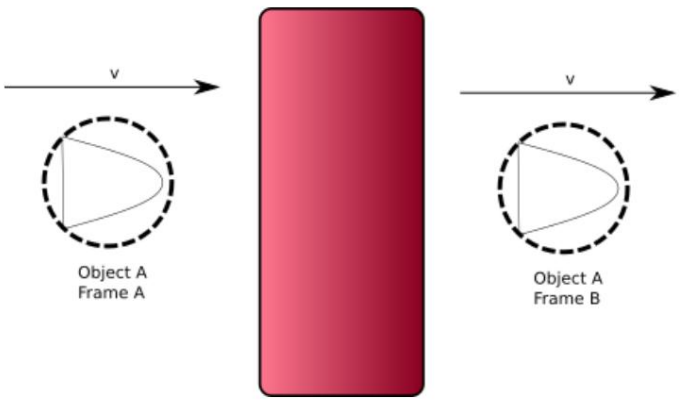


Isso nos permite saber se as formas estão colidindo, mas e as informações do ponto de colisão? A normal de colisão é o vetor de direção entre a origem da esfera e o ponto mais próximo da esfera. O ponto de colisão da esfera está então em um ponto r unidades ao longo desta colisão normal, enquanto para o AABB, precisaríamos escolher um ponto diretamente ao longo de um eixo, assim como acontece com as colisões AABB.
O eixo a ser escolhido seria baseado na colisão normal - escolheríamos qualquer eixo na normal que tivesse a maior magnitude. Finalmente, a distância de penetração pode ser calculada calculando a distância da posição da esfera até o ponto mais próximo e subtraindo o raio da esfera:



Considerações sobre detecção de colisão

Em nossas simulações, atualizamos as posições dos objetos por meio de uma série de etapas discretas por quadro; calculando objetos que se cruzam e separando-os à medida que avançamos. No entanto, há um problema potencial com isso. Considere o seguinte exemplo de uma bala, movendo-se a uma velocidade de 10 m/s, com uma esfera delimitadora como seu volume de colisão:

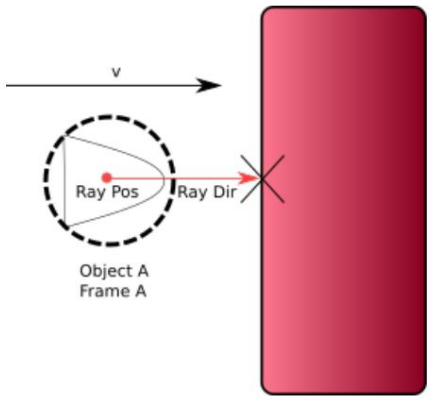


No quadro A, a bala estava na frente da parede, e no quadro B a bala estava atrás da parede - Este efeito é conhecido como tunelamento e é causado porque a bala viaja tão rápido que nunca houve um quadro onde ela se cruzasse a parede, então nenhuma colisão poderia ser calculada! Para certos tipos de cenário, isso é um prejuízo bastante sério para a precisão da nossa simulação física, então vamos investigar algumas maneiras de resolvê-lo (ou evitá-lo completamente!).

Raycasting

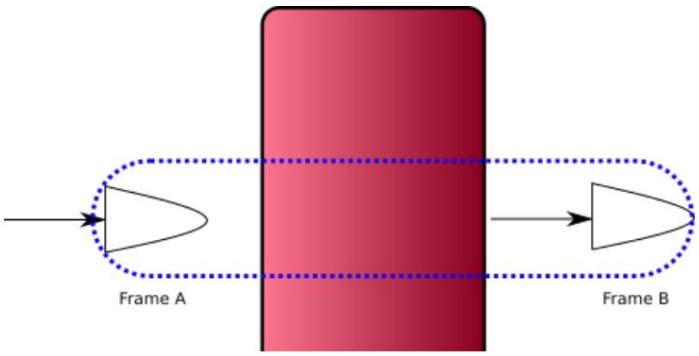
A maneira mais fácil de evitar esse problema, especificamente para muitas interações baseadas na física em nossos jogos, é não tratá-las como objetos físicos! Muitos jogos, como jogos de tiro em primeira pessoa e jogos RTS, não modelam as balas como corpos rígidos, mas simplesmente determinam em que direção a bala irá e, em seguida, executam um raycast para ver o que será atingido nessa direção. Armas que realizam um raycast para detecção de colisão em vez de dinâmica de corpo rígido são frequentemente conhecidas como armas hitscan - na vida real as balas viajam em um arco conforme a gravidade e a resistência do vento as afetam, mas os jogos geralmente são mais divertidos do que simulação realista, então um projéteis instantâneos e em linha reta geralmente funcionam bem.

Também podemos estender esse conceito um pouco mais. Para algo que sabemos que viajará em linha reta rapidamente (talvez o jogador tenha pegado um lançador de foguetes em vez de uma metralhadora?), poderíamos, em vez disso, lançar raios do objeto, na direção em que ele está viajando - se o objeto mais próximo estiver em a uma distância menor que a velocidade do objeto, podemos assumir que ele irá atingi-lo e marcar o objeto para colisão no próximo quadro no ponto p:

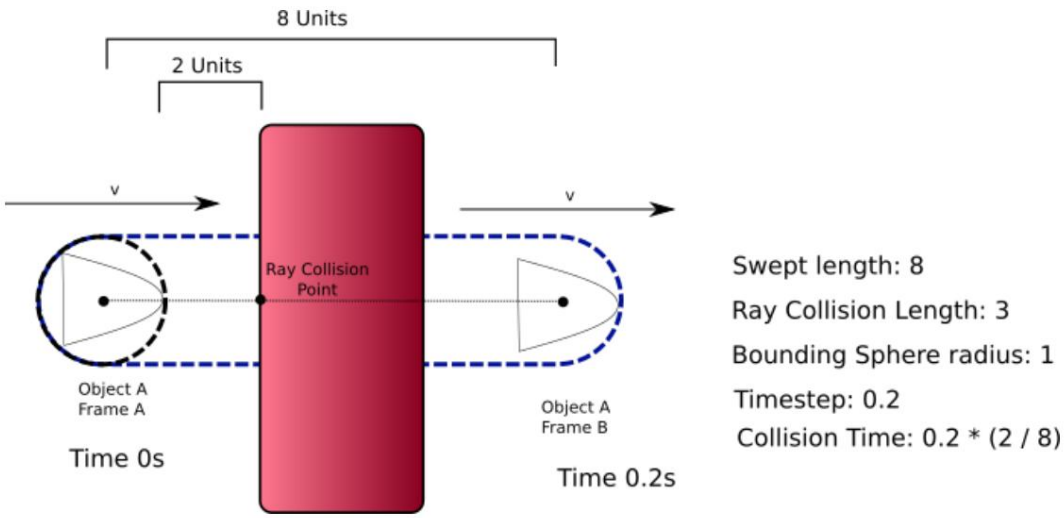


Volumes delimitadores varridos

Porém, apenas realizar um raycast muitas vezes não é suficiente - o raio é infinitamente fino, mas estamos tentando determinar as colisões entre formas e volume. Devemos, portanto, considerar como fazer alterações no próprio volume real de colisões ao tentar detectar colisões em alta velocidade. Considere o exemplo anterior da bala atravessando a parede. Nos dois quadros A e B, o volume de colisão da bala não foi grande o suficiente para cruzar com o da parede, mas na realidade sabemos que devem ter colidido. Para resolver isso, poderíamos 'esticar' o volume delimitador de um objeto à medida que ele se move, de modo que ele encapsule seu ponto no tempo A e no tempo B, assim:

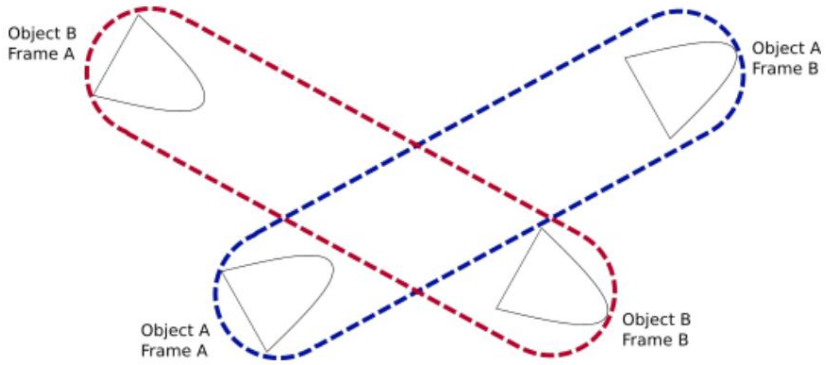


Isso é conhecido como volume delimitado por varredura. Se detectarmos então uma colisão com o volume varrido, então, em algum ponto entre o tempo do quadro A e B, ocorreu uma colisão. Podemos determinar aproximadamente ao usar o ponto de colisão no objeto colidido (neste caso, a parede), usando um raycast:

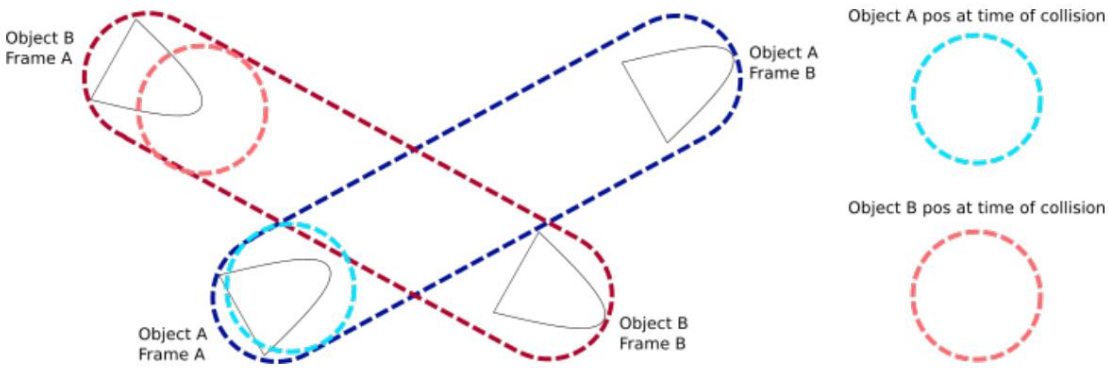


Conhecemos a posição do objeto antes e depois da colisão, portanto podemos formar um raio que começa no ponto “antes” e aponta na direção do ponto “depois”. Também sabemos a distância entre essas posições e também a distância entre a origem do raio e o ponto de intersecção,

a razão entre os dois, multiplicada por dt, nos permitirá saber até que ponto da atualização física ocorreu a colisão. Por que precisaríamos do tempo aproximado de colisão? Não seria apenas mover nossa bala para o ponto de colisão e então resolver a colisão normalmente? Para uma colisão contra um objeto estático, tudo bem, mas considere este exemplo:

