Física - Movimento Linear

Introdução

Os videogames modernos podem ter centenas de objetos se movendo e interagindo na tela de maneira verossímil. No centro de qualquer jogo com movimento realista de objetos está um mecanismo de física, composto pelas funções e classes necessárias para modelar as interações complexas entre objetos de maneira realista e em tempo real. Em sua essência, um mecanismo de física deve fazer duas coisas: deve permitir que os objetos se movam de maneira fisicamente realista à medida que as forças são exercidas sobre eles e deve detectar quando ocorrem colisões entre esses objetos e permitir que esses objetos reajam de acordo.

Neste tutorial, daremos uma olhada nos princípios básicos para ativar o movimento realista de objetos e aprenderemos como adicionar conceitos como aceleração, velocidade e massa em nossas simulações de jogos.

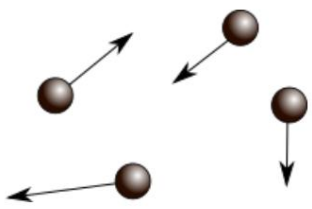
Órgãos de Física

As cenas nas quais os videogames são construídos podem conter muitas centenas de objetos, com muitas considerações a serem feitas quanto ao nível de precisão exigido em sua replicação do “mundo real”. Em termos gerais, existem dois factores que influenciam as propriedades físicas de um objecto – se podem causar interacções físicas e se reagem a interacções físicas. As paredes em um jogo FPS típico podem ser colididas, evitando que o jogador passe por elas como um fantasma - as paredes fazem com que uma colisão seja acionada em algum lugar no código do jogo, que em algum momento será resolvida (empurrando o jogador para trás , e fazendo-os deslizar ao longo dele). No entanto, geralmente não reagir a nenhuma interação, exceto talvez graficamente - esbarrar naquela parede não transmite nenhuma força sobre a parede para empurrá-la levemente, e mesmo se você atirar nela e causar buracos de bala em sua superfície, aqueles são apenas efeitos gráficos; as propriedades físicas da parede não foram alteradas.

Em termos de mecanismo de jogo, o exemplo de parede acima pode ter um volume de colisão (como os AABBs e esferas que analisamos para fazer raycasting) para detectar que o jogador a cruzou, mas carece de um corpo físico, que modele como as forças devem ser aplicadas a ele. um objeto e como ele deve reagir quando ocorrerem colisões. Esses corpos físicos podem ser amplamente categorizados como uma de três coisas – partículas, corpos rígidos e corpos moles, cada um dos quais pode ser usado em um jogo para simular diferentes tipos de objetos.

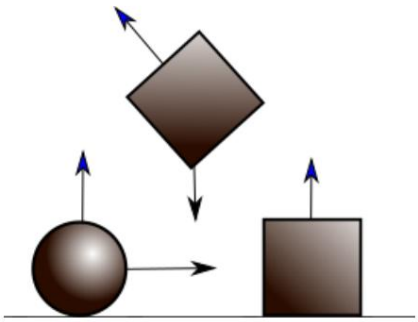
Partículas

A forma mais simples de representação física é a partícula. Uma partícula tem uma posição no espaço e pode ter velocidade e acelerações aplicadas a ela. Embora uma partícula possa ter massa, ela não tem orientação; é apenas um ponto no espaço que representa a presença de 'alguma coisa'. Isto pode ser visto como uma extensão dos “efeitos de partículas” frequentemente usados em videogames para representar visualmente coisas como fogo e fumaça – cada “partícula” é uma pequena textura no espaço mundial que pode se mover, mas sua orientação não importa particularmente. à simulação geral da qual faz parte. Às vezes, os sistemas de partículas empregam cálculos físicos para melhorar seu realismo - os efeitos das faíscas de um fogo brilhante podem ricochetear nas superfícies e as partículas de água emitidas por uma fonte de água podem eventualmente ser puxadas para baixo pelo efeito da gravidade.



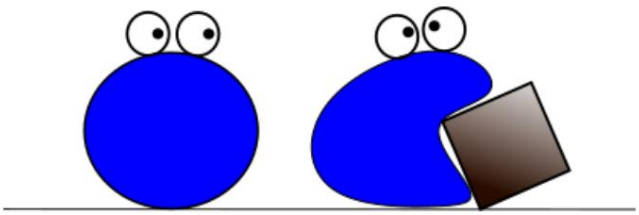
Corpos rígidos

Um passo acima de uma partícula é um corpo rígido - em vez de um conjunto de partículas construindo uma forma, este pode ser visto como um único corpo que tem volume e orientação (e como veremos mais tarde, a capacidade de mudar sua orientação por meio de forças). Podemos assumir que os corpos rígidos são as formas sólidas que constituem a maioria dos objetos no mundo - podemos determinar o volume e a orientação de uma xícara com a mesma facilidade com que o fazemos com uma nave espacial. Embora o copo (ou nave espacial!) seja realmente composto por múltiplos elementos (a tampa, o próprio copo e a capa de papelão), e além desses múltiplos átomos, podemos assumir que as posições entre eles permanecem fixas e, portanto, tratá-los todos como parte de um único todo. Como as forças interagem com um corpo rígido, elas não mudam de forma nem alteram de forma alguma a distribuição de sua massa ao longo de seu volume, elas têm uma forma totalmente 'fixa'.

