Física - Restrições e Solucionadores

Introdução

Na série de tutoriais até agora, vimos como aplicar forças a corpos rígidos, para produzir movimentos lineares e angulares. Também usamos a detecção de colisão para ver se os objetos estão sobrepostos e usamos a resolução de colisão para separá-los novamente. Até agora, nossos objetos estavam completamente separados uns dos outros - nunca criamos um objeto que estivesse preso a outro de alguma forma, como os rolamentos de esferas de um berço de Newton presos à estrutura com um fio.

Para aprimorar nosso mecanismo de física para suportar corpos rígidos conectados, neste tutorial investigaremos restrições e solucionadores. As restrições nos permitem definir formalmente as conexões entre objetos em termos de uma equação de igualdade e nos permitem aplicar impulsos para manter os corpos rígidos separados por uma certa distância ou por uma certa orientação relativa entre si. Usando-os, podemos simular cordas, dobradiças e juntas esféricas. À medida que mais restrições são adicionadas entre objetos, nos depararemos com casos em que um corpo rígido está sendo manipulado por múltiplas restrições, que puxam os objetos de maneiras diferentes. Essas combinações de restrições devem ser calculadas e corrigidas usando um solucionador - algo que considera um estado mundial e aplica os cálculos corretos para manter o movimento correto de nossos objetos com o mínimo de erros possível.

Restrições

Você pode pensar que a resposta para esse problema é apenas afastar um pouco os objetos uns dos outros, mudando sua posição. Embora isso funcione, não é particularmente preciso fisicamente. Fazer isso não leva em consideração as massas dos objetos ou as direções em que eles estão viajando no momento. Estaríamos essencialmente 'teletransportando' os objetos, o que pode significar que eles se movem através de objetos que não deveriam, e irá efetivamente adicionar energia ao objeto (se movermos um objeto do ponto A para o ponto B instantaneamente, ele não terá tanto arrasto ou atrito aplicado a ele como se chegasse ao ponto B sob suas próprias forças, então ele pode agora vá mais longe do que deveria, à medida que essas forças são gastas).

Para levar em consideração todas estas coisas, as restrições são geralmente mantidas através da manipulação das derivadas da posição; isto é, a velocidade e a aceleração do objeto. Portanto, em vez de nos “teletransportarmos” para uma posição/orientação que satisfaça as regras da restrição, aplicamos forças ou impulsos para ajustar rapidamente o objeto, para que as mudanças passem pelo nosso sistema físico e mantenham a consistência tanto quanto possível.

As restrições são extremamente úteis ao escrever código de física. A física dos jogos trata da resolução de interações entre objetos de uma forma fisicamente precisa, sejam elas colisões entre objetos, uma quantidade constante de força sendo adicionada em um ponto específico ou uma corda amarrando dois objetos; todas essas coisas e muito mais podem ser escritas como uma restrição. De certa forma, assim como os shaders são o alicerce para adicionar recursos gráficos exclusivos ao nosso jogo, as restrições são o método pelo qual adicionamos recursos físicos exclusivos ao nosso jogo.

Graus de liberdade

Como o nome sugere, uma restrição restringirá ou limitará nosso objeto físico de alguma forma.
As restrições normalmente precisarão, de alguma forma, ajustar a orientação ou posição de um objeto - chamamos esses aspectos modificáveis de graus de liberdade, então nossos objetos começam com 6 graus de liberdade (às vezes chamados de 6DoF) - os eixos x, y e z posições, juntamente com as rotações em torno dos eixos x, y e z. A função de uma restrição é, de alguma forma, limitar a capacidade de cada um desses graus de liberdade de

mudança, de modo que, no geral, a restrição seja considerada 'satisfeita'. Observe que embora até agora tenhamos usado quatérnios para armazenar uma rotação e, portanto, usado 4 valores, isso não é '4 graus de liberdade' - os quatérnios são uma ferramenta para armazenar os 3 eixos de rotação de forma eficiente e de uma forma que nos permite transformar e armazená-los de forma eficiente, mas ao contrário de uma posição, onde cada valor é totalmente separado, os valores de um quaternion estão todos relacionados, e apenas alterar um não mudará apenas 'um eixo' como uma posição.

Restrições como uma equação

Durante os dois últimos módulos, vimos ocasionalmente a equação plana, geralmente formulada assim:

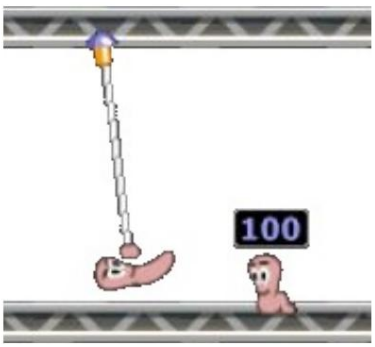
$$machado + por + cz + d = 0$$

Isto quer dizer que para qualquer ponto (xyz), se o resultado for 0, então o ponto está no plano. Podemos então dizer ainda que se o valor for maior que 0, o ponto deve se mover ao longo da normal do plano para alcançar o plano, e se for menor que 0, o ponto deve se mover no sentido contrário ao longo da normal para alcançar o plano. Por que a equação do plano está sendo mencionada novamente? Em termos gerais, a mesma lógica para mover esse ponto para ficar exatamente em um plano é como nossas restrições funcionam - geralmente temos algum tipo de condição que queremos que seja igual a 0. Ao lidar com corpos rígidos, essas condições de restrição geralmente estarão relacionadas para a posição ou orientação do corpo rígido, e queremos alterá-los para que sua diferença em relação ao que a restrição diz que deveriam ser seja 0 - se as propriedades do corpo rígido corresponderem a uma restrição, dizemos que a restrição é satisfeito

