Física - Raycasting

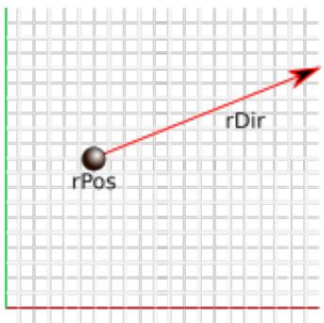
Introdução

Uma operação comum ao programar um videogame é determinar se um objeto pode ser visto ou tocado a partir de um ponto específico - clicar em um botão em uma tela de menu ou clicar em um tanque em um jogo RTS são exemplos disso. Às vezes, podemos desejar determinar se uma unidade de IA pode “ver” outra unidade de jogador a partir da sua posição atual. Embora essas verificações de visibilidade de IA e cliques do mouse no menu possam não parecer muito semelhantes superficialmente, ambos geralmente são realizados usando o mesmo processo - raycasting. Raycasting nos permite disparar linhas infinitamente finas de um ponto específico do mundo (seja a torre de um tanque ou o ponteiro do mouse do jogador) e ver com quais objetos ele colide ao longo do caminho. Dependendo do que colidir, podemos então chamar qualquer código personalizado necessário para adicionar interatividade às cenas do jogo.

Neste tutorial veremos como definir matematicamente uma semirreta, como construir semirretas que vão na direção que desejamos e observar os testes de interseção que nos permitem determinar se uma semirreta passa por uma forma simples ou não.

Raios

Um raio padrão é formado a partir de uma posição no espaço e de uma direção para viajar através do espaço. Podemos pensar nisso como um apontador laser - nós o posicionamos e, a partir dele, emitimos um laser que viaja em linha reta. para eventualmente acertar alguma coisa. Em termos de código, um raio é apenas dois objetos Vector3, um dos quais será normalizado para representar a direção do deslocamento. Podemos imaginar que a nossa direcção é uma linha infinita que se dirige para fora da origem do raio no espaço - o objectivo da emissão de raios é então ver quais os objectos no mundo que são intersectados por esta linha infinita.



Raios de uma matriz de transformação

É claro que podemos definir um raio em qualquer lugar do 'mundo' da nossa simulação e ver onde ele atinge, mas geralmente queremos iniciar o nosso raio a partir de algum ponto conhecido no mundo, como um objeto existente.
Portanto, é bastante útil poder definir um raio usando a matriz do modelo de um objeto. Vamos nos lembrar do que contém uma matriz modelo:

1 0 0 pixels
ÿ ÿ
0 1 0 ano
0 0 1 pz
ÿ 0 0 0 1 ÿ

Abaixo à direita temos a posição do objeto no espaço mundial, enquanto a matriz 3x3 superior contém a rotação do objeto no espaço mundial. A partir desta região superior 3x3, podemos determinar qual direção

o objeto está voltado para dentro. Se assumirmos que -z é a direção 'para frente' (como nas coordenadas OpenGL), então podemos formar o vetor 'para frente' [0,0,-1] negando a terceira linha ou coluna da região 3x3 superior - mas qual está correta? Podemos determinar isso examinando outro exemplo de matriz de modelo, uma que gira o objeto de modo que ele fique olhando 45ÿ para a esquerda:

$$\begin{bmatrix} xx & yx & yz & px \\ xy & yy & zy & py \\ xz & yz & zz & pz \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0{,}7071 & 0 & 0{,}7071 & \text{pixels} \\ 0 & 1 & 0 & py \\ -0{,}7071 & 0 & 0{,}7071 & pz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A partir disso, podemos ver que, quando negados, os valores da matriz [xz, yz, zz] nos fornecem o vetor de direção correto de [0,7071, 0, ÿ0,7071] para representar esta nova direção. A partir disso, podemos inferir que, a partir da matriz do modelo de qualquer objeto, [xx, yx, zx] apontará ao longo do 'eixo direito' desse objeto no espaço mundial, [xy, yy, zy] ficará voltado para cima e [xz, yz, zz] ficará voltado para frente.

