Física - Resposta a Colisões

Introdução

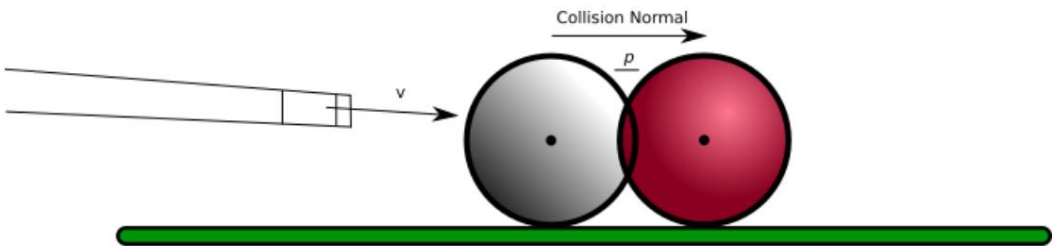
Agora conhecemos métodos para calcular interseções entre algumas formas simples e determinar o ponto de contato, a normal de colisão e a distância de interseção. No entanto, isso é apenas metade da solução - além de detectarmos as colisões entre os nossos corpos físicos, devemos então calcular a resposta correcta à colisão dessas colisões, para manter a precisão física da nossa simulação. Dois objetos físicos não podem ocupar o mesmo ponto no espaço ao mesmo tempo, então parte da resposta à colisão é separar os objetos. Além disso, também temos que determinar se os objetos transmitem forças um sobre o outro - para reutilizar o exemplo de sinuca dos tutoriais anteriores, se a bola branca atingir outra bola, primeiro detectamos que as duas bolas atingiram e, em seguida, determinamos a resposta física correta à colisão dos objetos, calculando a quantidade de energia a ser transferida de uma bola para a outra.

Resposta a Colisões

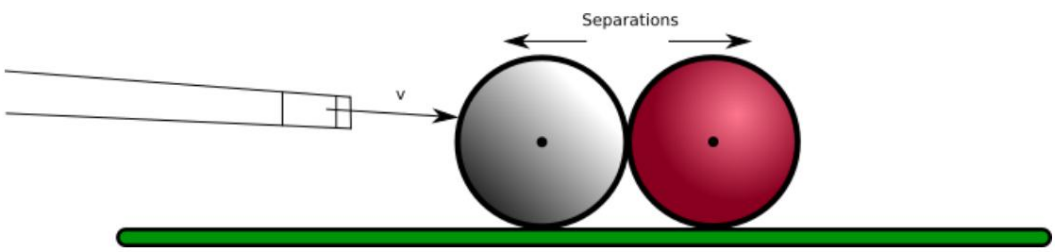
À medida que os objetos em nossa simulação se movem devido às forças exercidas sobre eles, eles se cruzarão em algum ponto - algo que agora podemos detectar para esferas e cubos. Para manter a consistência da simulação, bem como detectar a colisão, devemos resolvê-la, movendo os objetos de forma que parem de se cruzar e, se necessário, alterar o momento de cada objeto de forma realista. Para fazer isso, devemos de alguma forma alterar a posição dos objetos em colisão ao longo do tempo. Vimos que podemos alterar a posição de um objeto diretamente, mas também alterando as derivadas da posição – velocidade e aceleração. Cada uma delas é uma solução válida para a nossa resposta à colisão e, portanto, existem métodos que utilizam todas as três de alguma forma.

O Método de Projeção

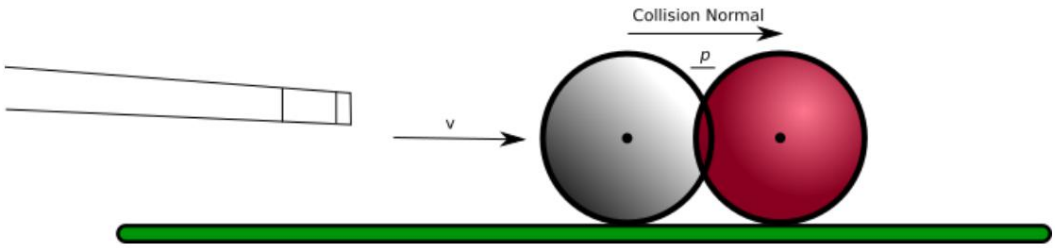
Talvez o método mais simples e intuitivo de resolver uma colisão seria simplesmente separá-los, alterando a sua posição, em direções opostas ao longo da normal de colisão. Isso é conhecido como método de projeção e, embora simples, funciona até certo ponto. No entanto, tem algumas desvantagens. Se considerarmos novamente o caso do jogo de sinuca e imaginarmos que a bola branca foi rebatida e viajou em linha reta até colidir com outra bola:



Podemos então 'resolver' a colisão separando-os ao longo da normal de contato:



Isso fornece algum tipo de resposta à colisão, para um quadro. O que acontece então no próximo quadro? A bola branca ainda está se movendo em sua velocidade anterior, e a bola que colidiu com a bola não tem velocidade atual, pois nenhuma energia foi transferida da bola branca para a outra, apenas modificamos seus vetores de posição. No quadro seguinte, então, eles colidem novamente, à medida que a bola branca avança e colide com a outra bola. Isso continua acontecendo a cada quadro, e a bola branca 'empurra' a outra bola ao longo da colisão normal até que elas não colidam mais - se a direção normal da colisão e a direção da velocidade da bola branca corresponderem exatamente (o produto escalar entre eles é 0) , então a bola é empurrada para sempre, ou pelo menos até que o atrito e o amortecimento reduzam a velocidade da bola branca a zero:



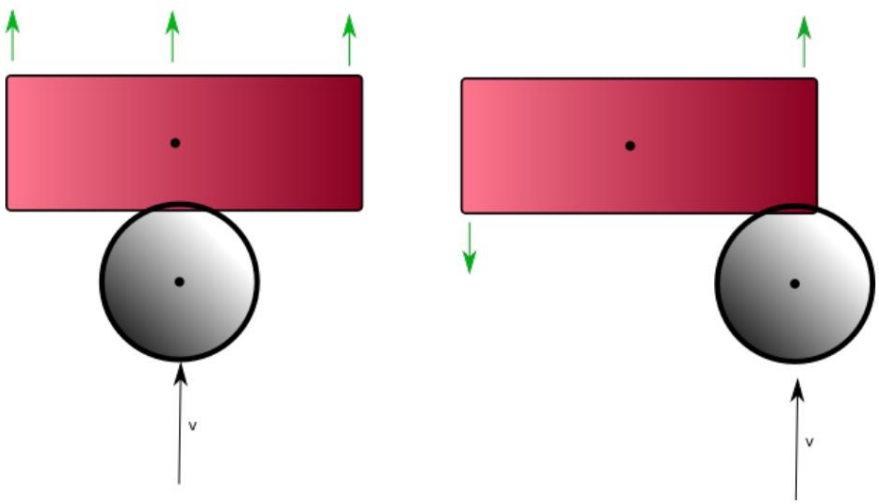
Como o método de projeção não leva em consideração as velocidades relativas, ou massas, entre os objetos, não é particularmente adequado por si só. Considere se, em vez disso, a bola branca atingisse um planeta - o método de projeção apenas moveria o planeta e a bola ao longo da colisão normal na mesma proporção, e a precisão física da simulação seria perdida.

Podemos melhorar este conceito de separação, em vez de apenas separar os objetos por uma quantidade igual, em vez disso, separando-os por uma quantidade proporcional à sua massa - por isso, se a nossa bola branca acima fosse de alguma forma duas vezes mais pesada que a bola vermelha, seria ser movido ao longo da normal de colisão pela metade da bola vermelha. Isso ajuda o exemplo da nossa bola a atingir um planeta, mas não impede que o exemplo simples do Snooker não 'pareça' certo, pois a bola branca ainda não perde velocidade, então ela continua avançando e empurrando a bola vermelha.

O Método do Impulso

Em vez de modificar diretamente a posição de um objeto, é possível resolver a colisão modificando a primeira derivada da posição; isto é, as velocidades dos objetos em colisão. Isto pode ser conseguido calculando uma nova velocidade para cada objeto que os afastará de maneira realista, levando em consideração sua massa relativa e velocidade atual (ou seja, o momento de cada objeto).
Uma mudança instantânea na velocidade, sem primeiro integrar a aceleração em relação ao tempo, é conhecida como impulso e, portanto, a resposta à colisão usando-os é conhecida como método do impulso.

Para aplicar corretamente um impulso, devemos levar em consideração as velocidades lineares e angulares dos objetos e modificá-las de acordo. Para entender o porquê, aqui estão alguns casos simples de colisão de objetos:



No exemplo à esquerda, a esfera atinge o cubóide diretamente e parece intuitivo o suficiente para que o cubóide seja empurrado para longe. No exemplo certo, a esfera está atingindo o canto do cubóide, e a coisa 'correta' que ocorreria seria que, embora nem a esfera nem o cubóide tivessem qualquer movimento angular no ponto de colisão, o cubóide deveria 'torcer 'sob o impacto - estes resultados esperados são indicados usando setas verdes no exemplo.

Para ver como um impulso J aplicado ao longo do tempo t se relaciona com uma mudança no momento, podemos dividi-lo em equações individuais. Geralmente temos o delta de tempo de um único quadro, pois queremos 'disparar e esquecer' nosso impulso e fazer com que ele faça a coisa certa. Quanto ao motivo pelo qual J é a notação matemática padrão para um impulso... ninguém consegue realmente se lembrar, mas pode estar relacionado ao fato de não ser capaz de usar 'i', pois já era comumente usado para denotar inércia em equações físicas, onde isso pode causar confusão.

Começamos com a definição simples do nosso impulso J sendo definido como alguma força F aplicada ao longo do tempo t:

$$J = F\ddot{y}t$$

A segunda lei de Newton afirma que F = ma, (isto é, massa vezes aceleração), ou alternativamente o taxa de mudança na velocidade (v) ao longo do período de tempo t:

$$J = F\ddot{y}t$$
$$J = ma\ddot{y}t$$
$$J = m \quad v\ddot{y}t\ddot{y}t \_$$
$$J = m\ddot{y}v$$

Agora que podemos ver que um impulso J é apenas uma mudança na velocidade, o que fazemos com ele e como determinamos realmente o que inserir na nossa equação para calcular o J correto para uma determinada colisão? O objetivo do impulso é alterar de alguma forma a velocidade de cada um dos nossos objetos em colisão, de tal forma que o momento seja conservado e, portanto, levar em consideração a velocidade e a massa:

$$\ddot{y}v = \frac{J.}{eu}$$

Se calcularmos o J correto, então cada objeto deverá se mover em uma quantidade proporcional à sua massa; embora o momento geral da colisão deva ser conservado, isso não significa que os dois objetos devam se mover na mesma proporção, e um objeto com massa maior deve se mover menos em uma colisão do que outro com menos massa (para conservar o momento dos nossos objetos em colisão, isso significa apenas que o objeto mais pesado adiciona mais velocidade ao objeto mais leve do que vice-versa).