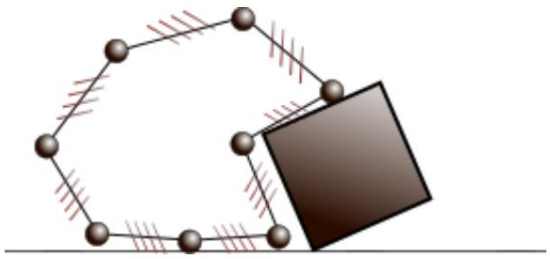


Corpos Suaves

A maioria dos objetos em nossos jogos podem ser considerados corpos rígidos - eles nunca mudam de forma, exceto talvez por meio de animações. Alguns objetos exigem dinâmicas mais precisas fisicamente para modelar completamente seu movimento no mundo - pense na capa de um super-herói balançando ao vento, ou talvez em um personagem do jogador feito de massa mole que pode esticar e esmagar de acordo com a entrada do jogador:



Para criar efeitos como este, precisamos de um corpo macio, em vez de rígido. Em vez de uma única forma sólida para representar as propriedades físicas do nosso objeto, podemos tratar o objeto como um conjunto de pontos, conectados por meio de molas:



Como cada um dos pontos tem forças aplicadas a ele (seja por colisão com o mundo de alguma forma, ou aplicação direta de força usando o controlador do joypad), as molas se esticam e contraem, espalhando essa força e formando a forma como um todo. mover ou 'esmagar'. Se imaginarmos que esses 'pontos' são os vértices de uma malha, é fácil visualizar como mover os pontos de uma forma fisicamente consistente criaria um efeito de ondulação de bandeira ou um efeito elástico/mole. No entanto, fazer com que esses pontos interajam corretamente com o mundo ao seu redor é muito mais difícil do que com corpos rígidos - em vez de uma forma simples como um cubo ou uma esfera (ambos têm métodos fáceis de determinar se algo os está tocando, como veremos veja mais adiante no tutorial), corpos moles exigem muitas verificações para ver quais pontos estão sendo empurrados por objetos no mundo. O efeito das molas também é complicado; embora as forças aplicadas em um ponto devam interagir com os pontos conectados de maneira consistente, uma pequena quantidade de erros de ponto flutuante pode fazer com que os corpos moles 'balançem' mesmo quando deveriam estar em repouso, e se uma quantidade suficiente dessa energia oscilante se acumular, o corpo mole pode deformar-se ou mesmo quebrar-se completamente.

Termos de Física

Nesta série de tutoriais, encontraremos vários termos que você pode ou não ter encontrado.
Por precaução, examinaremos alguns desses termos aqui, para resolver qualquer ambigüidade.

Deslocamento

No módulo gráfico, nos familiarizamos com a mudança da posição dos objetos no mundo no espaço mundial.
Em alguma literatura relacionada a cálculos físicos, você verá a posição às vezes chamada de deslocamento, onde se refere especificamente a quanto o objeto se moveu de sua posição inicial ao longo do tempo, e geralmente é armazenado como um vetor, com o deslocamento em cada eixo armazenado usando metros como unidade padrão de medida.
Em quaisquer cálculos relacionados à derivação de valores, usaremos p para denotar a posição; na mecânica clássica, o vetor de deslocamento real (ou o vetor de posição do nosso objeto, como estamos acostumados a pensar nele no módulo anterior) é adicionalmente denotado por s, do latim spatium.

Velocidade

A primeira derivada do deslocamento do objeto em relação ao tempo (ou seja, o quanto a posição do objeto muda ao longo do tempo) é conhecida como velocidade. Tal como acontece com o deslocamento, é um vetor 3D – nossos objetos se movem em uma direção específica (a direção do vetor velocidade), a uma determinada velocidade (a magnitude do vetor velocidade). Mudanças no vetor velocidade de um corpo físico significam que ele está desacelerando (a magnitude está se aproximando de zero) ou acelerando (a magnitude está ficando maior). Normalmente descreveríamos a magnitude da velocidade de um objeto usando metros por segundo como unidade de medida - após 1 segundo, um objeto com velocidade de x terá se movido no mundo x metros, na direção de ˆx; às vezes você também verá isso declarado como sendo uma velocidade de xm/s ou msÿ1 . Nos cálculos de física, a velocidade de um objeto é frequentemente denotada por ÿp quando se discute mudanças na posição de um objeto em geral, ou às vezes apenas por v.

Aceleração

A segunda derivada da posição de um corpo em relação ao tempo é conhecida como aceleração (ou, dito de outra forma, a aceleração mede a taxa de variação da velocidade). Nos cálculos de física, é muitas vezes denotado como ¨p, ou menos comumente ÿv, quando se raciocina especificamente sobre taxas de mudança, e muitas vezes apenas como quando aparece em outros cálculos. Num carro, pressionamos o pedal do acelerador para acelerar (dentro dos limites legais de velocidade, é claro!), fazendo com que o motor trabalhe mais; esse trabalho altera a velocidade do carro e, portanto, sua posição ao longo do tempo.
A aceleração é geralmente medida em metros por segundo por segundo, muitas vezes denotada como m/s2 ou, alternativamente, msÿ2 .