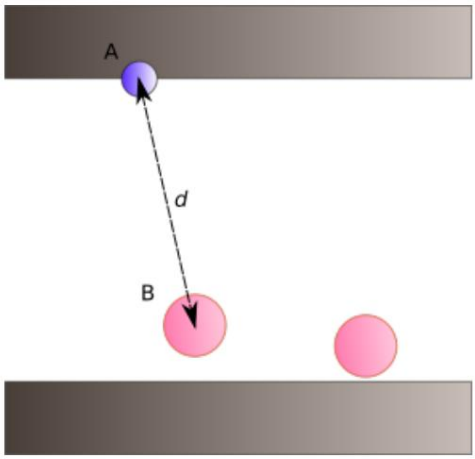
Restrição de distância

O exemplo clássico de restrição de distância em um jogo é a física do personagem 'ragdoll' - quando um inimigo 'morre', ele desmorona e pode ser empurrado pelo jogador, ou rolar colinas abaixo e cair de plataformas com resultados divertidos. Nos bastidores, podemos imaginar que o sistema normal de animação do esqueleto do inimigo foi substituído por cálculos físicos, com cada osso do esqueleto recebendo uma massa e um volume de colisão exatamente como um corpo rígido “normal”. Para evitar que os membros voem ou se desconectem de outra forma, eles estão conectados com restrições em sua posição - sua orientação pode mudar livremente, mas sua posição em relação ao conector 'pai' não pode mudar, mantendo o corpo unido, mas deixando-o oscilar. . Neste caso, estamos realmente descrevendo uma restrição que limita os graus de liberdade a 3 - permitindo que a orientação em torno do eixo 3 seja alterada à vontade, mas bloqueando completamente os 3 eixos posicionais. Tal restrição é às vezes conhecida como junta esférica, pois imita sua contraparte da vida real.

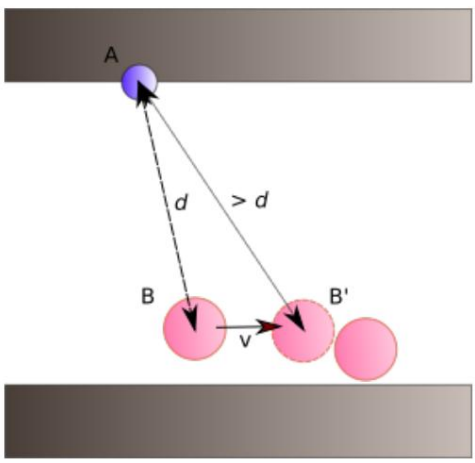
Para examinar as restrições de distância com mais detalhes, vamos considerar uma restrição bastante simples que podemos ver em jogos - o utilitário corda ninja como visto em Worms:



Aqui, o jogador disparou uma corda que colidiu com o teto (as técnicas de lançamento de raios que vimos anteriormente na série de tutoriais poderiam ser usadas para encontrar uma superfície adequada para a fixação da corda), permitindo que o personagem verme do jogador balançasse e, eventualmente, solte a corda e lance-se para outra parte do nível. Se adotarmos uma visão mais 'lógica do jogo' do que está acontecendo, poderemos ter algo assim:



Do ponto de vista das estruturas de dados e técnicas que vimos até agora neste tutorial, poderíamos replicar esta cena com alguns corpos rígidos AABB com massa inversa de 0 para representar as vigas do piso e do teto, e alguns de corpos rígidos esféricos para os personagens dos jogadores. Para implementar a corda ninja, queremos adicionar uma restrição de distância, entre o ponto A e nosso personagem jogador B, com comprimento máximo de d. Para tornar as coisas um pouco mais fáceis de acompanhar, vamos supor que o ponto A seja um corpo rígido temporário com massa inversa de 0, de modo que também não pode se mover. Quando o jogador move seu personagem para a esquerda ou para a direita, eventualmente ele estará a uma distância maior que d do ponto conectado A:



Dito de outra forma, em algum momento, o jogador irá aplicar uma força que resulta numa velocidade que quebraria a restrição, se fosse integrada na posição do personagem do jogador. Portanto, para não quebrar a restrição, temos que alterar essa velocidade para que, quando ela for integrada, a restrição de estar a uma distância d do corpo rígido A ainda seja mantida, ou satisfeita.

Podemos representar esta restrição de forma mais geral com a seguinte equação de restrição:

$$P = ||B \ddot{y} A||$$
$$C = P \ddot{y} d$$

Normalmente veremos nossa restrição anotada como sendo C desta forma, e podemos ver que seguindo o cálculo que se o objeto B estiver exatamente à distância B de A, o resultado de C será 0 e, portanto, satisfeito; se, no entanto, não for a mesma, a restrição será de alguma forma quebrada, seja por um valor positivo ou negativo.

Podemos dizer que P ÿ d representa o deslocamento dos objetos no espaço de restrições, um sistema de coordenadas onde a origem representa uma restrição 'satisfeita'. Portanto, para satisfazer a restrição, devemos de alguma forma modificar a primeira derivada da restrição. Como Pd é formado a partir da posição do objeto, podemos portanto dizer que a primeira derivada de C (que denotaremos Cÿ) é formada a partir da primeira derivada das posições do objeto; suas velocidades. Para satisfazer a restrição, devemos de alguma forma transformar as velocidades relativas do objeto, para garantir que, uma vez integradas na posição do objeto, elas as movam para mais perto de 0 no espaço da restrição. Como as velocidades são vetores, podemos transformá-las utilizando uma matriz:

$$C\ddot{y} = J \cdot v$$

Esta matriz J é conhecida como matriz Jacobiana, que define mais formalmente as derivadas parciais de uma função com valor vetorial - neste caso, a função está alterando os vetores de posição do objeto restrito em v ao longo do tempo. Geralmente não queremos perder a velocidade que está sendo aplicada (queremos que nossos objetos conservem seu momento sempre que possível), mas transformá-la de modo que satisfaça a restrição enquanto ainda se comporta fisicamente de forma consistente - é por isso que em uma restrição, nós não apenas defina as posições de forma que a restrição seja satisfeita.