Coeficiente de restituição

Antes de entrarmos nos cálculos completos para calcular os impulsos, vale a pena pensar brevemente sobre a composição dos objetos em colisão e como isso pode afetar a resposta às colisões entre eles. Se deixássemos cair duas bolas, cada uma pesando 1kg, no chão da mesma altura, esperaríamos que ambas reagissem da mesma maneira. Mas e se uma bola fosse feita de borracha e a outra fosse feita de aço? A bola de borracha provavelmente saltaria no ar, enquanto a bola de aço saltaria muito menos; dito de outra forma, depois de colidir com o chão, a bola de borracha retém mais velocidade do que a bola de aço.

A razão entre a velocidade de um objeto antes e depois de uma colisão é conhecida como coeficiente de restituição (geralmente denotado por e) e representa a perda de energia cinética devido à conversão em calor ou à deformação do material do objeto. Este coeficiente varia entre 0 e 1 para a maioria das combinações de materiais, com um valor de 1 sendo considerado uma colisão perfeitamente elástica, e menos de 1 sendo uma colisão inelástica. Um valor acima de 1 ganharia energia cinética, o que significa que a massa, o calor ou a luz estavam de alguma forma sendo transformados em uma mudança na velocidade - talvez o material esteja explodindo!

Na verdade, o coeficiente de restituição refere-se à mudança esperada na velocidade após a colisão de um par específico de objetos em colisão, de modo que o coeficiente seria diferente para duas bolas de aço colidindo do que seria para uma bola de aço e uma bola de borracha colidindo. Embora pudéssemos usar algum tipo de tabela de pesquisa para armazenar os coeficientes exatos para um determinado objeto 'material', no código é comum aproximar o coeficiente armazenando um valor de restituição por objeto e, em seguida, multiplicando as duas restituições para nos dar um valor que determina quanta energia cinética é perdida.

Cálculo do Impulso Linear

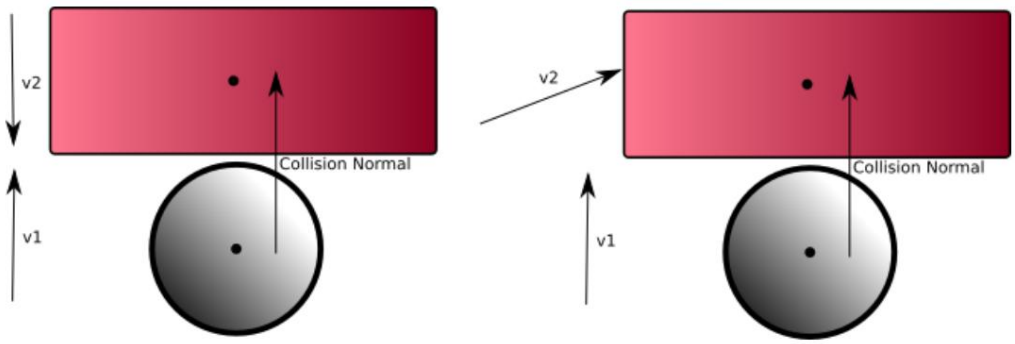
Para ver como construir o cálculo correto para nossa resposta ao impulso J, vamos considerar uma colisão simples entre dois corpos rígidos, cada um com uma velocidade vx, uma massa inversa mÿ1 e um $_{x,}$ tensor de inércia inversa I $^{ÿ1}_{x,}$ onde x em cada caso será 1 para o primeiro objeto e 2 para o segundo. Se você não consegue se lembrar por que estamos usando massa inversa e inércia, lembre-se de que isso nos ajuda a transformar algumas equações de divisões em multiplicações e nos permite representar facilmente objetos com massa 'infinita' e, portanto, nunca devem se mover.

Ao calcular o valor do componente linear da resposta ao impulso, precisamos levar em consideração consideração a velocidade relativa entre os dois objetos:

$$vr = v1 ÿ v2$$

Não importa se nossos objetos estão se movendo para frente a 1m/s e 2m/s ou 1001 m/s e 1002 m/s – a resposta à colisão entre eles deve ser a mesma.

Também precisamos de saber quanto dessa velocidade relativa está na direção da normal colisão. Para entender por quê, considere os dois casos a seguir de objetos se movendo nas mesmas duas velocidades, mas em direções diferentes:



Em ambos os casos, a velocidade relativa entre os objetos é a mesma, mas no caso direito os objetos estão colidindo apenas estreitamente na direção em que estão viajando, enquanto no caso esquerdo a normal de colisão corresponde à direção em que estão viajando, e assim deve resultar em mais energia sendo transferida.