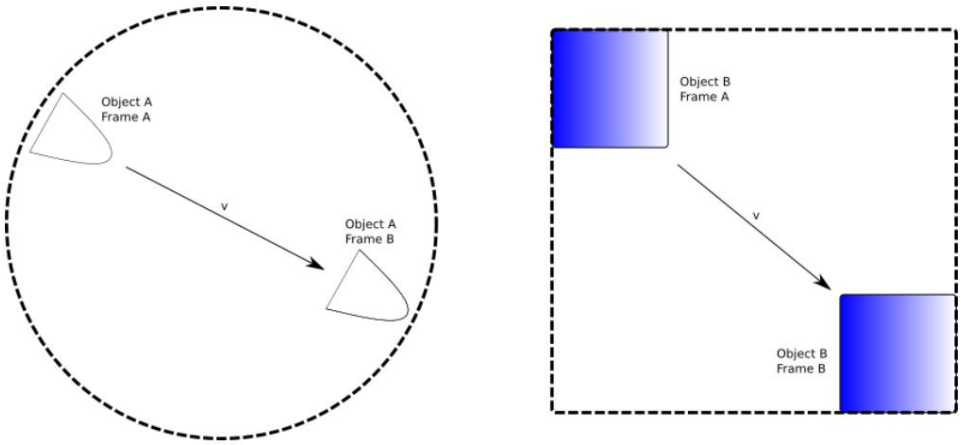


Agora temos dois objetos de alta velocidade! Se varrermos ambos os volumes, parece que haveria uma colisão, mas será mesmo? Para resolver isso, precisamos determinar quando entre o quadro A e o quadro B o objeto A colidiu com o volume varrido de B e, a partir desse momento, determinar onde o objeto B estava e, em seguida, realizar um teste adicional de detecção de colisão nessas novas posições, usando os volumes de colisão 'verdadeiros' do objeto (em nosso exemplo de marcador, estamos assumindo que esta é uma esfera delimitadora).

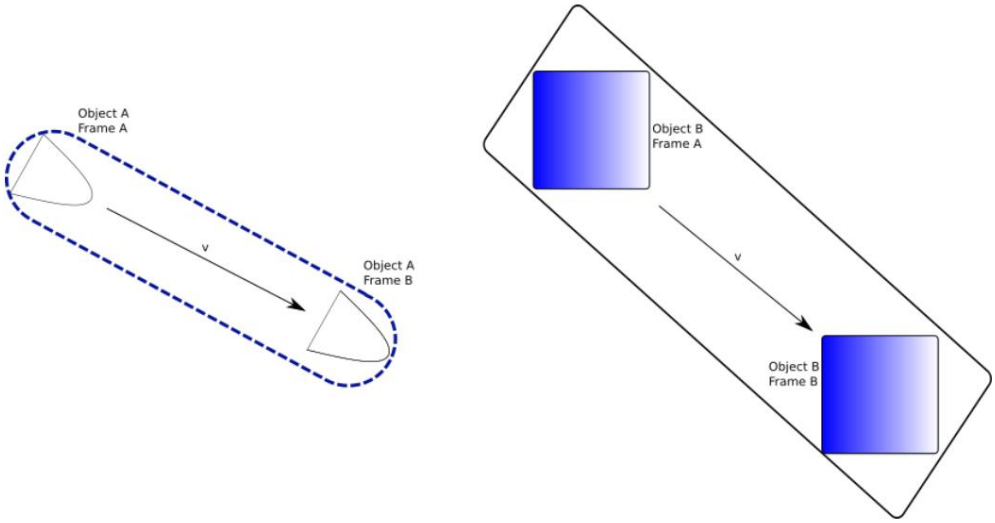


Isso deve ser suficiente para determinar com precisão razoável se duas formas colidem ou não durante um período de tempo - lembre-se, um sistema de física será executado muitas vezes várias vezes por quadro, e nosso objetivo é rodar talvez a 60FPS, o que significa que cada iteração do nosso sistema de física pode ter apenas 2 ou 3 milissegundos de duração, tornando a maioria dos volumes varridos bastante pequenos.

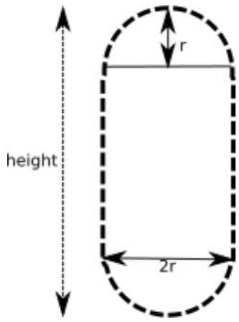
Uma nota final sobre volumes delimitados por varredura: o método usado para esticar o volume é muito importante para a precisão da simulação. Você pode pensar primeiro que para AABBs e esferas, apenas estendê-las para que agora abranjam o ponto final é suficiente, mas dê uma olhada nestes exemplos:



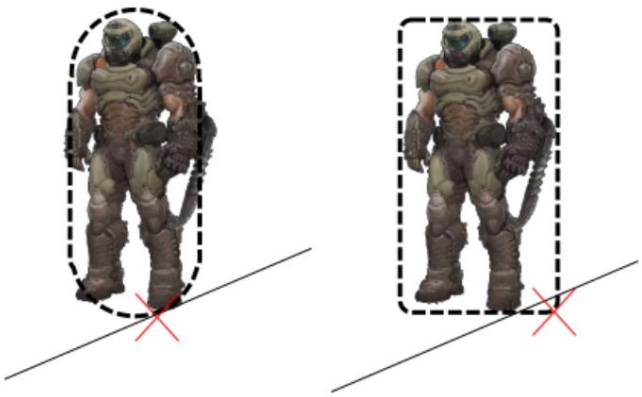
Essa é uma esfera muito grande! Embora certamente seja expandido para cobrir o ponto final, também é esticado para ocupar espaço que o objeto nunca preencherá durante seu movimento entre o quadro A e o quadro B. Isso pode gerar falsas colisões ou, pelo menos, causar testes de colisão adicionais que ocupam a computação do tempo de quadro. Em vez disso, devemos ser mais inteligentes com nossos volumes varridos - um AABB torna-se um OOBB varrido e uma esfera torna-se uma cápsula varrida:



Uma cápsula pode ser pensada como um cilindro com meia esfera em cada extremidade e geralmente é definida como uma altura e um raio, da seguinte forma:



As cápsulas são úteis além de apenas grandes volumes. Eles são frequentemente usados como o modelo básico de colisão de um personagem do jogador, já que a borda arredondada fornece um ajuste mais preciso contra a cabeça do personagem em comparação com uma caixa, e a parte inferior arredondada pode ser usada como uma representação mais precisa de onde o personagem pode se cruzar com rampas e degraus:



Detectando colisões em uma simulação

Agora conhecemos as fórmulas para determinar se várias formas simples estão se cruzando umas com as outras. outro, e quais dados usaremos posteriormente para resolver essas colisões. Mas atualmente não sabemos realmente como determinar quando usar esses métodos de detecção de colisão. Como é possível que objetos se movam ao redor do mundo, é portanto possível que qualquer objeto possa estar colidindo com qualquer outro objeto em o mundo em qualquer momento. Portanto, no código, para detectar nossas colisões e resolvê-las, podemos fazer algo assim:

```
1 para cada objeto x no mundo do jogo {
        para cada objeto y no mundo do jogo {
            if( IsColliding (x, y)) {
2               ResolverColisão (x, y)
            }
        }
3 4 5 6 7 }
```

Parece bastante simples, mas há um problema - é lento! Há um conjunto aninhado de loops for, cada um dos quais tem que passar por todos os objetos do mundo; dito de outra forma, devemos fazer operações N2 para detectar todas as nossas colisões. Isso rapidamente resulta em muitos testes de detecção de colisão! Imagina você estávamos fazendo um jogo de corrida onde 100 carros competiam para viajar pelo país; mesmo ignorando qualquer colisões com outros carros que possam estar na estrada, são 10.000 testes de detecção de colisão que devem ser executado em cada quadro.

No exemplo de código acima, testaremos se o objeto x está colidindo com o objeto y, E se o objeto y está colidindo com o objeto x - no que diz respeito às nossas interações físicas, esse é o mesmo teste, então realmente devemos evitar repeti-lo. Ainda mais estupidamente, nos casos em que x == y, estaríamos testando se um objeto está colidindo consigo mesmo! Podemos ajustar nosso loop for para algo um pouco mais sensato, assim:

```
1 para int x = 0; x <últimoObjeto; ++x{
2       para int y = x +1; y <últimoObjeto; ++ e) {
3           if( IsColliding (x, y)) {
4               ResolverColisão (x, y)
5           }
6       }
7}
```

Neste caso, nosso loop for interno irá pular qualquer combinação de teste que já tenha sido feita (procurando nos ints criados pelos loops, eles agora nunca gerarão o par de objetos (6,5) ou (6,6), mas ainda irão formulário (5,6), por exemplo). O loop em si ainda é caracterizado por O(N2) como o número de potenciais as colisões ainda aumentam, mas economizamos algum tempo e evitamos algumas possíveis dores de cabeça que podem ocorrer se a resolução de colisão empurrasse objetos várias vezes em um quadro e, portanto, potencialmente adicionasse também muita energia em nosso sistema físico.

Veremos em um tutorial posterior que podemos usar propriedades sobre nosso conhecimento do mundo para pular testes de colisão entre objetos que não podem estar colidindo, mas por enquanto vamos apenas lidar com o maneira 'lenta'.

Código do Tutorial

Para mostrar a detecção e resposta a colisões, modificaremos mais uma vez a cena de teste que criamos.
vem se acumulando. Além de poder clicar em objetos para empurrá-los, também poderemos
receber algum feedback sobre quando esses objetos colidiram. Ao pressionar o botão de gravidade, também
ser capaz de ver os objetos sendo puxados para baixo e quicando no chão, em vez de apenas passar por eles.

Alterações no namespace CollisionDetection

Para detectar colisões entre nossas formas, precisamos de algumas funções novas! Até agora fomos apresentados
para AABBs e esferas, então precisaremos de pelo menos 3 novas funções (para lidar com colisões AABB vs AABB,
colisões esfera vs esfera e colisões AABB vs esfera). Além destes, vamos aprofundar
dividir os testes AABB, para que obtenhamos a resposta de colisão verdadeiro/falso de uma função, mas o real
dados de colisão de outro - isso é algo que nos ajudará em um tutorial posterior. Finalmente nós
precisa de uma função que receba um par de GameObjects e chame a função de interseção correta
aplicar. Também precisamos de uma estrutura para armazenar as informações da colisão, para que possamos resolver a colisão
corretamente no próximo tutorial.

Esses métodos para realizar todas essas operações foram declarados no namespace CollisionDetec-tion, mas como no
tutorial de raycasting, eles estão atualmente vazios - preencheremos cada
para ver como a teoria delineada anteriormente se traduz em uma implementação prática. Nestes métodos,
preencheremos uma estrutura que ainda não encontramos, CollisionInfo, que se parece com isto:

```
1   estrutura ContactPoint {
2       Vetor3 localA ; // onde ocorreu a colisão...
3       Vetor3 localB ; // no quadro de cada objeto !
4       Vetor3 normal;
5       flutuador          penetração ;
6   };
7   estrutura CollisionInfo {
8       GameObject *a;
        GameObject *b;
9 10
11      Ponto de contato;
12
13      void AddContactPoint ( const Vector3 e localA const Vector3 e normal      ,
14          const Vector3 e ponto localB. ponto   ,                                  ,   flutuar p ){
15          localA. ponto localB.          =localA;
16          ponto normal.                  =localB;
17          penetração = p ;               =normal;
18
19      }
20  };
```

Adições de namespace CollisionDetection

À primeira vista, poderá parecer estranho que estejamos a dividir o conceito de ponto de contacto individual
(com suas posições associadas, normal e quantidade de penetração), a partir do conceito geral de colisão
entre um par de objetos, mas isso é bastante comum em motores de física por permitir a criação de
o que é conhecido como variedade de colisão, que representa uma superfície de interseção completa, em vez de apenas uma
ponto único.