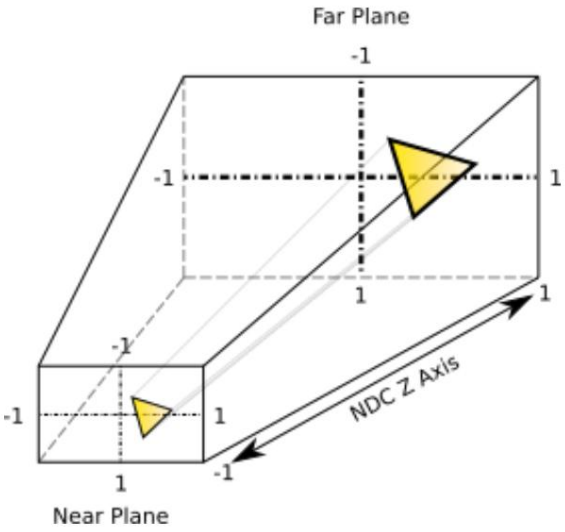
Raios do ponteiro do mouse

Às vezes, podemos querer definir um raio do ponto de vista da câmera. Uma maneira de fazer isso é usar a matriz de visualização - no entanto, isso não é tão simples quanto com a matriz do modelo. Lembre-se de que a matriz de visualização pode ser considerada o 'inverso' de uma matriz de modelo; portanto, para obter a posição e a orientação de uma matriz de visualização, ela deve primeiro ser invertida para transformá-la em uma matriz de modelo. Se tivermos certeza de que a matriz de visualização tem uma escala uniforme, então a transposição da matriz também funcionará, então o eixo 'direto' de uma matriz de visualização pode ser extraído usando [xx, xy, xz] (lembre-se, o a transposição inverte linhas e colunas, portanto, se já sabemos quais números extrair, não precisamos fazer a transposição 'completa').

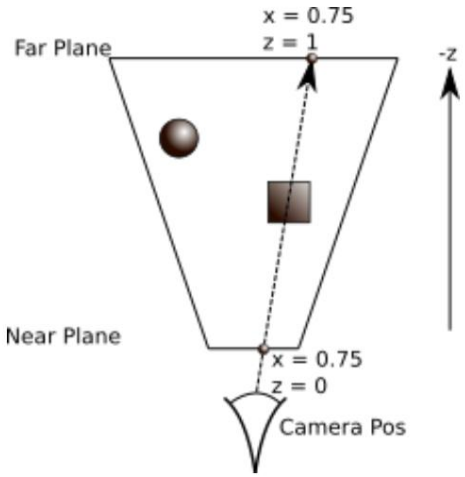
Normalmente não queremos apenas disparar um raio diretamente para a frente, mas clicar em algo na tela - seja uma caixa de menu em um RPG ou uma fábrica de tanques em um jogo RTS, de alguma forma, muitas vezes precisamos ser capazes para detectar o que está sob o ponteiro do mouse. Se estamos lidando com uma visão 3D, isso significa que precisamos lidar também com a matriz de projeção - lembre-se de que uma matriz de perspectiva tem um campo de visão que define o quão longe podemos ver 'para o lado', e assim quão 'lateralmente' do centro da tela um raio vindo da câmera pode apontar.

Supondo que temos a posição do mouse na tela, podemos descobrir onde está essa posição no espaço mundial. Se você se lembra, no tutorial de renderização adiada, fizemos exatamente a mesma coisa - dada a posição de um fragmento de tela, podemos chegar a uma posição no espaço mundial transformando a posição do espaço na tela pelo inverso da matriz de projeção de visualização e, em seguida, dividindo pelo 'inverso' w que ganhamos na multiplicação da matriz. Dito de outra forma, estamos desprojetando a posição de volta às coordenadas mundiais. Isso nos dá uma posição, mas como conseguir uma “direção”?