Vamos dar uma olhada mais de perto no que a equação acima é formulada. O valor do vetor v são as velocidades do nosso objeto restrito (estamos lidando com as primeiras derivadas da nossa restrição):

$$v = \begin{matrix} vA \\ vB \end{matrix}$$

Nossa matriz Jacobiana é, portanto, o que a mudança na velocidade de um objeto fará com a velocidade do outro objeto se a restrição permanecer satisfeita:

$$J = \begin{matrix} VA - VB \\ VB - VA \end{matrix}^{T}$$

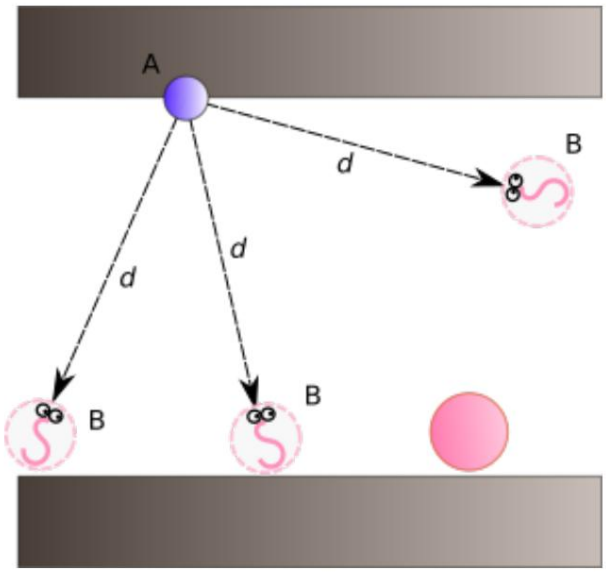
É comum ver um jacobiano denotado como uma transposta, como acima; isso ocorre porque para transformar nossa matriz 2 x 1 v, precisamos de uma matriz 1 x 2 - que é o mesmo que uma matriz 2 x 1 transposta. No campo mais amplo das funções com valor vetorial, os jacobianos podem ser difíceis de observar (há muitos operadores del:) e difíceis de $\frac{\ddot{y}u1}{\ddot{y}x1}$ visualizar (ou seja, o que eles estão tentando representar). Felizmente, estamos lidando inteiramente com conceitos com os quais já estamos familiarizados - posições no espaço 3D e velocidades que se movem através desse espaço 3D; se pensarmos em nossas transformações de restrição dessa maneira, então nosso Jacobiano se resume apenas a quanto a posição e orientação de um objeto afetam a posição e orientação de outro objeto ao tentar manter a restrição satisfeita.

Restrição Articulada

Não é apenas a posição relativa de um objeto que podemos querer restringir, mas às vezes também a orientação relativa. Consideremos o caso da porta da frente de uma casa – queremos que a porta possa abrir-se e deixar-nos entrar, mas não queremos que ela caia ou seja levantada e levada embora.
Na vida real, isso é conseguido conectando a porta ao batente por meio de uma dobradiça, que pode ser inclinada para dentro e para fora para permitir que a porta se mova. Em um ambiente simulado como um jogo, normalmente teríamos um objeto de parede com um “buraco” em forma de porta (ou vários objetos formando uma parede, cada um dos quais pode

então será totalmente convexo), um objeto de 'porta' e, em seguida, uma restrição entre os dois para permitir algumas mudanças rotacionais, mas nenhuma de posição. Em outras palavras, estamos limitando os graus de liberdade a 1 – a "guinada" da porta; todas as outras alterações não são permitidas.

Podemos continuar nosso exemplo de Worms para ver como uma restrição de dobradiça poderia funcionar. Talvez, em vez de uma distância máxima da corda d, a corda Ninja esteja restringindo o verme de modo que sua direção local 'para cima' sempre fique voltada para seu ponto de fixação, como em cada um dos locais potenciais B para nosso verme aqui:



Em termos gerais, o verme é a nossa porta e o ponto A é a nossa dobradiça - o conceito é exatamente o mesmo, pois estamos restringindo a orientação relativa; embora, é claro, nossa porta também teria sua posição restrita!

Nosso cálculo de restrição começa exatamente igual ao exemplo anterior:

$$C\dot{y} = J \cdot v$$

A diferença é que desta vez, v contém as velocidades angulares do verme e do ponto de fixação:

$$v = \begin{matrix} \dot{y}A \\ \dot{y}B \end{matrix}$$

Portanto, a matriz Jacobiana para nossa restrição é então:

$$J = \begin{matrix} (\dot{y}A - \dot{y}B) \times n & (\dot{y}B - \dot{y}A) \times n \end{matrix}^T$$

Neste caso, estamos tentando encontrar um eixo pelo qual girar a orientação de modo que a parte superior e os eixos inferiores do verme e seu ponto de fixação coincidem.

Resolução de colisão como uma restrição

Em nossas restrições de distância e orientação, estávamos formulando-as como uma equação que foi satisfeita quando se tornou 0. Se pensarmos na resolução de colisão que fizemos anteriormente na série de tutoriais, na verdade precisávamos fazer algo semelhante a uma distância restrição - tivemos que forçar os objetos que se cruzam a ficarem a uma distância de pelo menos 0 um do outro, levando em consideração a profundidade da interseção. Por outras palavras, estávamos a formar uma restrição de distância temporária que de alguma forma alterará as posições dos objectos de modo a que satisfaçam uma restrição nas suas posições de modo a que os seus volumes não se sobreponham. No entanto, algumas interseções também são resolvidas alterando as orientações dos objetos - um OBB caindo em um ângulo no chão será girado de modo que fique plano no chão. Assim, além de ser uma restrição nas posições do objeto, podemos modelar a resolução de colisões como também uma restrição nas orientações do objeto.