O exemplo mais popular de aceleração é provavelmente a gravidade da Terra, que você provavelmente conhece como 9,8 m/s2 . Isso significa que os corpos físicos se movem para baixo a uma taxa crescente de 9,8 metros por segundo por segundo - ou seja, a cada segundo, a taxa de variação da velocidade terá aumentado 9,8 metros por segundo, então o corpo acelera e, portanto, muda de posição. cada vez mais rápido (até atingir sua velocidade terminal - quando a desaceleração do 'arrasto' causada pelo impacto no ar se iguala à força de aceleração, cancelando-a).

A diferença entre aceleração e velocidade é importante - avançar a uma taxa constante com uma velocidade de 9,8 m/s é bastante rápido e você completaria uma corrida de 100 metros em pouco mais de 10 segundos; acelerar a uma taxa constante de 9,8 m/s2 significará que após 10 segundos você estará se movendo a 98 metros por segundo e lhe dará uma medalha de ouro nas Olimpíadas.

Força

Para se mover, um objeto deve ter uma força aplicada a ele. Esta força terá uma direção e uma magnitude e, portanto, pode ser representada no código como um vetor. A unidade de medida de força é o Newton, sendo um Newton a força necessária para mover um peso de 1 kg a uma taxa de 1 metro por segundo por segundo. Portanto, uma força é um ajuste à aceleração de um corpo rígido.

Massa

Massa é a medida da quantidade de matéria de que um objeto é feito e é medida em quilogramas. A matéria resiste às forças, portanto, quanto mais massa um objeto tiver, mais difícil será movê-lo. Não é o mesmo que pesa aquele objeto, embora os dois estejam relacionados. Peso é a quantidade de força que um objeto aplica sob a gravidade - uma massa de 1kg aplica 9,8 Newtons de força na Terra devido à gravidade do planeta, mas o mesmo objeto aplica menos força na Lua, pois há menos gravidade para puxá-lo para baixo - então ele 'pesa'. menos'...mesmo que ainda tenha 1kg de massa. É por isso que, embora as coisas não pesem nada no espaço, ainda requerem grandes motores de foguete – para neutralizar a massa do objeto.

Nos motores físicos, todo objeto que desejamos mover e interagir em nossa simulação terá um valor de massa, medido em gramas ou quilogramas (ou algum valor derivado disso). Geralmente, armazenamos o inverso da massa - isto é,. Este valor de massa 'inverso' $\frac{1}{\text{massa}}$ ajuda a transformar algumas operações de divisão em operações de multiplicação, e também
fornece alguns efeitos colaterais úteis que veremos mais tarde.

Momento

Outro termo que você pode encontrar na literatura sobre motores de física é impulso. Este é simplesmente o produto da massa e da velocidade de um objeto:

$$p = mv$$

O momento é medido em kg m/s - simplesmente as unidades de medida de ambas as partes do cálculo do momento. Você deve ter ouvido falar disso em termos de conservação do momento. O que isto se refere é que o momento total de um sistema de corpos físicos deve ser sempre constante - à medida que os objetos colidem, eles transmitem parte do seu momento ao outro objeto, reduzindo o seu próprio. É por isso que quando a bola branca atinge outra bola na sinuca, a bola branca pode parar, mas a outra bola acelera; a quantidade total de momento no sistema foi conservada, mesmo que a soma das velocidades não tenha sido conservada (a bola branca pode ser mais pesada ou mais leve que as outras bolas, mas ainda assim transmitiria um momento tal que o sistema como um todo permanecesse constante) .

Leis do Movimento de Newton

Os fundamentos do movimento em objetos podem ser descritos de forma concisa usando as três leis do movimento de Isaac Newton - tudo o que fazemos mais tarde com objetos em movimento e reagindo a colisões entre objetos está relacionado a eles, portanto, ter alguma compreensão deles será útil à medida que continuamos no tutorial Series.

Primeira Lei de Newton

A primeira lei afirma que um objeto permanece em repouso ou se move com velocidade constante, a menos que seja influenciado por alguma outra força. Para nosso mecanismo de física, isso significa que um objeto não deveria, por padrão, fazer muita coisa! Ele não deve se mover até que algo o atinja ou o próprio objeto exerça alguma força para movê-lo em uma direção.
Assim que o objeto começar a se mover, ele deverá continuar se movendo até que alguma outra força o desacelere ou mude sua direção. Pense em uma nave espacial, flutuando no espaço profundo - não há nada empurrando-a ou exercendo forças sobre ela (exceto uma força gravitacional incrivelmente pequena de corpos astrais próximos, que vamos ignorar por enquanto!), então ela pode permanecer imóvel. . Se a espaçonave ligar os motores do foguete um pouco, ela começará a se mover. Ele continuará então se movendo nessa direção, mesmo que os motores sejam desligados novamente, pois não há nada agindo sobre ele para desacelerá-lo. Mesmo girar o navio não mudará a direção ou a velocidade do movimento - até que os motores sejam ligados novamente, de qualquer maneira.