Para resolver isso, vale a pena dar uma olhada no tronco da vista formado pela matriz de projeção da vista:



Pense novamente no tutorial de renderização adiada - não foram apenas as coordenadas x e y da tela que usada para realizar nossa desprojeção, também utilizamos a amostra de profundidade, dando-nos uma coordenada z, também. Para formar um raio a partir de uma posição na tela, podemos realizar este processo de desprojeção em duas posições, cada um com as mesmas coordenadas xey, mas diferentes coordenadas do eixo z; uma subtração simples e a normalização destas duas posições deverá dar-nos um raio. Mas quais duas coordenadas do eixo z usar? Como com o qual já deveríamos estar familiarizados, temos um 'plano próximo' e um 'plano distante' delimitando tudo o que é visível em uma determinada direção. No OpenGL, e com as matrizes que estamos usando, isso mapeia quase plano para uma coordenada NDC de -1 no eixo z, enquanto o plano distante mapeia para o NDC de +1. Portanto, se desprojetamos as coordenadas NDC [x,y,-1] e [x,y,1] em coordenadas mundiais, podemos formar uma direção vetor no espaço mundial que passa direto por um ponto na tela:



Neste exemplo, clicamos na tela em aproximadamente 75% do caminho da tela em o eixo x; se então desprojetarmos esta coordenada em uma profundidade de -1 e 1 usando o inverso da vista matriz de projeção, acabamos com as coordenadas do espaço mundial, uma vez divididas pelo 'inverso' w eles ganham durante a transformação. A partir destas coordenadas, um raio pode ser determinado, e neste caso particular, podemos então ver que o raio viaja através do cubo, mas erra a esfera.

Calculando uma matriz de visão inversa

O processo de desprojeção requer o inverso da matriz de projeção da vista - mesmo para uma matriz 4x4, o processo de inversão é um processo bastante caro. Tanto para as matrizes de visualização quanto para as matrizes de projeção, sabemos quais valores os formaram (ou pelo menos podem armazená-los em algum lugar de nossas classes), o que nos permite em vez disso, derive uma matriz que fará o mesmo que uma matriz invertida, sem exigir que façamos o mesmo processo de inversão de uso geral em nossas matrizes.

Na classe Camera que usamos para manipular a matriz de visualização desde o módulo anterior, armazenamos separadamente a inclinação, a guinada e a posição, para facilitar a alteração desses valores conforme as teclas são pressionadas e o mouse é movido. Em seguida, geramos a matriz de visualização quando necessário usando o método BuildViewMatrix, que faz o seguinte:

```
1 Câmera Matrix4 :: BuildViewMatrix() const {
2         return Matrix4 :: Rotação ( - pitch ,                    Vetor3 (1       ,   0 ,   0)) *
3                         Matriz4 :: Rotação ( - guinada,        Vetor3 (0       ,   1 ,   0)) *
4                         Matriz4 :: Tradução ( - posição );
5}
```

Se invertermos a ordem dessas multiplicações de matrizes e usarmos os membros pitch, yaw e position variáveis (observe que o método BuildViewMatrix já nega as variáveis membro, pois elas são já definido no espaço mundial e deve ser 'invertido' para obter uma matriz de visão), então podemos obter uma matriz do modelo (e a variável membro posição também nos dá um ponto de origem do raio!):

```
1 Matrix4 cameraModel = Matrix4 :: Tradução (posição)                                   *
                        Matrix4 :: Rotação ( guinada ,   Vetor3 (0 2    ,  1 ,  0)) ,  *
3                       Matrix4 :: Rotação ( pitch ,     Vetor3 (1      ,  0 0));
```

Calculando uma matriz de projeção inversa

Calcular o inverso de uma matriz de projeção é um pouco mais complicado, mas tudo que precisamos é o mesmo variáveis com as quais definimos uma matriz de projeção - uma proporção de aspecto, um campo de visão, um plano próximo e um plano distante avião. Aqui está um lembrete da aparência da matriz de projeção em perspectiva:

$$\begin{pmatrix} \frac{f}{aspecto} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zPerto+zFar}{zPerto-zFar} & \frac{2 \cdot zPerto \cdot zFar}{zPerto-zFar} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Onde f é a cotangente do campo de visão. Se definirmos mais genericamente a matriz de projeção como:

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & d \\ 0 & 0 & e & 0 \end{pmatrix}$$