Vamos analisar essa ideia para ver como a restrição acaba sendo formulada. Se assumirmos nosso cálculo de restrição agora padrão:

$$C\dot{y} = J \cdot v$$

Nossa restrição de resolução teria um vetor av da seguinte forma:

$$v = \begin{matrix} \ddot{y} \\ \\ \\ \ddot{y} \end{matrix} \begin{matrix} vA \\ vB \\ \ddot{y}A \\ \ddot{y}\ddot{y}B \end{matrix} \begin{matrix} \ddot{y} \\ \\ \\ \ddot{y} \end{matrix}$$

Vimos no tutorial anterior que resolvemos colisões tentando adicionar forças ao longo da normal de colisão, que denotaremos n. Também presumimos que os objetos estavam colidindo em um ponto específico p do mundo. Para codificar o mesmo comportamento de resolução de colisão em uma restrição, terminamos com n e p dentro de nossa matriz Jacobiana:

$$J = \begin{matrix} \ddot{y} \\ \\ \ddot{y}(p\ \ddot{y}\ sA) \times n \\ \ddot{y}\ \ sB) \times n \end{matrix} \begin{matrix} -n \\ n \\ (p\ \ddot{y} \\ \end{matrix} \begin{matrix} \\ \ddot{y} \\ \\ \ddot{y} \end{matrix}$$

A partir daí, obtemos o seguinte cálculo de restrição completo:

$$C\dot{y} = \begin{matrix} \ddot{y} \\ \\ \ddot{y}(p\ \ddot{y}\ sA) \times n \\ \ddot{y}\ \ sB) \times n \end{matrix} \begin{matrix} -n \\ n \\ (p\ \ddot{y} \\ \end{matrix} \begin{matrix} \ddot{y}\ \ddot{y} \\ \\ \\ \ddot{y} \end{matrix} \begin{matrix} vA \\ vB \\ \ddot{y}A \\ \ddot{y}\ddot{y}B \end{matrix} \begin{matrix} \ddot{y} \\ \\ \\ \ddot{y} \end{matrix}$$

Para resolver nosso cálculo de restrição de interseção de objetos, precisamos que a distância de penetração se torne zero, então a derivada de uma restrição deve fazer com que a resposta se aproxime de zero, o que significa que temos que formar uma matriz Jacobiana que receba as primeiras derivadas de nosso objeto em colisão, e altera-as para se adequarem à nossa restrição - nós realmente temos feito essa forma específica de restrição de forma limitada desde o tutorial de resolução de colisão, só agora estamos vendo isso de uma forma formulada um pouco diferente. Nossas posições de objetos devem ser separadas ao longo da normal de colisão, e nossas orientações de objetos devem mudar por uma mudança angular proporcional à normal de colisão e à posição relativa da colisão - vimos no tutorial de resolução de colisão como calcular o produto vetorial destes fornece o eixo de rotação desse objeto.

Nesta restrição específica, vimos que as derivadas parciais da restrição podem acabar contendo dados extras (neste caso, a normal de colisão e a posição relativa). Podemos pensar nisso como sendo o elemento programável da restrição - diferentes restrições terão diferentes matrizes Jacobianas e podem exigir o cálculo de informações adicionais para serem formuladas adequadamente de forma que possam ser satisfeitas corretamente.

Multiplicador de Lagrange e força de restrição

A formulação básica de uma matriz Jacobiana não muda para um determinado 'tipo' de restrição - a restrição de resolução de colisão acima sempre terá uma matriz Jacobiana que transforma as posições e orientações de dois objetos, com base na colisão normal. O que não vimos até agora é a magnitude de como essas velocidades deveriam mudar, apenas em que direção elas deveriam mudar. Para satisfazer corretamente nossa restrição, devemos, portanto, dimensionar o Jacobiano de alguma forma, de modo a fazer com que resulte em C = 0 quando aplicado. Este fator de escala é conhecido como multiplicador de Lagrange, muitas vezes denotado como ÿ (e às vezes também chamado de 'lambda'). Conhecemos nosso Jacobiano e conhecemos nossas variáveis de entrada, apenas não sabemos (ainda) os valores corretos para ÿ, dando-nos uma equação que devemos resolver de alguma forma.

O verdadeiro propósito de uma restrição é adicionar algumas forças aos nossos objetos para que eles acabem em um estado que satisfaça essa restrição. Podemos, portanto, pensar na nossa restrição como sendo o processo de determinação da força de restrição linear e angular a aplicar aos nossos objetos, na forma de:

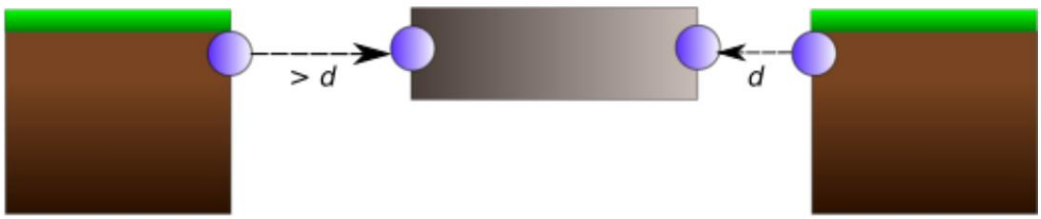
$$F = J^{T\ddot{y}}\ \_$$

Resolvendo sistemas de restrições

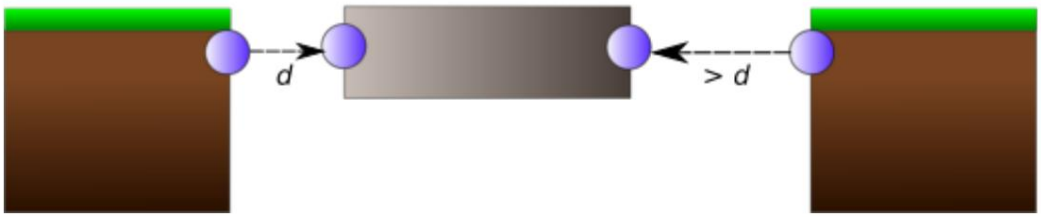
Muitas vezes acontece em videogames que queremos ter múltiplas restrições operando nos mesmos objetos. Para um exemplo rápido de 'jogo', pense em uma pequena ponte de corda, com tábuas interligadas por restrições de distância, como na imagem abaixo:

Se alguma força for aplicada sobre um dos elos, o resto será puxado pelas restrições, criando uma ponte de corda adequadamente instável, pelo menos em teoria. O que é mais provável de acontecer é que, mesmo sem quaisquer forças aplicadas diretamente, as ligações da ponte oscilem de qualquer maneira, e podem até começar a aumentar a velocidade de uma forma irrealista. Para entender por que isso acontece, precisamos pensar em como nossas restrições são satisfeitas. Se tivermos uma lista de restrições e resolvermos as forças de restrição para cada uma separadamente, nossa ponte de corda se comportará mal:

Update 0:



Update 1:



Se deixarmos a primeira restrição de distância calcular seu resultado, as peças da ponte se movem para a esquerda, fazendo com que a restrição à direita seja violada. Se calcularmos as restrições ao contrário, violaremos a restrição esquerda. O que fazemos para minimizar o erro em nossas duas restrições?

Para integrar totalmente as restrições em um mecanismo físico, não apenas a força de restrição deve ser determinada para uma restrição, mas para todo um sistema de restrições simultaneamente.

Calculandoÿ

Ao lidar com interações entre nossos corpos rígidos, vimos em tutoriais anteriores como isso pode ser alcançado modificando a posição de um corpo, sua primeira derivada de posição (velocidade) ou sua segunda derivada de posição (aceleração). Portanto, para satisfazer as nossas restrições, podemos fazer o mesmo (juntamente com a orientação e suas derivadas, é claro). Geralmente, porém, as restrições são resolvidas com mudanças na velocidade, pela mesma razão que a resolução de colisão que vimos anteriormente - podemos facilmente conservar o momento e convergimos rapidamente para uma resposta correta (as molas podem deixar uma restrição permanecer violada se não forem codificados corretamente para seu valor k, por exemplo). Para resolver as nossas restrições, geralmente determinamos quais impulsos aplicar aos nossos objetos restritos - um impulso é apenas uma força ao longo do tempo, por isso é facilmente mapeado na nossa ideia de uma força de restrição. Sabendo disso, resolvendo

qualquer restrição ÿ serve apenas para calcular o impulso correto para acomodar a massa dos objetos, na direção correta. Nossa matriz Jacobiana já contém a direção correta, então nosso multiplicador de Lagrange precisa trabalhar apenas nas velocidades lineares e angulares do objeto e em quanto essas velocidades são mapeadas para resolver a restrição, dividida pela massa da restrição.

Para a nossa restrição de distância, onde apenas as massas dos objetos importam (em vez do momento de inércia em um ponto específico), obtemos um ÿ final assim:

$$ÿ = \left(\frac{1}{mãe^{ÿ1} + mb^{ÿ1}}\right)J \cdot v$$

A multiplicação do nosso vetor objeto v e da nossa matriz Jacobiana v nos dá a velocidade relativa ao longo do eixo que resolverá a restrição (para uma restrição de distância, esse será o vetor de direção entre os objetos), e pela escala por massa , obtemos um valor ÿ resultante que, quando aplicado nessa direção como um impulso (que então leva em consideração a massa inversa), obtemos uma resposta física correta e precisa que conserva o momento - você deve se lembrar de adicionar as massas dos objetos a partir de quando a resolução da colisão foi adicionado à base de código em um tutorial anterior, e isso não é coincidência, já que esse impulso pode ser visto como uma restrição no valor de um único quadro entre os dois objetos.

Podemos calcular o mesmo resultado para ÿ formando uma matriz de massa para os corpos rígidos com restrição da seguinte forma (assumindo que a restrição leva em consideração a velocidade linear e angular):

$$Mÿ1 = \begin{bmatrix} mãe^{ÿ1} & ÿ \\ ÿ & ÿ \\ ÿ1mb\ ÿ1 & \\ I\ a & \\ ÿ\ Ib^{ÿ1} & ÿ \end{bmatrix}$$

A partir daí, podemos determinar ÿ da seguinte forma:

$$ÿ = \left(\frac{1}{JM^{-1}J^{T}}\right)J \cdot v$$

Isto nos permite codificar todas as propriedades de nossas restrições como um conjunto de multiplicações de matrizes em uma forma genérica - contanto que formemos o Jacobiano correto e dimensionemos o resultado pela matriz de massa correta, nossa força de restrição resultante será suficiente. magnitude para levar nossos objetos a um estado de satisfação de restrição.

Solucionadores globais vs iterativos

Grosso modo, agora sabemos como resolver qualquer restrição única - dimensionamos seu Jacobiano pela massa da restrição para determinar alguns impulsos a serem aplicados aos nossos objetos. Mas e quando há múltiplas restrições? No exemplo anterior, vimos como a resolução de qualquer restrição viola a outra; portanto, uma solução é calcular todas as restrições juntas em um único cálculo grande. Para fazer isso, precisaríamos de um vetor v expandido contendo todas as informações do nosso corpo rígido de restrição para as restrições de 0 a n, e uma matriz nxn maior de todas as restrições Jacobianas na diagonal.