Inércia

A primeira lei de Newton é às vezes conhecida como lei da inércia. A inércia é a resistência de um objeto a uma mudança na sua velocidade e está relacionada à sua massa - assim como é mais difícil mover um objeto mais pesado, também é mais difícil mudar a direção de um objeto quando ele começa a se mover.

Fricção e amortecimento

Na Terra, você provavelmente já percebeu, enquanto andava de bicicleta ou de carro, que se parar de pedalar ou desligar o motor, você não apenas continua andando, mas, em vez disso, para - a menos que esteja descendo uma ladeira! A gravidade exerce uma força sobre tudo que tem massa e, portanto, puxará os objetos morro abaixo, mas em uma superfície nivelada, pararíamos devido ao atrito - os pneus da bicicleta ou do carro raspam na estrada, adicionando alguma força, e o chassi do carro/bicicleta também fica lento pela resistência do ar – a força de atrito causada pela tentativa de movimento no ar. Calcular a quantidade de atrito (do pneu ou do ar) sobre um objeto é um problema computacionalmente complicado, por isso geralmente não fazemos isso em um mecanismo de física em tempo real para jogos. Em vez disso, modelamos o atrito amortecendo ligeiramente a velocidade a cada atualização, multiplicando-a por um valor escalar - desde que esse valor seja ligeiramente menor que 1,0, fará com que os objetos percam lentamente sua velocidade, como se estivessem sendo desacelerados pelo atrito.

Segunda Lei de Newton

A segunda lei afirma que a soma das forças que atuam sobre um objeto é igual à massa desse objeto multiplicada pela aceleração do objeto, ou:

$$F = mãe$$

Em um mecanismo de física, geralmente aplicamos forças a objetos no código, em vez de aplicar diretamente uma aceleração. Poderíamos dizer que o personagem saltando exerce uma força de 1000 Newtons (a unidade de medida de força vem desta lei!), por exemplo. Essa força seria então integrada à velocidade e à posição do objeto do nosso jogador ao longo do tempo. Portanto, é melhor reorganizar a equação assim:

$$\text{uma} = \frac{F}{\text{eu}}$$

Desta forma, podemos pensar no cálculo da nova aceleração de um objeto como sendo a soma de todas as forças aplicadas, dividida pela massa. Lembra-se de quando foi mencionado anteriormente que em motores físicos geralmente lidamos com massa inversa? Esta equação é uma boa explicação do porquê. Em vez de dividir por m, podemos multiplicar pelo inverso da massa:

$$\text{uma} = \text{Fmÿ1}$$

A divisão geralmente é uma operação mais lenta que a multiplicação (ou seja, leva mais ciclos de CPU); portanto, ao fazer isso, aceleramos nossos cálculos físicos. Mas há outro benefício! Como foi mencionado anteriormente, a massa inversa permite adicionar mais facilmente objetos que não queremos mover em nossos jogos, simplesmente definindo uma massa inversa de 0,0:

$$\text{uma} = 0 = F \cdot 0$$

Desta forma, não importa quanta força seja aplicada ao objeto (de coisas que colidem com ele, etc.), ele não se moverá, pois a aceleração resultante sobre ele também acabará sendo zero. Isto é útil para coisas como o 'chão' do nível, onde queremos que o jogador seja capaz de ficar de pé e pular sobre o objeto (exigindo algumas interações físicas para resolver qualquer detecção de colisão), mas realmente não queremos que o andar para se mover - imagine se os níveis de Mario fossem ligeiramente empurrados para baixo cada vez que Mario saltava!

Terceira Lei de Newton

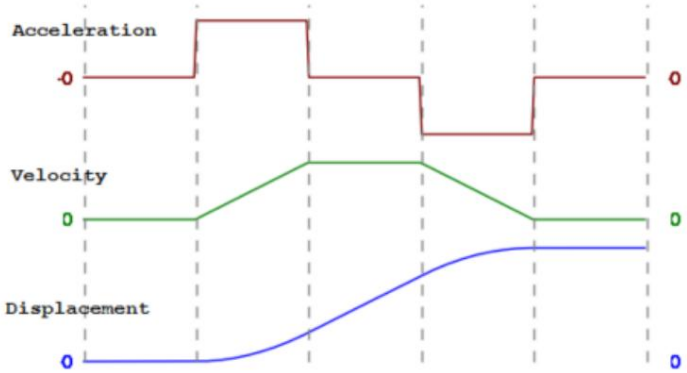
No exemplo do Mario acima, você deve estar se perguntando por que o chão se moveria sempre que Mario saltasse, se não fosse pela massa “infinita” proporcionada pelo uso da massa inversa em nossos cálculos. Isto se deve à terceira lei do movimento de Newton, que afirma que quando um objeto exerce força sobre o segundo objeto, o segundo objeto exerce simultaneamente uma força igual na direção oposta; isso é mais comumente afirmado como - para cada ação, há uma reação igual e oposta.

De modo geral, isso significa em nosso mecanismo de física que sempre que ocorre uma colisão, ambos os objetos acabam recebendo forças - em um jogo de Snooker, se a bola branca atingir outra bola, a segunda bola recebe alguma força exercida sobre ela pela bola branca, mas a bola branca também desacelera quando uma força igual e oposta é aplicada a ela por meio da segunda bola.