A primeira função que implementaremos é ObjectIntersection; Atualmente é uma função definida que faz
nada exceto return false, então precisaremos preenchê-lo com alguma lógica real. O propósito disto
A função é pegar dois de nossos GameObjects e determinar qual das funções de detecção de colisão usar. Há uma
variedade de maneiras pelas quais isso pode ser implementado (funções virtuais, dupla
envio, ponteiros de função e assim por diante), mas para tornar óbvio o que está acontecendo, faremos uma abordagem direta
abordagem e uso de instruções if. Começaremos o método obtendo ponteiros para os volumes de colisão

de nossos objetos (linhas 3 e 4), e se uma forma não tiver volume, retorne falso imediatamente, como eles não podem colidir com nada (linha 7). Se os objetos puderem colidir, preenchemos o início do nosso CollisionInfo com os dois objetos e obter as transformações do objeto (linhas 13 e 14 - não estritamente necessário fazer isso, mas reduz chamadas repetidas para os mesmos métodos).

```
1  bool CollisionDetection :: ObjectIntersection (
2          GameObject *a , GameObject * b const , Informações de colisão e informações de colisão) {
3      CollisionVolume * volA = a - > GetBoundingVolume ();
4      const CollisionVolume * volB = b -> GetBoundingVolume ();
5
6      if (! volA || ! volB ) {
7          retorna falso ;
8      }
9
10     colisãoInfo. uma = uma;
11     colisãoInfo. b = b;
12
13     const Transform & transformA=a -> GetConstTransform();
14     const Transform & transformB=b -> GetConstTransform();
```

Função CollisionDetection::ObjectIntersection

Em seguida, precisamos enviar nossos objetos e a estrutura CollisionInfo para a função correta para detectar a possível colisão. Para fazer isso, vamos fazer algumas operações bit a bit no tipos de objetos. Este é um enum (VolumeType, definido em BoundingVolume.h). Como cada entrada no VolumeType enum é uma potência de dois (portanto, apenas um bit é definido), se fizermos um OR bit a bit de ambos os objetos tipos, obtemos uma máscara de bits do tipo de colisão. Se após o OR bit a bit o tipo ainda for igual exatamente para AABB (valor 1) ou Esfera (valor 2), isso significa que ambas as formas devem ser iguais tipo e, portanto, sua ordem realmente não importa. Na linha 15 realizamos o OR bit a bit (com o | símbolo) e, em seguida, use ifs para retornar o resultado de uma interseção AABB ou de uma interseção de esfera nos objetos. Em cada um desses casos, precisamos converter o volume de colisão para o tipo correto:

```
15     VolumeType pairType =( VolumeType )(( int ) volA - > tipo |( int ) volB - > tipo );
16
17     if (pairType == VolumeType :: AABB) {
18         retornar AABBInterseção (( AABBVolume &)* volA , transformarA ,
19                 ( AABBVolume &)* volB colisãoInfo, transformarB ,
20     }
21
22     if (pairType == VolumeType :: Esfera) {
23         return SphereIntersection (( SphereVolume &)* volA , transformarA ,
24                 ( EsferaVolume &)* volB , transformarB , colisãoInfo );
25     }
```

Função CollisionDetection::ObjectIntersection

Mas o que fazer se os objetos testados não forem do mesmo tipo? Poderíamos fazer algumas mudanças quando criamos a variável pairType para diferenciar entre casos em que o objeto b é uma esfera versus quando a é uma esfera e testar isso, mas por enquanto vamos apenas lidar com os dois casos que poderíamos obter distante:

```
26      if (volA -> type == VolumeType :: AABB &&
27            volB -> type == VolumeType :: Esfera ) {
28          retornar AABBSphereIntersection (( AABBVolume &)* volA                    ,   transformarA      ,
29                      ( SphereVolume &)* volB colisionInfo ); transformarB      ,
30      }
31      if (volA -> type == VolumeType :: Esfera &&
32            volB -> type == VolumeType :: AABB ) {
33          colisãoInfo. uma =b;
34          colisãoInfo. b = uma;
35          return AABBSphereIntersection (( AABBVolume &)* volB ( SphereVolume &)* volA   ,   transformarB      ,
36                      colisionInfo );                    ,   transformarA      ,
37      }
38
39      retorna falso ;
40}
```

Função CollisionDetection::ObjectIntersection

Observe que talvez precisemos trocar os objetos de nossa variável CollisionInfo, para corresponder à ordem dos parâmetros que a função espera - AABBSphereIntersection sempre leva em consideração o Objeto de volume AABB primeiro, então nós os invertemos na estrutura CollsionInfo para refletir isso (linhas 33 e 34).

Implementando detecção de colisão Esfera-Esfera

Com a seleção de uma função de detecção de colisão concluída, podemos começar a implementar o atual funções de detecção. O mais fácil de seguir são as colisões esfera-esfera, então começaremos por aí. Encontre o função SphereIntersection atualmente vazia no namespace CollisionDetection e adicione no seguinte código:

```
1 bool CollisionDetection :: SphereIntersection (
2 const SphereVolume e volumeA const Transform & worldTransformA 3 const SphereVolume e volumeB const      ,
Transform & worldTransformB 4                    ,                                                          ,
        Informações de colisão e informações de colisão) {
5
6       raios flutuantes            = volumeA . GetRadius() + volumeB. ObterRaio();
7       Vector3 delta = worldTransformB . ObterPosiçãoMundo() -
8                                   mundoTransformaA . GetWorldPosition();
9
10      float deltaComprimento = delta. Comprimento ();
11
12      if (deltaComprimento < raios ) {
13          penetração flutuante            = ( raios - deltaComprimento );
14          Vetor3 normal                   =delta. Normalizado ();
15          Vetor3 localA                   = normal * volumeA . ObterRaio();
16          Vetor3 localB                   = - normal * volumeB . ObterRaio();
17
18          colisãoInfo. AddContactPoint(localA return true ;// estamos      , localB      , normal      , penetração );
19          colidindo!
20      }
21      retorna falso ;
22}
```

Função CollisionDetection::SphereIntersection

Isso corresponde muito bem à teoria descrita anteriormente. Obtemos a soma dos raios dos dois objetos (linha 6) e, em seguida, calcule a distância entre as posições dos dois objetos (linhas 7 e 10). Se a distância entre eles é menor que a soma de seus raios, eles devem estar se cruzando (linha 12), e assim podemos calcular a distância de penetração (a soma dos raios, menos a distância entre os objetos),

a normal de colisão (o vetor de direção entre os dois objetos) e os pontos de colisão - neste caso estejamos viajando ao longo da colisão normal pelo raio do objeto A, ou para trás ao longo da colisão normal pelo raio do objeto B, para calcular os pontos de colisão relativos aos respectivos centros do objeto. Se a distância for maior que a soma dos raios, não podemos estar colidindo e, portanto, retornaremos falso (linha 21).