Podemos ver que as partes aeb não interagem com nenhum outro eixo, pois estão na diagonal, e não há mais nada em sua linha/coluna. Portanto, para 'desfazer' qualquer efeito que eles tenham sobre um vetor durante a transformação, podemos usar seu recíproco. As partes d e e são um pouco mais complicadas - a parte d nos dá uma interação entre os eixos z e w, enquanto a parte e mapeia o eixo z da entrada vetor no vetor de resultado. Para “desfazê-los”, não só temos de considerar a sua recíproca, mas também transponha-os para mapear os valores de volta ao seu eixo original, de modo que a parte e vai da linha 4, coluna 3, para a linha 3, coluna 4, enquanto a parte d vai para o outro lado. A parte c é a mais complicada de desfazer – em uma projeção matriz é usada para dimensionar o valor z gravado no buffer de profundidade que define a distância de um fragmento é, mas como este é o valor que estamos fornecendo agora, precisamos de alguma forma de dimensionar todos os outros coordenadas por ele para 'inverter' seu uso. Para fazer isso, a parte c precisa ir para o eixo w do vetor resultante, para que possamos dividir ainda mais por 'w inverso', estendendo todas as coordenadas de Espaço NDC de volta ao espaço mundial. Também precisamos desfazer a tradução original que seria adicionada ao eixo z, então isso também vai para o nosso resultado w - no total, obtemos a seguinte matriz para inverter o projeção:

$$\begin{pmatrix} \frac{1}{a} & 0 & 0 & 1 & 0 \\ 0 & \frac{}{} & 0b & 0 & 0 \\ 0 & 0 & 0 & 1 & \frac{1}{e} \\ 0 & 0 & \frac{}{d} & - & \frac{c}{de} \end{pmatrix}$$

Veremos como isso se relaciona com os valores específicos que calculamos para uma projeção em perspectiva matriz no código do tutorial. Uma derivação mais completa do inverso de uma matriz de projeção pode ser encontrado no site de Brian Hook em:

http://bookofhook.com/mousepick.pdf.

Volumes de colisão

Formar um raio é uma coisa, mas também precisamos de ser capazes de descobrir o que esse raio está a intersectar. Poderíamos, se quiséssemos, testar cada triângulo da malha de um objeto contra um raio, já que existe um cálculo comum para determinar se um raio intercepta um triângulo, mas isso geralmente é um exagero para muitos propósitos - não nos importamos se clicamos em um torre ou trilhos do tanque, queremos apenas descobrir se clicamos em uma unidade específica em um jogo RTS. Geralmente, tentamos descobrir se fizemos a intersecção com uma forma aproximada que encapsula o objeto contra o qual queremos fazer o raio, como uma esfera ou uma forma de caixa. Essas aproximações aproximadas serão usadas ao longo da série de tutoriais, para nos permitir calcular as interações físicas entre objetos de maneira eficiente, de modo que a realização de raycasts contra essas formas sirva como uma boa introdução a elas.

Aviões

Já usamos uma forma de colisão, embora não seja realmente um volume como tal. Ao longo do módulo gráfico, vimos como o tronco da vista pode ser representado como um conjunto de 6 planos, cada um dos quais divide o espaço em dois subespaços, algo como uma parede infinitamente grande e infinitamente fina. Assim, podemos formar volumes a partir deles (como nosso tronco de visão), e os planos geralmente fazem parte dos testes de colisão que veremos nestes tutoriais.

Podemos representar um plano usando 4 valores, sendo a equação clássica do plano a seguinte:

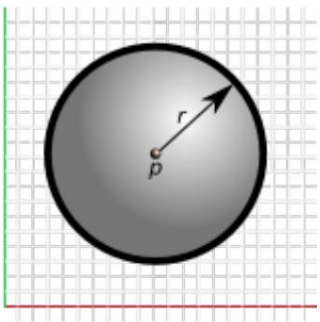
$$Machado + Bx + Cx + D = 0$$

Qualquer ponto no espaço nas posições x,y,z é considerado 'no' plano se o produto escalar entre os valores do plano ABCD e [x,y,z,1] for igual a 0. Além disso, podemos considerar um ponto dentro (ou na frente) do plano se o resultado do produto escalar for maior que 0, e fora (ou atrás) do plano se o produto escalar for menor que 0.