No nosso exemplo do Mario, isto significa que quando ele salta, deveria realmente haver uma força oposta aplicada ao mundo. Na realidade, a interação do salto provavelmente seria codificada para apenas aplicar a força ascendente diretamente em Mario e pular a interação - mas agora pense em quando ele pousará novamente; essa é uma colisão que deve ser detectada e resolvida, e a massa inversa garante que não movimentaremos o mundo quando pousarmos.

Integração numérica

Sabemos que a mudança na posição de um objeto ao longo do tempo é descrita pela propriedade da velocidade, e a taxa de variação da velocidade como aceleração. Para realmente realizar essas mudanças, precisamos fazer mais cálculos e alguma integração. Como você já deve saber, para integrar uma função (o integrando), avaliamos o resultado da função em intervalos cada vez menores para obter uma ideia progressivamente mais precisa da área sob a curva no gráfico da função:



Ao descrever a taxa de variação da velocidade e aceleração, frequentemente vemos as seguintes equações:

$$v = \frac{dp}{dt}$$

$$\text{uma} = \frac{dv}{dt}$$

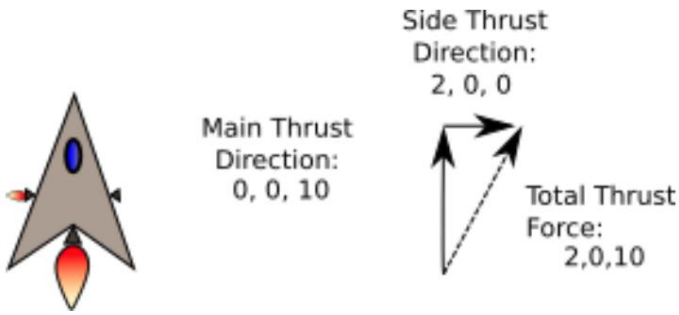
Ao descrever a integração de velocidade e aceleração, usaríamos estas equações:

$$v = adt$$

$$p = vdt$$

Em cada caso, dt é a mudança no tempo. Como estamos lidando com uma simulação de videogame em tempo real composta por uma série discreta de quadros que são renderizados sequencialmente, essa mudança no tempo é razoavelmente óbvia - 1 segundo dividido pela taxa de quadros nos dará o dt para cada quadro.

Em um motor de física, geralmente determinamos a quantidade total de força aplicada a um objeto que enquadra, somando todas as forças individuais que podem estar afetando-o - uma nave espacial que está acionando seus motores principais para avançar, e seu propulsor lateral ao mover-se para a direita, terá aplicada uma força total que deverá fazê-lo mover-se na diagonal:



A partir daí, podemos determinar a quantidade de aceleração que a nave espacial está realizando usando a equação mencionada anteriormente a = Fmÿ1 . Para realmente determinar a nova posição da nave espacial, o valor da aceleração deve ser integrado, para determinar a mudança na velocidade da nave espacial, e a partir daí integrar a velocidade para determinar a mudança na posição da nave no referencial atual.

Há um problema, no entanto. Se a nave estava acelerando a 10msÿ2 após 1 segundo, e 25msÿ2 após 2 segundos, com que rapidez o objeto estava acelerando em 1,5 segundos? Na verdade, apenas a partir desses momentos, não sabemos se a aceleração estava aumentando linearmente ou não, portanto, tratá-las como mudanças discretas pode levar à imprecisão em comparação com as condições da “vida real”. Isso torna muito importante o número de vezes que atualizamos o sistema físico por segundo - quanto mais dividimos cada segundo em quadros discretos (e depois integramos pelo dt desse quadro), mais preciso o sistema físico é capaz de modelar as mudanças na aceleração e velocidade, e mais próximo da resposta 'real' podemos chegar.

Há uma série de métodos de integração numérica comumente usados em motores físicos para determinar a nova posição dos objetos físicos na simulação de cada quadro, cada um dos quais tem características e desvantagens diferentes.

Euler explícito A

integração Euler explícita (às vezes simplesmente 'Integração Euler') é o mais simples dos métodos de integração discutidos. A cada atualização da simulação, determinamos os seguintes novos valores:

$$v_{n+1} = v_n + e_t p_{n+1}$$
$$= p_n + v_n dt$$

Isso afirma que, para os valores do nosso próximo quadro (denotado como n+1), são formados pegando os valores do quadro atual (denotado n) e adicionando a derivada, multiplicada pelo intervalo de tempo. Sabemos implicitamente qual será a aceleração do referencial atual, pois quaisquer forças aplicadas no referencial podem ser somadas e multiplicadas pela massa inversa para obtê-la, como vimos anteriormente.