Para calcular quanto da velocidade relativa está na direção da normal de colisão ˆn, podemos simplesmente usar o operador de produto escalar. Isto, combinado com o coeficiente de restituição dos objetos em colisão, permite-nos determinar a velocidade total da colisão:

$$vj = ÿ(1 + e)vr \cdot ˆn$$

Ainda não terminamos! Ainda precisamos levar em consideração a conservação do momento. Lembre-se de que o momento é a massa vezes a velocidade, e que queremos que o momento geral dos nossos objetos em colisão seja o mesmo depois de terem colidido como era antes (o que não quer dizer que as direções ou magnitudes da velocidade não mudarão, é apenas que entre os dois objetos será igual). Para conservar o momento, dividimos a velocidade do impulso pela soma das massas inversas dos objetos - isso essencialmente dimensiona a velocidade para que, quando a aplicamos em cada objeto, ela obtenha a proporção correta da energia total da colisão (lembre-se que J = mÿv; calculamos o total da mudança em v, só então precisa ser escalonado pelo mÿ1 de cada objeto).

$$J = \frac{vj}{(mÿ_1 + mÿ1)_2}$$

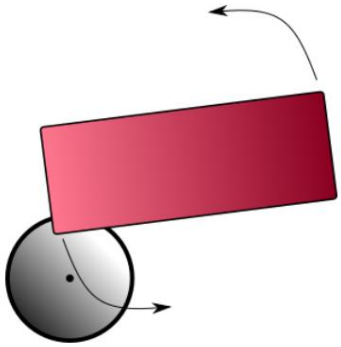
Assim que tivermos o impulso J, podemos ajustar as velocidades de cada um dos objetos em colisão, de modo que eles reajam à colisão e conservem o momento do sistema:

$$v^+_1 = v1 + mÿ1_1$$
$$v^+_2 \quad Jnˆ = v2 ÿ {}_2 mÿ1N$$

Cada objeto acabará, portanto, ganhando alguma velocidade ao longo da colisão normal (seja para frente ou para trás) proporcionalmente à sua massa inversa, afastando os objetos em uma quantidade variável. Agora, se a nossa bola branca atingir outra bola, ela irá parar, enquanto a outra bola será lançada para frente, permitindo-nos jogar uma partida de Snooker.

Cálculo do Impulso Angular

Mas isso não é tudo. Além da velocidade linear, precisamos levar em consideração também a velocidade angular dos nossos objetos. Para entender o porquê, aqui está um exemplo simples:



Imagine que nem a esfera nem a caixa estão se movendo no espaço (sua velocidade linear é zero), mas que a caixa vermelha está girando. Em algum momento a caixa irá colidir com a esfera, gerando um ponto de contato na borda da caixa. Claramente, a esfera deve ser derrubada pela caixa giratória, mas como até agora consideramos apenas a velocidade linear em nosso impulso, J terá neste caso uma magnitude zero e nada acontecerá.

Claramente, então, devemos aumentar a equação do impulso para levar em consideração a velocidade angular.
Se você se lembra do tutorial sobre movimento angular, fomos apresentados aos tensores de inércia e à ideia de que aplicamos um torque aos nossos objetos quando uma força é aplicada em sua superfície, com a magnitude do torque sendo proporcional à distância do Centro da massa. A partir deste conceito, é lógico que a velocidade angular pode criar uma força, com magnitude proporcional à distância do centro de massa - afinal, a terceira lei de Newton afirma que toda ação tem uma reação igual e oposta!

Para levar em consideração a força adicional que a velocidade angular pode aplicar, precisamos dimensionar ainda mais a velocidade do impulso, não apenas pela massa inversa dos objetos, mas também pelos seus tensores de inércia inversa. Devemos calcular a quantidade de velocidade angular que cada objeto tem no ponto de colisão tomando o produto vetorial entre a posição relativa rx no objeto (que é o ponto de colisão no espaço mundial menos a posição do objeto no espaço mundial) e a colisão normal, e então cruze isso pela posição relativa no objeto mais uma vez - isso efetivamente dimensiona a velocidade angular de acordo com o quanto ele está se movendo na direção da colisão normal:

$$\ddot{y}1 = I1 \quad \ddot{y}1 \; (r1 \times \hat{n}) \times r1$$
$$\ddot{y}2 = I2 \quad \ddot{y}1 \; (r2 \times \hat{n}) \times r2$$

Podemos então modificar a velocidade angular dos objetos em colisão obtendo o produto vetorial entre a posição de colisão local e a força de impulso aplicada e dimensionando o resultado pelo tensor de inércia inversa - da mesma forma que fizemos quando aplicamos forças usando raios. av+ = av1 + I1

$$av+\tfrac{1}{2} = av2 \; \ddot{y} \; I2 \quad \ddot{y}1 \; (r1 \times Jn\hat{})$$
$$\ddot{y}1 \; (r2 \times Jn\hat{})$$

Cálculo de Impulso Combinado

Para transformar as velocidades angulares dos nossos dois objetos de volta em valores escalares adequados para serem adicionados à nossa função de impulso, consideramos o produto escalar deles versus a direção normal da colisão. Podemos então adicionar isso ao divisor do nosso vetor de impulso, juntamente com as massas inversas, dando-nos uma equação de impulso final de:

$$J = \frac{\ddot{y}(1 + e)vr \cdot \hat{n}}{\ddot{y}1 \; \ddot{y}1 + m2 \; m1 \quad + (I1 \quad \ddot{y}1 \; (r1 \times \hat{n}) \times r1 + I2 \quad \ddot{y}1 \; (r2 \times \hat{n}) \times r2) \cdot \hat{n}}$$

Parece bastante desagradável, mas se resume a um único valor escalar e nos dá tudo o que precisamos para determinar a força de colisão e nos permite conservar o momento (tanto linear quanto angular). Depois de calcularmos a soma total da velocidade do impulso, ajustamos a velocidade linear e angular dos objetos e, ao fazer isso, resolvemos a colisão.