Até aí tudo bem - ao codificar restrições na forma de matriz Jacobiana, elas são facilmente combináveis em matrizes maiores. No entanto, cada restrição pode invalidar outra, por isso não são totalmente independentes - na forma de matriz, isso significa que não temos apenas uma matriz diagonal de Jacobianos, mas um mapeamento das mudanças dessas combinações Jacobianas - mais derivadas! Além disso, à medida que o número de restrições aumenta, o tamanho desta matriz Jacobiana combinada aumenta ao quadrado do aumento; como cada Jacobiano é em si uma matriz com um número de elementos, pode rapidamente acontecer que uma matriz combinada tenha milhares de entradas, muitas das quais são apenas zeros (para restrições que não afetam umas às outras), mas que ainda precisam em processamento. Juntos, isso é conhecido como um solucionador global e, embora produza a resposta correta, não pode fazê-lo rapidamente e nem mesmo é garantido que exista uma solução - se duas restrições de distância disserem que o objeto B precisa ser 20 unidades do objeto A ou B, mas A e B estão separados por 5.000 unidades, essas restrições nunca serão satisfeitas de qualquer maneira, e tentar resolvê-las como parte de um todo maior fará com que outras restrições dependentes dêem errado junto com elas.

Em vez de um solucionador global, a maioria dos sistemas de física de videogame usará um solucionador iterativo para suas restrições. Em um solucionador iterativo, cada restrição é repetida e resolvida por si só, com os resultados realimentados para a próxima iteração do solucionador, de modo que, ao longo de múltiplas iterações, as restrições que influenciam vários objetos convergem para que todas estejam tão próximas de serem satisfeitas que possível. Isto tem a vantagem de permitir que cada restrição seja calculada da maneira mais simples possível, em vez de inserir valores genéricos em um solucionador global - por exemplo, em nossa restrição de distância podemos usar multiplicação vetorial em vez de multiplicação matricial se soubermos que a velocidade em um eixo não tem influência na velocidade de outros eixos e nunca influenciará nenhuma outra restrição diretamente (apenas indiretamente por meio do processo iterativo).

Como os resultados da nossa solução de restrições envolvem cálculos de impulsos, que alteram as velocidades dos corpos rígidos, os cálculos que cada iteração do solucionador faz devem ser baseados na velocidade - como veremos no código mais adiante, isso não há problema, já que nossas restrições operam na velocidade de qualquer maneira. A forma mais comum de solucionador iterativo usado em motores de física é conhecida como Impulso Sequencial, que por sua vez é uma forma de Gauss-Seidel Projetado. Seja qual for o nome que chamamos de nosso solucionador iterativo, a forma geral é a mesma:

```
1 while (! terminou) { para ( cada
2         restrição no sistema) {
3             Calcular Jacobiano
4             Calcular lambda
5             Aplique o impulso resultante
6         }
7}
```

Pseudocódigo do solucionador iterativo

Como cada iteração resulta em impulsos diferentes, a próxima iteração do solucionador vê diferentes velocidades relativas (seja da iteração anterior da mesma restrição, ou como resultado de alguma outra restrição operando nos mesmos objetos) e, portanto, diferentes maneiras pelas quais a restrição está sendo violada, de modo que após uma série de iterações, os objetos são empurrados para uma resposta aceitável para sua velocidade linear e angular. A condição de saída para o loop acima pode ser um simples número máximo de iterações, ou cada restrição pode se tornar inativa para aquele quadro após um determinado limite de proximidade ser atingido - um número máximo de iterações é recomendado pelo mesmo motivo que o exemplo do solucionador global acima, na medida em que pode haver algumas restrições que simplesmente não podem ser satisfeitas por qualquer motivo.

Adicionando preconceito

Embora a maioria das restrições sejam resolvidas através da sua primeira derivada, a solução pode não mover a restrição tão perto de ser satisfeita. Por exemplo, os impulsos aplicados para manter unida uma restrição de distância podem ser lentamente vencidos pela gravidade, resultando na violação da restrição ao longo do tempo. A solução para isso é de alguma forma distorcer os resultados do cálculo da restrição:

$$C\ddot{y} = J \cdot v + b$$

Um método comum de adicionar viés é a estabilização Baumgarte, que assume a seguinte forma:

$$b = \frac{\ddot{y}}{dtC}$$

Onde ÿ é um fator de polarização dentro do intervalo de 0 a 1 (geralmente muito próximo de 0). Isso pegará o valor atual pelo qual nossa restrição C está sendo violada e o adicionará aos nossos componentes de v, inflando-os artificialmente e, assim, fazendo com que os impulsos resultantes sejam maiores quanto mais a restrição for violada. Ao adicionar isso a cada iteração do solucionador iterativo, espera-se que qualquer erro que ocorra devido a forças externas ou instabilidade numérica seja minimizado e a correção do sistema como um todo seja mantida tanto quanto possível.

Código do Tutorial

Para ver uma demonstração prática das restrições em ação, faremos um caso de uso típico para um
restrição persistente - vamos fazer uma simples 'ponte de corda' de cubos, conectada com a distância
restrições. A capacidade de adicionar restrições ao mundo do jogo já está presente na base de código - o
A classe GameWorld armazena uma coleção de ponteiros para objetos Constraint, que são declarados da seguinte forma:

```
espaço para nome NCL {
        espaço para nome CSC8503 {
                restrição de classe {
                público :
                        Limitação () {}
                        ~ Restrição () {}

                        virtual void UpdateConstraint ( float dt) = 0;
                };
        }
}
```

Cabeçalho da classe de restrição

Não há muito que possamos dizer sobre o que uma restrição faz, a não ser que, com o tempo, ela tentará
e aplicar impulsos para tentar satisfazer seus requisitos, então tudo o que a classe base possui é um UpdateConstraint
método, que considera o intervalo de tempo do quadro (ou, como veremos mais tarde, uma subdivisão desse intervalo de tempo para
permitir que a restrição itere várias vezes, como um solucionador iterativo).