Euler implícito

A integração explícita de Euler não é perfeita - a velocidade deveria mudar ao longo do intervalo de tempo devido à integração da aceleração, mas estamos tratando isso simplesmente como uma adição constante para alterar a posição. Essa aproximação leva à imprecisão ao longo do tempo, para qualquer objeto com uma velocidade variável ao longo do tempo. Em vez disso, uma integração mais completa calcularia as derivadas no próximo passo de tempo, assim:

$$v_{n+1} = v_n + a_{n+1} dt$$
$$p_{n+1} = p_n + v_{n+1} dt$$

Isso é conhecido como integração de Euler implícita ou, às vezes, integração de 'Euler reversa'. Isso é bom para integrar dados de um conjunto completo de dados, já que a 'próxima' velocidade e aceleração estariam disponíveis para integração. Isso não é realmente bom para uma simulação em tempo real, a menos que saibamos exatamente qual aceleração será aplicada no próximo quadro (talvez esteja aumentando constantemente por uma taxa fixa ou alguma outra função simples), pois nossas interações de jogo poderiam resultar em combinações únicas de colisões e forças em cada quadro. Podemos tentar prever as derivadas tentando ajustá-las às curvas, mas as previsões podem estar incorretas, resultando em imprecisões, o que estamos tentando evitar!

Euler semi-implícito Existe um

meio-termo entre a integração de Euler implícita e explícita, conhecida como integração de Euler semi-implícita (ou às vezes também de Euler simplético). Neste caso, integramos a nossa segunda derivada (aceleração) utilizando o estado atual, permitindo-nos então integrar a nossa primeira derivada (velocidade) com um estado mais atualizado:

$$v_{n+1} = v_n + e_t p_{n+1}$$
$$= p_n + v_{n+1} dt$$

Na prática, isso é tão simples quanto inverter a ordem em que calcularíamos a nova velocidade e posição de um objeto, tornando-o não mais lento do que o cálculo de Euler explícito, mas mais preciso e, portanto, menos provável de resultar em problemas ao longo do tempo. .

Verlet

Se conhecermos a posição atual de um objeto, a posição anterior e o intervalo de tempo entre essas duas medições, podemos determinar a velocidade diretamente, sem precisar armazená-la separadamente. Esta é a base para o método de integração Verlet:

$$p_{n+1} = p_n + (p_n ÿ p_{nÿ1}) + e_{t2}$$

Este método às vezes é usado para sistemas de partículas computados na GPU, pois o custo de reconstrução da velocidade pode ser menor do que o impacto da leitura de outro buffer que contém os dados de velocidade, e o cache resultante perde a leitura desse buffer. A desvantagem da integração Verlet é que temos que fazer alguns cálculos adicionais se quisermos ter um objeto que já esteja se movendo no início da simulação - se não houver posição anterior (ou pior, a variável de posição anterior será padronizada para a origem), então obtemos uma velocidade prevista totalmente imprecisa para o objeto.

Métodos Runge-Kutta

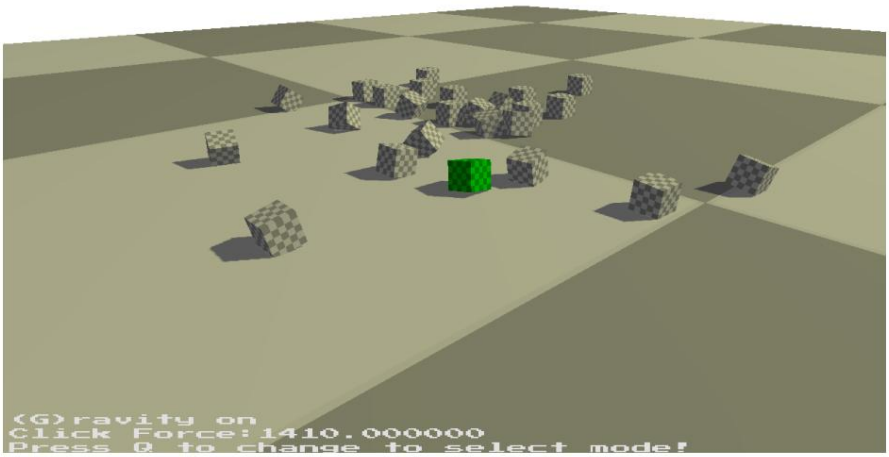
Podemos levar a integração adiante dividindo o timestep e realizando a integração diversas vezes, para tentar e obter uma melhor compreensão de como a velocidade e a posição mudam ao longo do intervalo de tempo. Um tal método de previsão é o método Runge Kutta, que obtém uma média de múltiplas divisões do quadro intervalo de tempo. Diferentes números de passos podem ser dados, levando a resultados ligeiramente diferentes - é sabido que são dados 2 passos como 'RK2' e 4 como 'RK4'.

Usar isso permite maior precisão na posição ao longo do tempo, mas tem o custo da integração de cada objeto agora levando 4 vezes mais cálculos. Nos casos em que a precisão total não é necessária, muitas vezes é melhor gastar esse tempo de cálculo avançando a simulação e elaborando novas colisões e movimentos dessa maneira - Taxas de quadros mais altas costumam ser melhores para a experiência de jogo do que quadros individuais mais “precisos”.