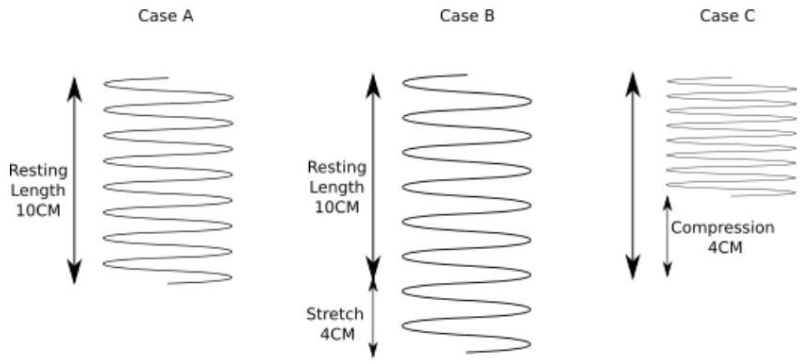
O Método da Penalidade

Vimos como podemos resolver colisões modificando diretamente a posição dos objetos em colisão (através do método de projeção) e através da primeira derivada da posição (o método do impulso). É lógico que também poderíamos ajustar a segunda derivada da posição (a aceleração do objeto) para afastá-los e, assim, resolver a colisão. Os métodos de resolução de colisão que ajustam a aceleração são conhecidos como métodos de penalidade. Vimos no tutorial sobre movimento linear que alteramos a aceleração de um objeto adicionando uma força, então os métodos de penalidade calculam uma força a ser aplicada que irá, com o tempo, resolver a colisão. Isso geralmente é feito aplicando cálculos de mola aos objetos em colisão - quanto mais os objetos penetram, mais a 'mola' que os conecta deseja retornar ao seu comprimento de repouso.

Podemos modelar uma mola usando a Lei de Hooke, que calcula a força aplicada em cada extremidade de uma mola quando ela é comprimida ou estendida. A lei pode ser representada como uma única equação:

$$F = ÿkx$$

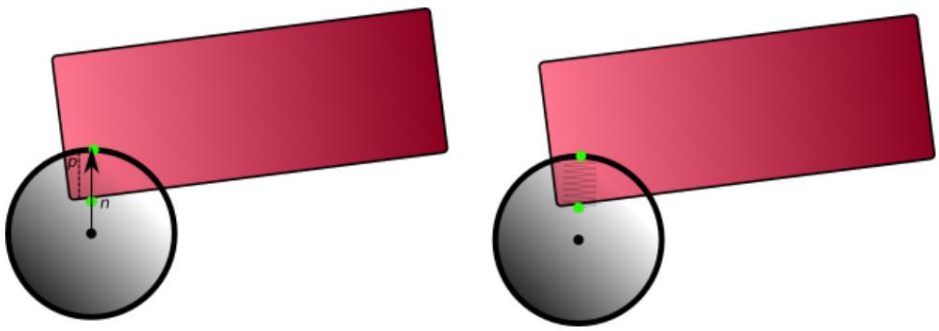
Isso afirma que a força da mola F é o resultado da diferença entre o comprimento da mola x e o comprimento de 'repouso', multiplicado pela constante da mola k, que representa o quão 'ágil' a mola é, então quanto maior for k, mais força é aplicado, e mais rápido a mola tenta retornar ao seu comprimento de repouso. Aqui estão alguns exemplos de molas e como a lei de Hooke determina quanta força elas aplicam:



Neste caso, o comprimento de repouso da mola é de 10 cm - se tentarmos esticar a mola em 4 cm ou 'esmagá-la' em 4 cm, obteremos a mesma magnitude de força resultante (assumiremos que a constante k é 1,0 por enquanto), apenas com direções opostas:

$$Fa = 0 = ÿ(10 ÿ 10)$$
$$Fb = 4 = ÿ(10 ÿ 14)$$
$$Fc = ÿ4 = ÿ(10 ÿ 4)$$

A partir disto, podemos ver que esticar e comprimir uma mola produz a mesma quantidade de força, e que se adicionarmos a constante da mola, apenas dimensionaremos a força resultante. As molas parecem interessantes, mas como exatamente as usamos para resolver a detecção de colisões? Podemos usar esta equação da mola modelando uma mola com comprimento de repouso igual a zero, ligada aos pontos de colisão da colisão:



Podemos afirmar que a mola temporária tem comprimento de repouso zero e que está sendo estendida ao longo da normal de colisão n pela distância de penetração p. A partir disso, podemos aplicar uma força proporcional à distância de penetração, no ponto de colisão de cada objeto, na direção normal da colisão

(e assim nos dando uma aceleração e um torque para esse quadro), da mesma forma que quando aplicamos forças em um ponto específico quando realizamos a projeção de raios. Neste ponto, vale a pena recordar que a aceleração resultante de uma força é a = Fmÿ1 , o que significa que ainda obtemos algum “realismo” da nossa resolução de colisão, na medida em que os objectos mais leves se moverão mais do que os objectos mais pesados. À medida que os objetos começam a se afastar devido à força da mola, nos quadros subsequentes ela aplicará menos força (a distância de penetração está diminuindo e, portanto, x na lei de Hooke). Se os objetos continuarem a se mover juntos, a força da mola ficará cada vez maior, modificando sua velocidade até que comecem a se separar.