Os aviões são frequentemente usados em jogos como limites absolutos no mundo do jogo - às vezes, erros no nível do jogo significam que um jogador pode escapar do mundo do jogo e cair no chão, por isso os desenvolvedores de jogos costumam colocar 'aviões letais' abaixo do chão, que são conectado a um código que matará automaticamente o jogador se sua posição mundial estiver do lado 'errado'.

Esferas

Também podemos representar uma esfera usando quatro valores - uma posição vetorial p representando o meio da esfera e um raio r. Qualquer ponto no espaço que esteja a uma distância menor que r de x,y,z está, portanto, 'dentro' da esfera e, se for igual, estará na superfície da esfera.

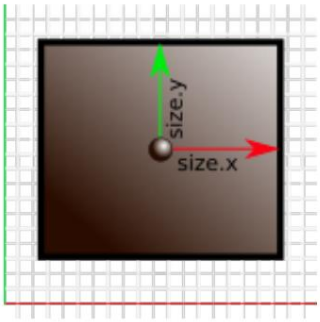


Caixas

As caixas são um pouco mais complexas que as esferas. Existem dois tipos de caixas usadas para detecção de colisão (às vezes chamadas de caixas delimitadoras) - caixas delimitadoras alinhadas ao eixo e caixas delimitadoras orientadas a objetos.

AABBs

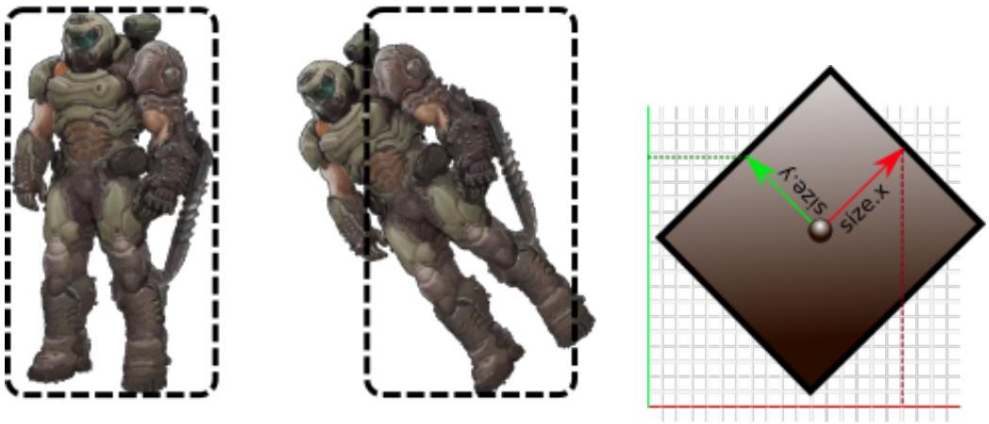
Uma caixa delimitadora alinhada ao eixo possui uma posição no espaço 3D e também um tamanho, definindo o comprimento, largura e altura da caixa. Geralmente estes são armazenados como 'meios tamanhos', ou seja, a distância do meio da caixa até a borda:



Não importa a orientação do objeto representado com um AABB, o AABB nunca gira - portanto, seu eixo x sempre aponta ao longo do espaço mundial x, e o mesmo para y e z. Isso torna os AABBs um pouco mais simples de realizar cálculos, já que a caixa e qualquer outra construção com a qual estamos testando (seja um raio como neste caso, ou outro volume de colisão para detectar colisões entre pares de objetos) sempre estarão no mesmo 'espaço'.

OBBs

O problema de representar o volume delimitador de um objeto com um AABB é que, à medida que a orientação de uma forma muda ao longo do tempo (devido ao movimento do jogador ou às forças físicas aplicadas a ela), o tamanho da caixa não refletirá as novas extensões do objeto em os eixos mundiais:



À medida que a forma acima gira, o AABB de seu volume de colisão fica cada vez pior, portanto, qualquer detecção de colisão ou projeção de raios realizada nele será imprecisa. Podemos estender a ideia de uma caixa com origem e meio tamanho para também ter uma orientação, armazenando o meio tamanho como um vetor e depois transformando-o pela região superior 3x3 da matriz do modelo de um objeto - criando uma caixa delimitadora orientada. Então, por que nos preocupamos com AABBs se os OBBs correspondem melhor ao formato dos objetos? Como veremos ao longo desta série de tutoriais, algumas das propriedades dos AABBs tornam muito mais fácil detectar colisões entre eles, portanto, a menos que seja necessária precisão extra, os AABBs são frequentemente preferidos aos OBBs.