Implementando detecção de colisão AABB-Sphere

Esfera-esfera é muito fácil, mas examiná-la nos permitiu ver como a estrutura CollisionInfo deve ser preenchido. Agora podemos tornar as coisas um pouco mais avançadas e observar a detecção de colisão da esfera AABB, usando a função AABBSphereIntersection atualmente vazia. Como descrito anteriormente, o núcleo da intersecção da esfera AABB é encontrar o ponto na caixa mais próximo da localização da esfera. Podemos fazer isso simplesmente fixando a posição relativa da esfera, para limitá-la ao tamanho da caixa - por qualquer eixo onde a posição for maior que o tamanho da caixa, nós o 'travamos' no tamanho da caixa. Podemos fazer isso em código obtendo as dimensões da caixa (linha 5) e a posição relativa da esfera em relação à caixa (linha 7), e então formando uma nova posição no espaço, closePointOnBox (linha 10). Para fixar um valor em um intervalo, podemos usar a função Clamp dentro do namespace Maths, que recebe um valor a, junto com um mínimo e máximo e retorna um a modificado que é limitado ao intervalo passado (por exemplo Clamp(10,3,12) tentará limitar o valor 10 entre 3 e 12 e, portanto, retornará 10, como está dentro do intervalo, mas Clamp(20,3,12) retornará 12, pois 20 está além do valor máximo permitido).

```
1  bool CollisionDetection :: AABBSphereIntersection (
2  transformação const e worldTransformA , const AABBVolume & volumeA ,
3          const SphereVolume & volumeB CollisionInfo , transformação const e worldTransformB ,
4          & CollisionInfo ) {
5          Vector3 boxSize = volumeA . GetHalfDimensions();
6
7          Vetor3 delta = worldTransformB . ObterPosiçãoMundo() -
8                                          mundoTransformaA . GetWorldPosition();
9
10         Vector3 mais próximoPointOnBox = Matemática :: Grampo (delta , - tamanho da caixa , tamanho da caixa);
```

Função CollisionDetection::AABBSphereIntersection

Assim que tivermos o ponto mais próximo da esfera na caixa, podemos determinar a que distância a esfera está do este ponto, subtraindo este ponto da posição relativa da esfera (linha 11). Se este ponto estiver a uma distância menor que o raio da esfera (linha 14), então ela deve estar colidindo e podemos preencher nosso estrutura de informações de colisão. O ponto de colisão para nossa esfera (linha 19) ficará então em unidades de raio para trás ao longo da normal de colisão (lembre-se, nossas normais estão apontando para o objeto B), enquanto nossa AABB será, como de costume, assumido como estando colidindo em sua posição relativa (linha 18). A distância de penetração será o raio da esfera menos a distância da esfera ao ponto mais próximo da caixa (linha 16).

```
11         Vector3 localPoint = delta -maispróximoPointOnBox ;
12         distância de flutuação =pontolocal. Comprimento ();
13
14         if (distância < volumeB . GetRadius ()) {// sim , estamos colidindo!
15                 Vector3 colisãoNormal = localPoint . Normalizado ();
16                 float =(volumeB . GetRadius()-distância);
                           penetração
17
18                 Vetor3 localA = Vetor3();
19                 Vetor3 localB = -colisãoNormal * volumeB . ObterRaio();
20
21                 colisãoInfo. AddContactPoint(localA, penetração); , localB ,
22                         colisãoNormal
23                 retornar verdadeiro ;
24         }
25         retorna falso ;
26 }
```

Função CollisionDetection::AABBSphereIntersection

Implementando detecção de colisão AABB-AABB

A detecção de colisão AABB-AABB foi dividida em duas funções - uma para nos dizer se os objetos estão colidindo e um para preencher as informações de colisão. O mais fácil de entender é o testando, então começaremos implementando-a dentro da função AABBTest:

```
1  bool CollisionDetection :: AABBTest (
2          const Vector3 e posA, const Vector3    const Vetor3 e posB,
3          e halfSizeA                                    ,    const Vetor3 e halfSizeB) {
4          Delta do vetor3                          = posB - posA;
5          Vector3 totalSize = halfSizeA + halfSizeB ;
6
7          if (abs (delta. x) <tamanho total. x &&
8                  abs(delta.y) <tamanhototal. sim &&
9                  abs(delta.z) <tamanhototal. z) {
10                retornar verdadeiro ;
11         }
12         retorna falso ;
13 }
```

Função CollisionDetection::AABBTest

A lógica segue a descrição anterior da intersecção AABB - estamos procurando ver se a distância entre os objetos em cada eixo for menor que a soma dos tamanhos das caixas nesse eixo, e somente se isso é verdadeiro para todos os eixos se considerarmos os objetos que se cruzam. Na linha 4 calculamos a distância em cada eixo, e na linha 5 calculamos a soma dos tamanhos das caixas e, em seguida, na linha 7 calculamos se as distâncias são menores que os tamanhos das caixas. Precisamos usar a função abs para obter o valor absoluto de a posição relativa, já que a caixa A pode estar à esquerda da caixa B ou à direita - não importa, tudo o que nos importa é se a diferença de posição é maior que as dimensões da caixa.