Resposta do jogo a colisões

Às vezes, não é apenas o mecanismo de física que precisa saber sobre uma colisão - o código do jogo também pode precisar. Em um jogo FPS, se você atingir outro jogador com seu foguete, ele poderá ser repelido (a resposta física à colisão), mas também perderá um pouco de saúde ou armadura (a resposta do jogo à colisão). Isso geralmente é feito em jogos marcando certos tipos de objetos físicos como colidindo entre si e, em seguida, tendo alguma maneira de informar ao jogo que uma colisão começou (ou às vezes até terminou - talvez o jogador pare de perder saúde quando parar de colidir com as telhas de 'lava'). Por exemplo, em jogos construídos com o motor de jogo Unity, os programadores têm acesso aos métodos OnCollisionEnter e OnCollisionExit, que são chamados pelo motor de física para informar o código de jogo de qualquer objeto que implemente esses métodos em seu script de jogo e que tenha atingido. outro objeto.

Implementar algum tipo de método pelo qual o mecanismo de física possa informar ao jogo que estão ocorrendo colisões é importante para criar uma experiência interativa. Há uma variedade de maneiras diferentes de conseguir isso e não existem regras rígidas e rápidas, embora algumas maneiras possam ter implicações no desempenho. A base de código que você recebeu possui funções virtuais finais OnCollisionBegin e OnCollisionEnd, permitindo a criação de interações de jogo personalizadas - o jogo será informado quando ocorrerem colisões e você poderá adicionar algum código de jogo em várias subclasses para representar o foguetes/carros/pessoas/robôs que compõem seu jogo. Porém, chamá-lo desta forma é prejudicial à eficiência do seu mecanismo de física, devido ao cache de instruções da CPU. Os computadores funcionam melhor quando fazem a mesma coisa repetidamente, em um conjunto ordenado de dados - todo o código estará no cache de instruções, tornando sua execução rápida e o acesso aos dados é previsível, ajudando a CPU para pré-armazenar em cache os dados corretos, para que não seja necessário parar e esperar que os dados sejam carregados da memória principal. Se decidirmos chamar uma função virtual no meio de nossos cálculos de física, a CPU então para o que estava fazendo e começa a executar sua função virtual, o que pode fazer com que o código de 'física' saia do cache, de modo que quando o função virtual termina, a CPU deve esperar um pouco antes de poder executar o próximo bit do código de detecção de colisão. O que é pior é que se o mecanismo de física executar vários cálculos de detecção de colisão por 'frame de jogo', ou for multithread, então essas funções poderão ser chamadas na hora errada, no núcleo errado, no número errado de vezes por atualização do jogo!

Uma solução melhor pode ser adicionar colisões a uma lista, que pode então ser iterada assim que toda a detecção de colisão for concluída, ou talvez até mesmo marcar objetos físicos 'especiais' que tenham efeitos colaterais de jogo (como aqueles foguetes no anterior exemplo) e adicione apenas esses objetos a uma lista.

Código do Tutorial

Para obter resposta à colisão em nosso mecanismo de física, implementaremos uma combinação do método de projeção e do método de impulso. Isso nos permite separar os objetos rapidamente (remover a penetração do objeto naquele quadro) e, em seguida, afastá-los de maneira realista. Precisamos apenas preencher mais um método na classe PhysicsSystem, ImpulseResolveCollision. Como a seção sobre o método de impulso pode ter sugerido, o código será bastante longo e envolverá muitas operações vetoriais, mas só precisamos calculá-lo uma vez e não precisamos considerar as diferentes colisões formas, já que qualquer uma das colisões se reduz ao normal de colisão, à distância de penetração e ao ponto de colisão, cada um dos quais usaremos em diferentes partes deste código. Aqui está a primeira parte do método, que pegará um par de GameObjects e um ponto de colisão, e então aplicará as forças corretas aos objetos, com base nas informações de colisão:

```
1  void PhysicsSystem :: ImpulseResolveCollision (
2  GameObject & a, GameObject & b CollisionDetection ::,
3          ContactPoint & p) const {
4          FísicaObject * physA = a . GetPhysicsObject();
5          FísicaObject * physB = b . GetPhysicsObject();
6
7          Transformar e transformarA = a . ObterTransforma();
8          Transformar e transformarB = b . ObterTransforma();
9
10         float totalMass = physA - > GetInverseMass ()+ physB - > GetInverseMass ();
11
12 // Separe-os usando projeção
13         transformarA. SetWorldPosition (transformA. GetWorldPosition () -
14           ( p . normal * p . penetração *( physA -> GetInverseMass () / totalMass )));
15
16         transformarB . SetWorldPosition (transformB. GetWorldPosition () +
17           ( p . normal * p . penetração *( physB -> GetInverseMass () / totalMass )));
```

Método PhysicsSystem::ImpulseResolveCollision

Para tornar as coisas um pouco menos propensas a erros (e mais fáceis de ler!), a primeira coisa que faremos é obter os objetos físicos de nossos dois objetos em colisão e suas transformações, e armazená-los em algumas variáveis (linhas 4 a 8). Também vamos determinar a massa inversa total dos dois objetos, à medida que vamos usar isso mais tarde para calcular o impulso J, mas também para nossa simples projeção de objeto.