Para realmente criar uma implementação de uma restrição, vamos criar uma nova subclasse, Posi-tionConstraint, com um
arquivo de cabeçalho parecido com este:

```
# pragma uma vez
# incluir "Restrição .h"

espaço para nome NCL {
        espaço para nome CSC8503 {
                classe GameObject;

                classe PositionConstraint: restrição pública {
                público :
                        PositionConstraint ( GameObject * a , GameObject * b objectA , flutuante d) {
                                objectB = uma;
                                distância =b;
                                =d;
                        }
                        ~ PositionConstraint() {}

                        void UpdateConstraint( float dt) substituição;

                protegido :
                        GameObject *objetoA;
                        GameObject *objetoB;

                        distância de flutuação ;
                };
        }
}
```

Cabeçalho da classe de restrição

Isso nos permitirá representar uma restrição que tenta manter uma distância definida de d entre dois GameObjects a e b. O funcionamento da restrição está inteiramente dentro do método de substituição UpdateCon-straint:

```
1 void PositionConstraint :: UpdateConstraint ( float dt ) 2 Vector3 relativoPos =                    {

3 objectA -> GetConstTransform(). ObterPosiçãoMundo() -
4objetoB ->GetConstTransform(). GetWorldPosition();
5
6       float distância atual = relativoPos. Comprimento ();
7
8       deslocamento flutuante = distância - distância atual;
```

Método PositionConstraint::UpdateConstraint

A primeira coisa que precisamos fazer é descobrir se os dois objetos estão violando as regras da restrição ou não. Como esta é apenas uma restrição de distância, tudo o que precisamos verificar é a sua posição relativa a cada outro e, em seguida, calcule o quão longe isso está de corresponder à variável de distância predefinida da restrição (linha 8).

No caso em que os objetos estão muito próximos ou muito distantes (qualquer forma quebra esta restrição, então podemos imaginar a restrição como sendo uma haste sólida conectando nossos objetos), precisamos descobrir como muitas das velocidades dos dois objetos estão trabalhando contra a satisfação da restrição (isto é, quão grande parte da velocidade está resultando em uma mudança na restrição, que neste caso será quase qualquer velocidade relativa) e use isso como um fator de escala para as forças corretivas (linha 24). Para tentar vir Para encontrar uma solução para o multiplicador de lagrange, também adicionaremos uma pequena quantidade de viés, usando o método de estabilização Baumgarte descrito anteriormente. O biasFactor na linha 26 é uma variável ajustável - diferentes restrições podem exigir diferentes quantidades de polarização adicionadas para tornar sua restrição estável.

Em seguida, apenas adicionamos nossa tendência à velocidade efetiva e dividimos a resposta pela massa de restrição (linha 31) - dividindo pela massa (inversa), calculamos uma quantidade total correta de força para tentar aplicar, pois o método ApplyLinearImpulse dividirá o valor novamente pelo valor real desse objeto massa. Isto significa que a nossa restrição ainda obedece às leis do movimento - há uma força igual e oposta aplicado a ambos os objetos, e o momento será conservado.

Observe que nunca criamos a matriz Jacobiana diretamente, apenas construímos o equivalente vetores e dimensioná-los por lambda. Nossa restrição de distância deve ter derivadas parciais de posição equivalente a vetores que aproximam as posições (o vetor normalizado entre os dois objetos), e enquanto tentamos resolver a restrição usando velocidade, aplicamos impulsos, que ao longo no curso do solucionador iterativo, trabalhamos para satisfazer nossa restrição de maneira realista.

```
10      if (abs (deslocamento) > 0,0 f) {
11            Vector3 offsetDir = relativoPos . Normalizado ();
12
13            PhysicsObject * physA = objectA -> GetPhysicsObject ();
14            PhysicsObject * physB = objectB -> GetPhysicsObject ();
15
16            Vector3 relativaVelocity = physA -> GetLinearVelocity () -
17                                                      physB -> GetLinearVelocity();
18
19            float restriçãoMass = physA -> GetInverseMass () +
20                                                  physB -> GetInverseMass();
21
22            if (massa de restrição > 0,0 f) {
23      // quanto de sua força relativa está afetando a restrição
24                  float velocidadeDot = Vector3 :: Dot (relativeVelocity, offsetDir);
```

Método PositionConstraint::UpdateConstraint

```
                float BiasFactor = 0,01 f;
                tendência de flutuação = -(fator de polarização /dt) * deslocamento;

                float lambda = -(velocidadeDot + polarização)/restriçãoMass;

                Vetor3 aImpulso = deslocamentoDir * lambda;
                Vetor3 bImpulso = -offDir * lambda;

                physA -> ApplyLinearImpulse (aImpulse); // multiplicado pela massa aqui
                physB -> ApplyLinearImpulse ( bImpulse ); // multiplicado pela massa aqui
            }
        }
}
```