Após a função AABBTest, você também poderá encontrar uma função AABBIntersection vazia. Começaremos a adicionar algum código para que ele realmente faça alguma coisa:

```
1  bool CollisionDetection :: AABBIntersection (
2  const AABBVolume e volumeA 3 const AABBVolume ,    const Transformação e worldTransformA const Transformação e    ,
   e volumeB                                      ,    worldTransformB                                                 ,
4          Informações de colisão e informações de colisão) {
5
6          Vetor3 boxAPos = worldTransformA . GetWorldPosition();
7          Vetor3 boxBPos = worldTransformB . GetWorldPosition();
8
9          Vector3 boxASize = volumeA . GetHalfDimensions();
10         Vector3 caixaBSize = volumeB . GetHalfDimensions();
11
12         sobreposição de bool = AABBTest (boxAPos              ,  caixaBPos  ,  caixaASize    ,  boxBSize );
```

Função CollisionDetection::AABBIntersection

É um começo bastante simples - extraímos as posições e tamanhos dos AABBs e os inserimos no Função AABBTest para determinar se eles estão sobrepostos. Se eles estiverem sobrepostos, as coisas ficam um pouco mais interessante, como demonstra o próximo trecho de código a ser adicionado a esta nova função:

```
13        if (sobreposição) {
14            static const Vector3 faces [6] =
15            {
16                Vetor3 (-1 , 0 , 0) , Vetor3 (1 , 0 , 0) ,
17                Vetor3 ( 0 , -1 , 0) , Vetor3 ( 0 , 1 , 0) ,
18                Vetor3 ( 0 , 0 , -1) , Vetor3 ( 0 , 0 , 1) ,
19            };
20
21            Vetor3 maxA = boxAPos + boxASize ;
22            Vetor3 minA = boxAPos - boxASize ;
23
24            Vetor3 maxB = boxBPos + boxBSize ;
25            Vetor3 minB = boxBPos - boxBSize ;
26
27            distâncias de flutuação [6] =
28            {
29                ( maxB . x - minA . x ) ,// distância da caixa 'b' à 'esquerda' de 'a'.
30                ( maxA . x - minB . x ) ,// distância da caixa 'b' à 'direita' de 'a'.
31                ( maxB . y - minA . y ) ,// distância da caixa 'b' ao 'fundo' de 'a'.
32                ( maxA . y - minB . y ) ,// distância da caixa 'b' ao 'topo' de 'a'.
33                ( maxB . z - minA . z ) ,// distância da caixa 'b' até 'far' de 'a'.
34                (maxA . z - minB . z ) // distância da caixa 'b' até 'perto' de 'a'.
35            };
36            penetração flutuante = FLT_MAX ;
37            Vetor3 melhorEixo;
38
39            para (int i = 0; i < 6; i ++)
40            {
41                if (distâncias [i] <penetração) {
42                    penetração = distâncias [i];
43                    melhor eixo = faces [eu];
44                }
45            }
46            colisãoInfo. AdicionarContactPoint (Vetor3() , Vetor3 () ,
47                bestAxis , penetração );
48            retornar verdadeiro ;
49        }
50        retorna falso ;
51    }
```

Função CollisionDetection::AABBIntersection

Conforme a teoria delineada anteriormente, precisamos do eixo de penetração mínima entre os dois objetos. Para determinar isso e ajudar a determinar a colisão normal, há muito o que fazer. A partir de um posição mundial da caixa, podemos determinar a posição mínima e máxima para cada eixo por adicionando ou subtraindo o tamanho da caixa. Nas linhas 21 a 25 calculamos isso e usamos para calcular o quantidade de sobreposição em cada eixo - a extensão máxima do objeto b, menos a extensão mínima do objeto a em cada eixo nos dá a quantidade de sobreposição (em outras palavras, estamos vendo o quão longe na caixa a, a posição máxima da caixa b se estende). Em seguida, iteramos por essas extensões de penetração (linha 39), e descobrir qual é o menor valor, armazenando a penetração, e o eixo (das faces estáticas array, que é organizado para corresponder ao array de distâncias) toda vez que encontramos um valor menor - isso é por isso que o float de penetração começa com o valor máximo que um float pode conter, em vez de começar em 0.

Depois de percorrermos todos os valores de penetração possíveis e encontrarmos o melhor, podemos então comece a construir as informações de colisão usando a penetração calculada e a variável bestAxis para o colisão normal. Como estamos verificando AABBs e não queremos que eles torçam mais tarde, definimos seus a colisão relativa aponta para a origem (linha 46) - você verá por que isso é importante em um tutorial posterior!

Mudanças na classe PhysicsSystem

Quase todo o código complexo foi adicionado no namespace CollisionDetection, portanto não há muito pendência. Primeiro, precisamos criar uma maneira de determinar se estão ocorrendo colisões em cada quadro, conforme descrito anteriormente. Para fazer isso, a classe PhysicsSystem possui um método BasicCollisionDetection, mas mais uma vez foi deixado vazio. Adicione o seguinte código:

```
1  void PhysicsSystem :: BasicCollisionDetection () {
2      std :: vector < GameObject * > :: const_iterator primeiro ;
3      std :: vector < GameObject * > :: const_iterator last ;
4      mundo dos jogos . GetObjectIterators (primeiro , durar );
5
6      for ( auto i = primeiro; i! = último; ++ i) {
7          if ((* i ) -> GetPhysicsObject () == nullptr ) {
8              continuar ;
9          }
10         for ( auto j = i +1; j! = último; ++ j) {
11             if ((* j ) -> GetPhysicsObject () == nullptr ) {
12                 continuar ;
13             }
14             CollisionDetection :: Informações de CollisionInfo ;
15             if (CollisionDetection :: ObjectIntersection (* i info )) { , *j , <<
16                 std :: cout << “Colisão entre " (* i ) -> GetNome ()
17                             << " e " << (* j ) -> GetName() << std :: endl ;
18                 informações. framesLeft = numCollisionFrames;
19                 todasColisões. inserir (informações);
20             }
21         }
22     }
23 }
```