Código do Tutorial

Agora sabemos como a posição, a velocidade e a aceleração estão relacionadas e como as forças aplicadas aos objetos interagem com massa para resultar em acelerações sobre um objeto. Para demonstrar isso, vamos expandir o teste programa criado no tutorial anterior para implementar alguma integração simples de movimento linear em alguns corpos rígidos. Como parte disso, vamos expandir a classe PhysicsObject para permitir a implementação de métodos rígidos. dinâmica corporal, com um integrador Euler semi-implícito, para nos permitir ver como a maioria dos jogos lida com sua física cálculos.

Para demonstrar a dinâmica do corpo rígido e o movimento linear, vamos expandir o tutorial anterior, e não apenas ser capaz de clicar em um objeto, mas também empurrá-lo, aplicando forças nele.



Para fazer isso, precisamos preencher outros métodos atualmente vazios, IntegrateAccel e IntegrateVelocity, que estão ambos na classe PhysicsSystem. Para ver quais funcionalidades esses novos métodos exigirão, vamos dar uma outra olhada na classe PhysicsObject e nas variáveis que ela contém relacionadas ao movimento linear:

Classe FísicaObject

```
1  público :
2  ClearForces vazios ();
3  void AddForce ( const Vector3 & force );
4  Vector3 GetLinearVelocity () const { return linearVelocity ;}
5  protegidos :
6  const CollisionVolume * volume;
7  Transformar * transformar;
8  // Usaremos essas variáveis!
9              flutuador          massa inversa;
10             Vector3 velocidade linear;
11             Força do vetor3;
```

Cabeçalho da classe PhysicsObject

Aplicar uma força é fácil - temos um único método, que utiliza um Vector3 representando uma direção e uma quantidade de Newtons de força para aplicar nessa direção, e adiciona-a à nossa nova variável de força. Um objeto pode ter múltiplas forças aplicadas a ele em um único quadro, portanto, certifique-se de adicionar e não definir esse valor!

```
1 void PhysicsObject :: AddForce ( const Vector3 & addForce ) {
2 força += força adicionada;
3}
```

Método PhysicsObject::AddForce

No final de cada quadro, vamos zerar quaisquer forças, para que elas sejam aplicadas apenas uma vez, usando o comando Método ClearForces:

```
4 void PhysicsObject :: ClearForces () {
5 força =Vetor3();
6}
```

Método PhysicsObject::AddForce

Isto significa que quaisquer forças aplicadas são aplicações instantâneas de n Newtons, em vez de n Newtons. ao longo do tempo.

Classe de Sistema de Física

Na classe PhysicsSystem, precisamos adicionar código a dois métodos. O primeiro é o IntegrateAccel, que, por um determinado intervalo de tempo dt determina a quantidade correta de aceleração a ser aplicada a um determinado conjunto de forças aplicadas cada quadro. Adicione o seguinte código ao método atualmente vazio:

```
1 void PhysicsSystem :: IntegrateAccel ( float dt ) {
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 PhysicsObject * object = (* i ) -> GetPhysicsObject ();
8 if (objeto == nullptr) {
continuar ; // Nenhum objeto de física para este GameObject!
9 }
10 float inverseMass=objeto->GetInverseMass();
11 12
13 Vector3 linearVel=objeto->GetLinearVelocity();
14 Vector3 força=objeto -> GetForce();
15 Vector3 aceleração = força * massa inversa ;
16
17 if (applyGravity && inverseMass > 0) {
18 aceleração += gravidade; // não mova coisas infinitamente pesadas
19 }
20
21 linearVel += aceleração *dt; //integra aceleração!
22 objeto -> SetLinearVelocity (linearVel);
23 }
24}
```

Método PhysicsSystem::IntegrateAccel

Este método opera em cada GameObject no mundo do jogo por vez e, portanto, usa alguns iteradores e em seguida, itera através de cada GameObject e, se tiver um PhysicsObject, calcula a quantidade de aceleração como simplesmente f ·mÿ1 (linha 15), e a partir dela, integra uma nova velocidade usando o timestep (linha 21). Na linha 17, estamos também aplicarei alguma gravidade; esta força não é afetada pela massa de um objeto, a menos que assumamos isso é infinitamente massivo e, portanto, nunca deve se mover.

Integrar a velocidade resultante na posição de um objeto é praticamente o mesmo processo que integrar a aceleração. No método IntegrateVelocity, precisamos adicionar o seguinte código:

```
1  void PhysicsSystem :: IntegrateVelocity ( float dt ) {
2          std :: vector < GameObject * > :: const_iterator primeiro ;
3          std :: vector < GameObject * > :: const_iterator last ;
4          mundo dos jogos . GetObjectIterators (primeiro float , durar );
5          damperFactor = 1,0 f - 0,95 f;
6          float frameDamping = powf (amortecimentoFactor, dt);
7
8          for ( auto i = primeiro; i! = último; ++ i) {
9                  PhysicsObject * object = (* i ) -> GetPhysicsObject ();
10                 if (objeto == nullptr) {
11                         continuar ;
12                 }
13                 Transformar & transformar = (* i ) -> GetTransform ();
14                 //Coisas de posição
15                 Posição do vetor3 = transformação. GetLocalPosition();
16                 Vector3 linearVel=objeto->GetLinearVelocity();
17                 posição += linearVel * dt ;
18                 transformar. SetLocalPosition(posição);
19                 // Amortecimento Linear
20                 linearVel = linearVel * frameDamping ;
21                 objeto -> SetLinearVelocity (linearVel);
22         }
23 }
```

Método PhysicsSystem::IntegrateVelocity

É quase exatamente o mesmo processo, exceto que desta vez estamos obtendo a velocidade e integrando-a por dt (linha 17). Também simularemos uma pequena quantidade de arrasto ou resistência do ar por quadro, reduzindo ligeiramente o velocidade linear por um valor baseado na taxa de quadros, de modo que devemos obter uma quantidade consistente de velocidade redução, não importa a taxa de quadros.