Nas linhas 13-17, empurramos cada objeto ao longo da normal de colisão, por uma quantidade proporcional à distância de penetração e massa inversa do objeto. Dividindo a massa de cada objeto pela totalMass variável, fazemos com que entre os dois cálculos, o movimento do objeto em uma quantidade total de p.penetração um do outro, mas um objeto 'mais pesado' se moverá menos, pois contém menos a 'massa total' (lembre-se, estamos lidando com massa inversa, então objetos mais pesados têm valores mais baixos).

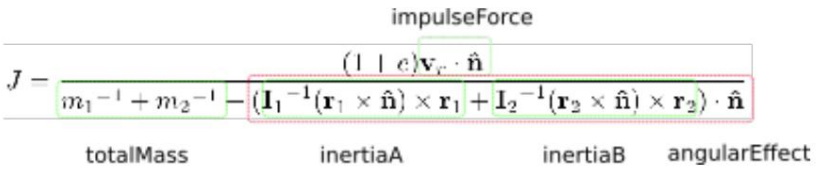
Isso é tudo o que precisamos fazer para cumprir a resposta à colisão através do método de projeção - se usarmos apenas isso, os objetos se separariam e as caixas e esferas do nosso mundo de teste ficariam felizes no chão. O piso nunca se moveria, pois sua massa inversa é 0, e portanto será projetado por uma quantidade de 0 unidades ao longo da colisão normal, e qualquer objeto caindo sobre ele se moveria pela penetração total quantia. O que não acontecerá, porém, é qualquer conservação do momento, através de uma mudança na quantidade da velocidade linear ou angular dos objetos. Para fazer isso, precisamos começar a construir as variáveis que precisa calcular o valor do impulso J e onde em cada objeto aplicar o impulso. Para fazer isso, precisamos da posição do ponto de colisão em relação a cada objeto, que podemos determinar simplesmente via subtração da posição do objeto do ponto de colisão (linhas 18 - 19). A partir desses pontos, podemos determine quanta velocidade angular o objeto tem naquele ponto (quanto mais longe do centro do objeto é um ponto, mais rápido ele se move à medida que o objeto gira). Podemos determinar quanto objeto está se movendo através do produto vetorial (linhas 21 e 22), da mesma maneira que determinamos o quantidade de torque usando o produto vetorial ao aplicar forças angulares. Agora, a partir de uma combinação da velocidade linear e angular de cada objeto, podemos determinar as velocidades nas quais os dois objetos estão colidindo, via simples adição (linhas 26 - 27) e, portanto, a velocidade relativa da colisão (linha 29).

```
18         Vetor3 relativoA = p . posição - transformA . GetWorldPosition();
19         Vetor3 relativoB = p . posição - transformB . GetWorldPosition();
20
21         Vetor3 angVelocidadeA =
22                 Vector3 :: Cruzar ( physA -> GetAngularVelocity ()                 ,   relativoA );
23         Vetor3 angVelocidadeB =
24                 Vector3 :: Cruzar ( physB -> GetAngularVelocity ()                 ,   relativoB);
25
26         Vector3 fullVelocityA = physA -> GetLinearVelocity () + angVelocityA ;
27         Vector3 fullVelocityB = physB -> GetLinearVelocity () + angVelocityB ;
28
```

```
29      Vetor3 contactVelocity = fullVelocityB - fullVelocityA ;
```

Método PhysicsSystem::ImpulseResolveCollision

Agora, para começar a construir esse vetor de impulso J, que para referência foi calculado da seguinte forma:



Para facilitar o acompanhamento, os nomes das variáveis que armazenam cada parte da equação foram anotado, para que você possa ver como a base de código implementa cada um dos bits que compõem J. Por enquanto, vamos codificar o coeficiente de restituição para algo que irá dissipar alguma energia - veja o que acontece com as colisões quando esta variável é alterada (especialmente quando o valor é maior que 1...).

```
30      float impulsoForce = Vector3 :: Dot ( contactVelocity , p. normal );
31
32      // agora vamos calcular o efeito da inércia....
33      Vector3 inérciaA = Vector3 :: Cross ( physA -> GetInertiaTensor () *
34                                  Vetor3 :: Cruzar (relativoA, p. normal)                    ,    relativoA );
35      Vector3 inérciaB = Vector3 :: Cross ( physB -> GetInertiaTensor () *
36                                  Vetor3 :: Cruzar (relativoB, p. normal) relativoB);        ,
37      float angularEffect = Vector3 :: Dot ( inertiaA + inertiaB , p. normal );
38
39      float cRestituição = 0,66 f ; // dispersa alguma energia cinética
40
41      float j = (-(1,0 f + cRestituição) * impulsoForce) /
42                          (massa total + efeito angular);
43
44      Vetor3 fullImpulse = p . normal *j;
```