PhysicsSystem::BasicCollisionDetectionMétodo

Não é tão diferente de como fizemos o raycasting, exceto que agora, em vez de iterar todos os objetos uma vez (por meio dos iteradores que obtemos da classe GameWorld nas linhas 2-4), precisamos repassar o objetos duas vezes. O importante para acertar aqui é a linha 10 - estamos iniciando o loop for interno em um elemento após o loop for externo, para que não resolvamos colisões idênticas várias vezes em um quadro. Como podemos inserir objetos no mundo do jogo que não possuem objetos físicos, também preciso ignorá-los (um pequeno arbusto ou flor pode ter uma representação gráfica, mas não física representação, por exemplo). Por enquanto, vamos apenas mostrar que ocorreu uma colisão no console, mas em breve adicionaremos algum código para realmente resolver a colisão neste método, também. A instrução if na linha 15 chama nosso novo método VolumeIntersection, que retornará verdadeiro se ocorreu uma colisão e, em caso afirmativo, preencherá a estrutura de informações com os dados apropriados para uso posterior.

Há uma última coisa que fazemos, que é inserir a colisão detectada com sucesso em um STL::List, o que nos permitirá acompanhar os objetos que estão colidindo - mais tarde poderemos precisar usar essas informações em vários quadros.

Como exemplo de por que podemos querer manter uma lista de objetos em colisão, veremos outra função atualmente vazia, UpdateCollisionList. Isso é chamado pelo PhysicsSystem a cada frame e deve conter o seguinte código:

```
void PhysicsSystem :: UpdateCollisionList() {
	for (std :: set < CollisionDetection :: CollisionInfo >:: iterador i =
		todasColisões. começar (); eu != todasColisões . fim (); ) {
		if ((* i). framesLeft == numCollisionFrames ) {
			i - >a - >OnCollisionBegin (i - > b );
			i ->b -> OnCollisionBegin (i -> a );
		}
		(* eu ). quadrosEsquerda = (* i ). quadrosEsquerda - 1;
		if ((* i). framesLeft < 0) {
			i - >a - >OnCollisionEnd (i - > b );
			i - >b - >OnCollisionEnd (i - > a );
			i = todasColisões. apagar (eu);
		}
		outro {
			++eu;
		}
	}
}
```

Método PhysicsSystem::UpdateCollisionList

O que isso está fazendo? Jogos sem qualquer feedback do sistema físico sobre quais colisões ocorreram ocorreram não são muito emocionantes - precisamos saber quando um foguete atingiu um jogador para reduzir sua saúde, ou quando eles colocarem o cubo companheiro na fornalha do Portal. Então nesta função, estamos vamos percorrer a lista de colisões que preenchemos no método BasicCollisionDetection, e se eles são novos na lista deste quadro, chame uma função virtual nos GameObjects contidos dentro da estrutura CollisionInfo - por padrão, isso não fará nada, mas permite subclasses do GameObject para implementar a lógica específica do jogo, substituindo o método OnCollisionBegin. Às vezes, podemos até precisar saber quando um objeto parou de colidir com alguma coisa (talvez o jogador entrou em alguma lava e deve perder saúde até sair), então neste caso podemos detecte-o e chame outra função virtual OnCollisionEnd. Caso contrário, reduzimos um contador em cada par de colisão e, se necessário, atualizar o iterador ou remover o par da lista - mais tarde veremos por que precisamos manter objetos por vários quadros. Cada vez que o mesmo par de objetos colide, esse contador é redefinido (reinserindo-o no método BasicCollisionDetection), portanto, somente se um par de objetos não colidem entre si há algum tempo, o método OnCollisionEnd será chamado.

Isso é tudo por enquanto! Se usarmos o ponteiro do mouse para juntar objetos como fizemos em tutoriais anteriores, os objetos ainda se sobreporão (não estamos resolvendo a colisão, apenas detectando-a), mas deveríamos ser capazes de ver no console que uma colisão está ocorrendo, permitindo-nos testar se nosso esferas e caixas retornam o estado de colisão correto.

Conclusão

A detecção de colisão é uma parte importante de qualquer mecanismo de física, pois sem ela não podemos determinar como os objetos devem interagir uns com os outros - os objetos cairão no chão devido à gravidade, e powerups serão impossíveis de pegar! Ser capaz de detectar com eficiência que colisões estão ocorrendo entre as formas em nosso mundo é, portanto, extremamente importante e, portanto, o tutorial mostrou o suficiente para começarmos a construir toda a detecção de colisão de que precisamos. No próximo tutorial, daremos uma veja como resolver as colisões que detectamos, permitindo que nossos objetos saltem pelo mundo, e venha descansar no chão.

Trabalho adicional

1) Embora o texto do tutorial tenha discutido cápsulas e como elas podem ser úteis, ainda não vimos qualquer código para implementá-los. Tente criar uma classe CapsuleCollider que herde de Collision-Volume e que use um novo enum VolumeType para identificá-lo. Lembre-se que precisamos de uma altura

variável e um raio para representar nossa forma - tratar as extremidades como esferas e o meio como um AABB é um bom ponto de partida, mas você também pode querer pensar em projetar a posição de um volume de colisão em um plano.

2) Se você adicionar cápsulas, talvez queira pensar em adicionar a capacidade de varrer um volume delimitador - experimente com esferas por enquanto e teste disparando esferas em alta velocidade em direção a uma parede muito fina; você deve ser capaz de criar um cenário onde as esferas às vezes saltarão, a menos que esferas varridas sejam usadas.