Método PositionConstraint::UpdateConstraint

Classe TutorialGame

Para demonstrar nossa nova restrição, faremos o clássico teste de restrição de distância - uma corda ponte. Para fazer isso teremos dois corpos rígidos fixados em posição usando uma massa inversa de 0, e têm uma cadeia de corpos rígidos de adição de cubos entre eles, cada um ligado a uma restrição de distância. O código para isso vamos adicionar ao método BridgeConstraintTest vazio - isso pode ser chamado dentro do InitWorld muito parecido com o InitCubeGridWorld que criou o primeiro ambiente de física que vimos no tutorial 1. Esta é a aparência do código:

```
TutorialGame nulo :: BridgeConstraintTest () {
    Vector3 cubeSize = Vector3 (8 , 8, 8);

    flutuador invCubeMass = 5; // quão pesadas são as peças do meio
    interno numLinks = 10;
    flutuador distânciamáx = 30; //distância de restrição
    flutuador distância do cubo = 20; //distância entre links

    Vetor3 startPos = Vetor3 (500 , 500 , 500);

    GameObject * start = AddCubeToWorld (startPos + Vector3 (0, 0); , 0 , 0)
                                , tamanho do cubo
    GameObject * end = AddCubeToWorld ( startPos + Vector3 (( numLinks + 2)
                                * distância do cubo , 0 , 0) , tamanho do cubo , 0);

    GameObject * anterior = início;

    for (int i = 0; i < numLinks; ++ i) {
        GameObject * bloco = AddCubeToWorld ( startPos + Vector3 (( i + 1) *
                                distância do cubo , 0 , 0) , cubeSize invCubeMass );
        PositionConstraint * restrição = novo PositionConstraint (anterior,
                                bloquear , distânciamáx);
        mundo -> AddConstraint (restrição);
        anterior = bloco;
    }
    PositionConstraint * restrição = new PositionConstraint (anterior,
                                fim , distânciamáx);
    mundo -> AddConstraint (restrição);
}
```

Método TutorialGame::BridgeConstraintTest()

Começamos fazendo nossos objetos fixos com massa infinita (linha 11 e 13), e depois adicionamos progressivamente mais objetos entre eles (linha 18) - usando o ponteiro anterior sempre conhecemos o outro objeto ao qual conectar nosso bloco recém-adicionado. Tudo o que precisamos fazer é garantir que nossa última caixa fique restrita ao outro lado (linha 26), ou nossa ponte de corda simplesmente cairá devido à gravidade - tudo bem se estivermos fazendo a corda balançando em Pitfall, mas por um ponte adequada, precisaremos conectá-la corretamente.

O mecanismo de atualização de cada restrição já está presente no método PhysicsSystem::Update, assim:

```
1 float restriçãoDt = iterationDt / ( float ) restriçãoIterationCount; 2 for ( int i = 0; i < restriçãoIterationCount ; ++ i ) { 3
4 }
        UpdateConstraints (restriçãoDt);

5 IntegrateVelocity ( iteraçãoDt );
```

Método PhysicsSystem::Update()

A variável restriçãoIterationCount começa a ser definida como 10, mas é fácil de alterar rapidamente com a adição de algumas verificações de teclado - você deve descobrir que mais iterações resultam em uma cadeia de restrições geral mais estável, ao custo de mais computação.

Classe GameWorld

A classe GameWorld precisa de um pequeno ajuste - quando o mundo é limpo, as restrições também devem ser removidas, e quando apagadas também devem ser deletadas:

```
6 void GameWorld :: Clear () { gameObjects .
7         claro (); restrições . claro (); //
8         nova linha !
9 }
10
11 void GameWorld :: ClearAndErase () { for ( auto & i:
12         gameObjects) { delete i;
13
14         }
15         for ( auto & i: restrições) { delete i; // novo loop
16                 for !
17         }
18         Claro ();
19 }
```

Método PhysicsSystem::Update()

Conclusão

Se você conseguiu fazer tudo corretamente, deverá encontrar uma linha de cubos em seu mundo de jogo. Habilitar a gravidade deve fazer com que a linha caia no meio e se estabilize em forma de 'U' - as extremidades são fixadas no lugar devido à sua massa inversa de 0, mas os elementos do meio estão livres para cair sob a gravidade, até que sejam mantidos juntos pelas restrições adicionando impulsos corretivos.

As restrições são os mecanismos internos pelos quais os motores físicos mantêm a estabilidade do mundo do jogo e nos permitem definir outras relações entre os nossos objetos, para que eles não necessariamente apenas colidam, mas de alguma forma causem mudanças na posição e orientação uns dos outros. ao longo do tempo. De certa forma, temos trabalhado com restrições desde o tutorial sobre resolução de colisões, mas agora vimos o esqueleto de uma estrutura geral para restrições e como elas podem ser usadas para definir mudanças nos corpos rígidos do nosso mundo. Também aprendemos um pouco sobre solucionadores - principalmente que os solucionadores globais são difíceis e os solucionadores iterativos precisam de um pouco de ajuda por meio da estabilização Baumgarte para manter o mundo em uma situação corret

estado. Com a matriz Jacobiana, a relação entre a posição e a orientação de dois objetos pode ser representada e, a partir daí, podemos apenas resolvê-los como parte do processo iterativo geral.

Trabalho adicional

1) O código até agora fornece apenas um único tipo de 'restrição de posição'. Tente adicionar uma restrição adicional que também restrinja a orientação dos dois objetos anexados. Ao fazer isso, você poderá limitar a capacidade de girar livremente em torno dos cubos, adicionando forças a eles; o torque será distribuído por toda a ponte de corda.

2) Outra variação de restrição pode ser algo que adiciona força e torque propositalmente, como um efeito motor. Isto deve ser conseguido como uma ligeira modificação da restrição de posição (para movimento linear) ou da restrição de orientação acima (para movimento angular) para modificar a verificação de distância para adicionar um deslocamento e calcular um novo valor normal para onde o o objeto deve estar se movendo para / voltado para.

3) Além das restrições persistentes, também é possível criar restrições temporárias que cumpram algum propósito e sejam então removidas. Podemos usar isso para fornecer uma restrição de resolução de colisão, que substitui o método ImpulseResolveCollision atual pela adição de um ResolutionConstraint ao PhysicsSystem. Você precisará adicionar algum método para determinar quando uma restrição deve ser removida do sistema (neste caso, quando não houver mais colisão entre os objetos) e uma nova classe para representar uma restrição que calculará o valor correto. impulsos para separar os objetos de modo que o momento permaneça conservado.