Isso é tudo que temos a acrescentar à base de código, pois os métodos IntegrateAccel e IntegrateVelocity já estão sendo chamado pelo método PhysicsSystem::Update, só que antes os métodos estavam vazios! Juntamente com estes, ele também chama um método ClearForces, que passa por cada PhysicsObject, e chama ClearForces em pronto para o código de jogo do próximo quadro.

TutorialMudanças de classe do jogo

Para finalizar nossa adição de movimento linear em nosso mecanismo de física, vamos preencher outro método, este vez na classe TutorialGame. O método MoveSelectedObject é chamado a cada quadro no UpdateGame método, e usaremos a funcionalidade raycasting que preenchemos no tutorial anterior para detectar mais uma vez qual objeto está sendo clicado - se for o mesmo objeto que foi 'selecionado' anteriormente, colocaremos o PhysicsObject em a direção do raio, dimensionada por um número selecionável de Newtons, com base em quanto a roda de rolagem foi pressionado (adicionalmente enviamos esta variável para a tela na linha 25 para que seja mais fácil saber quanto um objeto será empurrado).

```
24 void TutorialGame :: MoveSelectedObject () {
       renderizador -> DrawString ("Click Force:" + std :: to_string (forceMagnitude) 25                                                    ,
26             Vetor2 (10          , 20)); // Desenha o texto de depuração em 10,20
27     forceMagnitude += Janela :: GetMouse () -> GetWheelMovement () * 100,0 f ;
28
29     if (! objeto de seleção) {
30         return ;// não selecionamos nada!
31     }
32     // Empurre o objeto selecionado!
33     if (Window :: GetMouse () -> ButtonPressed ( NCL :: MouseButtons :: MOUSE_RIGHT )) {
34         Raio raio = CollisionDetection :: BuildRayFromMouse (
35                                             * mundo -> GetMainCamera ());
36
37         RayCollision mais próximaColisão ;
38         if (mundo -> Raycast ( raio ,          colisão mais próxima          ,    verdadeiro )) {
39             if (mais próximoCollision. node == objeto de seleção) {
40                 seleçãoObject - > GetPhysicsObject () - >
41                     AddForce(ray.GetDirection()*forceMagnitude);
42             }
43         }
44     }
45}
```

Método PhysicsSystem::IntegrateVelocity

Conclusão

Depois de fazer essas alterações, deveremos ter algumas novas funcionalidades no código do nosso jogo. Uma vez que um objeto tenha sido selecionado, podemos voltar ao modo de movimento da câmera e clicar no objeto selecionado, onde deveria (espero!) ser empurrado na direção do mouse. Ainda mais, deveríamos poder pressionar o 'G' tecla para alternar a gravidade, sobre a qual todos os objetos que aparecem flutuando no ar cairão no chão, enquanto o chão permanece imóvel (por ser infinitamente massivo em nossa simulação física). Até agora, porém, não há nada para detectar quando os objetos realmente atingiram o chão, para que pareçam simplesmente se mover diretamente pelo chão!

Pressionar Q ainda alternará entre o modo de seleção de objeto e o modo de movimento da câmera e, além de poder clicar em objetos com o botão esquerdo do mouse, o botão direito do mouse aplicará uma força nele, o cuja magnitude pode ser ajustada usando a roda do mouse.

Neste tutorial, vimos o básico de como implementar um corpo rígido básico em nosso mundo de jogo, que pode se mover usando a integração de aceleração e velocidade em sua posição mundial. Nós vimos quão importante é o tempo de quadro para a integração e estabilidade adequadas de nossa simulação física, e como a massa e o amortecimento interage com o movimento dos nossos objetos ao longo do tempo. Enquanto brincava com o projeto, você provavelmente notou que as coisas não parecem muito certas - não há colisões para uma coisa, mas também não há rotação de qualquer tipo. À medida que a série de tutoriais avança, abordaremos esses problemas para produzir uma versão mais completa simulação de física começando com a adição de movimento angular ao nosso método de integração, permitindo que objetos girar à medida que as forças são aplicadas.

Trabalho adicional

1) O valor da gravidade para o mecanismo de física pode ser definido através do método SetGravity - tente alterar este valor para veja como isso afeta os objetos do jogo.

2) O valor de amortecimento do mecanismo de física pode ser definido através do método SetGlobalDamping - alterando isso deve ajustar a rapidez ou lentidão com que os objetos do jogo perdem velocidade ao longo do tempo.

3) Tente adicionar chaves que modificarão a posição do objeto selecionado usando forças. O MoveSelectedObject método é um bom lugar para adicionar essa funcionalidade extra.