Método PhysicsSystem::ImpulseResolveCollision

A última parte da função é aplicar os impulsos lineares e angulares, em direções opostas, e fornecendo uma resposta realista às nossas colisões de objetos:

```
45      physA -> ApplyLinearImpulse (-fullImpulse);
46      physB -> ApplyLinearImpulse (fullImpulse);
47
48      physA - > ApplyAngularImpulse ( Vector3 :: Cross ( relativoA physB - >                    , - fullImpulse ));
49      ApplyAngularImpulse ( Vector3 :: Cross ( relativoB                                        , fullImpulse ));
50}
```

Método PhysicsSystem::ImpulseResolveCollision

Como não usamos os métodos ApplyLinearImpulse ou ApplyAngularImpulse antes, precisaremos preencha-os com o seguinte código:

```
1 void PhysicsObject :: ApplyAngularImpulse ( const Vector3 & force ) {
2 angularVelocity += inverseInteriaTensor * force ;
3}
4
5 void PhysicsObject :: ApplyLinearImpulse ( const Vector3 & force ) {
6       linearVelocity += força * inverseMass ;
7}
```

Classe FísicaObject

Cada um apenas dimensiona sua entrada pela representação de massa inversa apropriada e a adiciona ao vetor de velocidade apropriado.

Para ver onde nosso método recém-preenchido está sendo usado, verifique o método BasicCollisionDetection fomos apresentados no tutorial anterior. Se a colisão for detectada, passamos os objetos em colisão para o novo método ImpulseCollisionDetection, que adicionará os impulsos instantaneamente. mudar nossa velocidade linear e angular de modo que os objetos se afastem.

```
1       if (CollisionDetection :: ObjectIntersection (* i info )) {                                   , *j ,
2               ImpulseResolveCollision (* info .a info . point );              ,     * informações .b  ,
3               informações. framesLeft = numCollisionFrames;
4               todasColisões. inserir (informações);
5       }
```

Mudanças no método PhysicsSystem::BasicCollisionDetection

Este método também adiciona o par de colisão em um stl::set, nos dando uma lista única de interseções para aquele quadro de jogo específico. Por que ele faz isso? No final de cada quadro, o UpdateCollisionList método também é chamado:

```
6 void PhysicsSystem :: UpdateCollisionList () {
7 for (std :: set < CollisionDetection :: CollisionInfo >:: iterador
8                 i = todasColisões. começar (); eu != todasColisões . fim (); ) {
9               if ((* i). framesLeft == numCollisionFrames ) {
10                      i - >a - >OnCollisionBegin (i - > b );
11                      i ->b -> OnCollisionBegin (i -> a );
12              }
13              (* eu ). quadrosEsquerda = (* i ). quadrosEsquerda - 1;
14              if ((* i). framesLeft < 0) {
15                      i - >a - >OnCollisionEnd (i - > b );
16                      i - >b - >OnCollisionEnd (i - > a );
17                      i = todasColisões. apagar (eu);
18              }
19              outro {
20                      ++eu;
21              }
22      }
23}
```

Mudanças no método PhysicsSystem::BasicCollisionDetection

No primeiro quadro um par de colisão é adicionado à lista, os respectivos GameObjects têm seu método OnCol-lisionBegin chamado, caso contrário um contador é reduzido dentro da estrutura CollisionInfo - se este torna-se menor que zero, assumimos que os objetos não estão mais colidindo e removemos a colisão, chamando OnCollisionEnd nos objetos, para que mais interações de jogo possam ser codificadas (talvez o o jogador para de perder saúde quando sai de uma poça de lava). Isso permite que as chamadas da função 'gameplay' sejam ser agrupado no final de um quadro, em vez de ser chamado no meio do nosso mecanismo de física loop de atualização, onde pode resultar em problemas de uso de cache para nosso código.

Conclusão

Se você criar agora um mundo de jogo usando uma das funções fornecidas no arquivo principal, você deverá agora ser capaz de derrubar objetos usando raios ou aplicando gravidade para que caiam no chão, onde esperançosamente, eles deveriam saltar de volta no ar e descansar, em vez de apenas continuar enquanto fizemos no passado.

Neste tutorial, examinamos métodos para resolver as colisões entre nossos objetos no mundo. Vimos como os impulsos lineares e angulares ajudam a afastar os objetos em colisão e como o atrito e a

resposta à colisão de mudança de elasticidade. No próximo tutorial, daremos uma olhada em maneiras pelas quais a detecção de colisão pode se tornar mais eficiente por meio do uso de verificações de colisão de fase ampla e de fase estreita.

Trabalho adicional

1) No código para determinar a resolução de colisão, atualmente temos um coeficiente de reposição codificado. Tente adicionar uma nova variável à classe PhysicsObject que armazena a elasticidade de cada objeto e, em seguida, multiplique-as na variável cRestitution do método ImpulseResolveCollision. Você deverá então ser capaz de definir bolas de borracha (uma variável de alta 'elasticidade') e bolas de aço (uma elasticidade baixa) e ver como cada uma responde ao cair da cavidade no chão.

2) O ImpulseResolveCollision atual usa uma combinação dos métodos de projeção e impulso para manter a consistência dos objetos na cena. Este tutorial também discutiu o ideal de usar um cálculo de mola para alterar as acelerações dos objetos, em vez da posição ou velocidade. Tente criar um método ResolveSpringCollision que use os cálculos de mola de Hooke para mover os objetos - lembre-se de que uma mola tem um comprimento de repouso e um coeficiente de mola para indicar a rapidez com que a mola deve 'voltar' ao seu comprimento de repouso quando esticada ou comprimida.