Intersecções de Raios

Embora as formas de colisão acima não sejam as únicas comuns, elas são um bom ponto de partida para entender como funcionam as colisões entre formas e raios. Vamos agora dar uma olhada em alguns dos algoritmos para determinar se um raio cruzou alguma dessas formas e quais informações podemos obter delas.

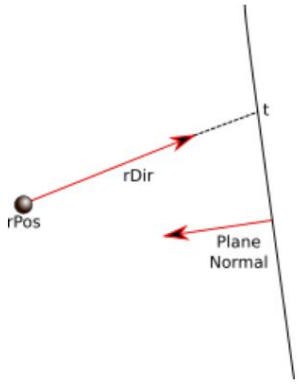
Intersecções de raios/planos

O teste de interseção mais fácil de calcular é entre um raio e um plano. A menos que a normal do plano e a direção do raio estejam voltadas na mesma direção, sempre haverá uma interseção entre um raio e um determinado plano, que pode ser calculada da seguinte forma:

$$t = \frac{ÿ(Raypos \cdot P\ laneabc + planed)}{Raydir \cdot P\ laneabc}$$

$$p = Raypos + t(Raydir)$$

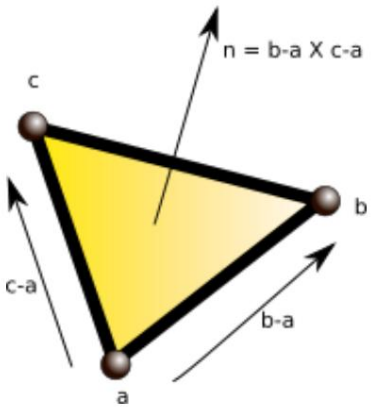
Isso nos permite obter o comprimento ao longo do raio até a interseção t e, a partir daí, o ponto de interseção real p. Ao codificar uma intersecção raio/plano, você pode querer calcular primeiro Raydir ·Planeabc , pois isso verificará a ortogonalidade entre o plano normal e a direção do raio, para que você possa evitar uma divisão por 0 nesses casos.



Raio / Triângulos

Há casos em que podemos querer realizar testes de interseção de raios contra um triângulo - alguns jogos têm decalques realistas de 'buracos de bala' aplicados aos modelos quando eles são disparados, cujo ponto pode ser determinado pela projeção de raios ao longo da trajetória da bala.

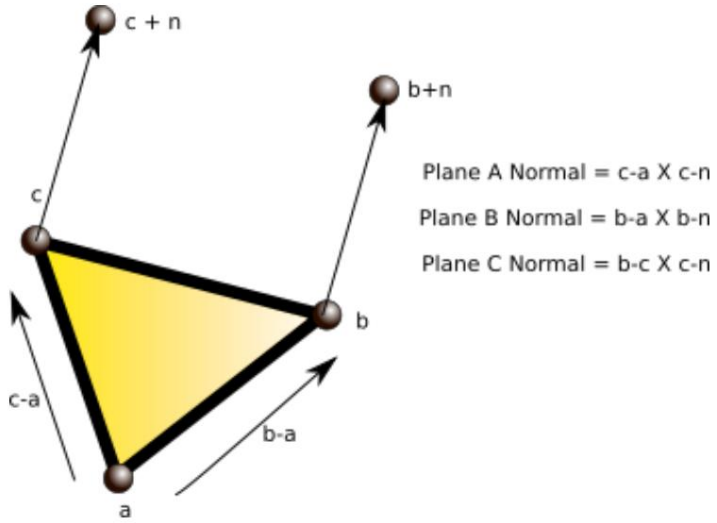
O primeiro passo para realizar a interseção do raio/triângulo é formar um plano a partir do triângulo - os triângulos são sempre superfícies planas, então podemos usar isso para tornar o cálculo mais simples. Para calcular o plano de um triângulo, precisamos da normal do triângulo, que como você deve se lembrar do módulo anterior, pode ser calculada a partir de dois vetores e um produto vetorial:



O triângulo normal nos dá 3 coeficientes planos Planeabc, enquanto realizamos o produto escalar em qualquer ponto do triângulo e o normal nos dá o plano. A partir daí, podemos realizar a interseção do raio/plano descrita anteriormente para encontrar o ponto no 'plano do triângulo' no qual o raio se cruza.

A partir daí, podemos formar outros 3 planos - em vez de seguir ao longo da superfície do triângulo, cada plano desliza ao longo de uma das arestas do triângulo. Somente se o ponto no plano do triângulo estiver dentro de cada um desses planos é que um ponto será considerado dentro do triângulo. Se o ponto estiver dentro do plano, então

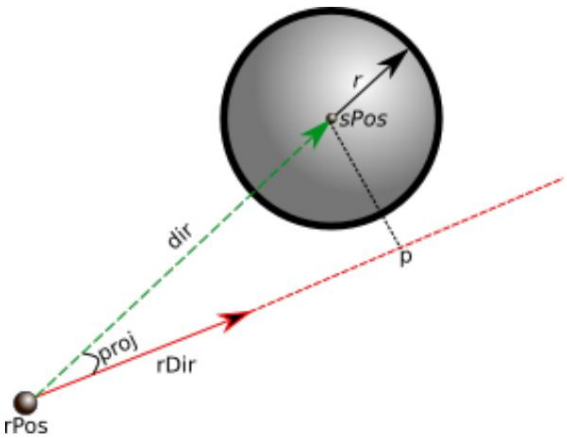
tem o ponto de intersecção e a distância da origem do raio, como antes. Para formar os planos extras, podemos determinar pontos extras neles movendo-nos ao longo do triângulo normal a partir de qualquer canto - isso nos dará 3 pontos para usar, assim como fizemos para obter o triângulo normal acima.



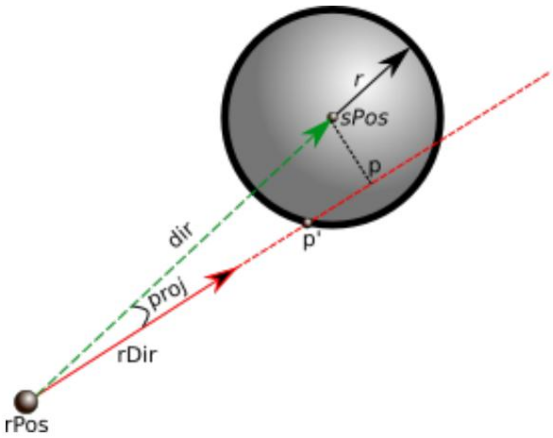
Intersecções de raio/esfera A intersecção

de raio/esfera é um pouco diferente dos triângulos, mas não muito difícil. Neste caso, vamos calcular o ponto mais próximo ao longo do raio da esfera s projetando a origem da esfera no vetor de direção do raio - isso parece difícil, mas é apenas um produto escalar e alguma multiplicação vetorial:

dir = spos ÿ rpos
proj = rdir · dir p =
rpos + proj(rdir)



Assim que tivermos esse ponto, podemos determinar rapidamente se o raio intercepta a esfera calculando a distância da origem da esfera ao ponto p - se for menor que o raio da esfera, o raio intercepta a esfera. Neste caso, se nos movermos ao longo de rDir por unidades proj para chegar ao ponto p mais próximo, podemos ver que ele é maior que o raio da esfera distante e, portanto, nosso raio erra a esfera.
Isto pode dizer-nos se o raio se cruza ou não, mas muitas vezes também precisamos de saber a que distância do raio está o ponto de intersecção. Você pode pensar que a distância entre rPos e p nos daria a resposta correta, mas isso não é bem verdade:



Quando o raio intercepta a esfera, o cálculo acima para retornar p ainda nos dá o ponto mais próximo entre a origem da esfera e a direção do raio - então, neste caso, p acaba dentro da esfera.
Para calcular a distância real de intersecção dos raios d, precisamos de um cálculo adicional:

$$d = \|p \ddot{y} rpos\| - \sqrt{r^2 \ddot{y} (\|p \ddot{y} spos\|2)}$$

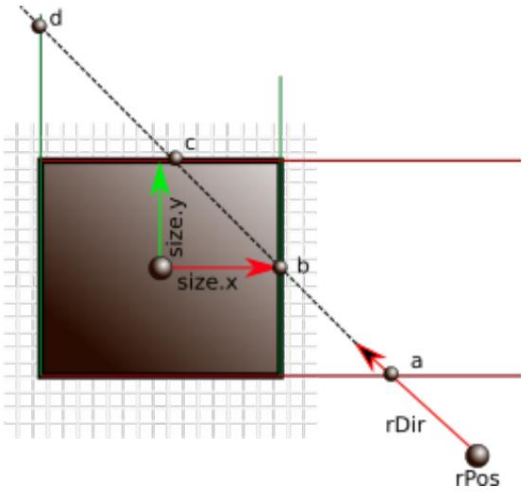
Ou seja, obtemos a distância entre o ponto de intersecção e o raio e subtraímos o raio ao quadrado - a distância entre o ponto p e a origem da esfera. Podemos então encontrar o verdadeiro ponto de intersecção p' viajando ao longo do vetor rDir por uma distância d.

Intersecções de raio/caixa

A detecção de colisões entre caixas e raios reduz novamente os testes de interseção de planos. Assim como em nossa visão do módulo anterior, podemos representar uma caixa usando 6 planos para formar um volume encapsulado.

Interseção da AABB

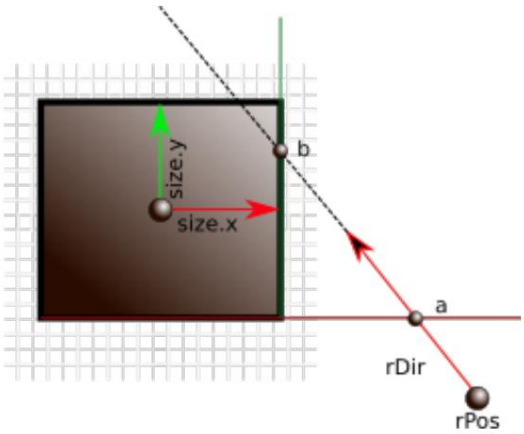
Para realizar a intersecção dos raios contra uma caixa delimitadora alinhada ao eixo, poderíamos calcular o ponto de intersecção contra todos os 6 planos que compõem a caixa e descobrir se algum deles está na superfície da caixa. Da nossa definição anterior de uma caixa de ter uma posição e meio de tamanho, se subtrairmos a posição do ponto no espaço que desejamos testar do ponto da caixa, o ponto de teste deve estar dentro da caixa se o vetor resultante tiver um valor absoluto posição inferior à metade do tamanho da caixa em cada eixo.



Neste caso, os pontos a e d (representando o ponto em que o raio cruza os planos 'esquerdo' e 'inferior' da AABB) estão fora da superfície da AABB, mas os pontos b e c (representando as interseções ao longo do plano 'direito' e do plano 'superior') tocam a superfície (o ponto b está exatamente size.x unidades de distância da origem do cubo no eixo x, e o ponto c é igual a size.y unidades no eixo y) , e assim sabemos que há uma interseção - se realmente precisarmos saber o ponto exato da interseção, precisaremos

para determinar se o ponto b ou o ponto c está mais próximo de rPos, calculando o comprimento dos vetores b-rPos e c-rPos e escolhendo o comprimento mais curto - neste caso, o ponto b deve ser o ponto de intersecção.

Embora pudéssemos testar todos os 6 lados, na verdade é desnecessário. Se um raio vem da direita de uma caixa, então o raio sempre atingirá o plano do lado direito antes do plano do lado esquerdo:

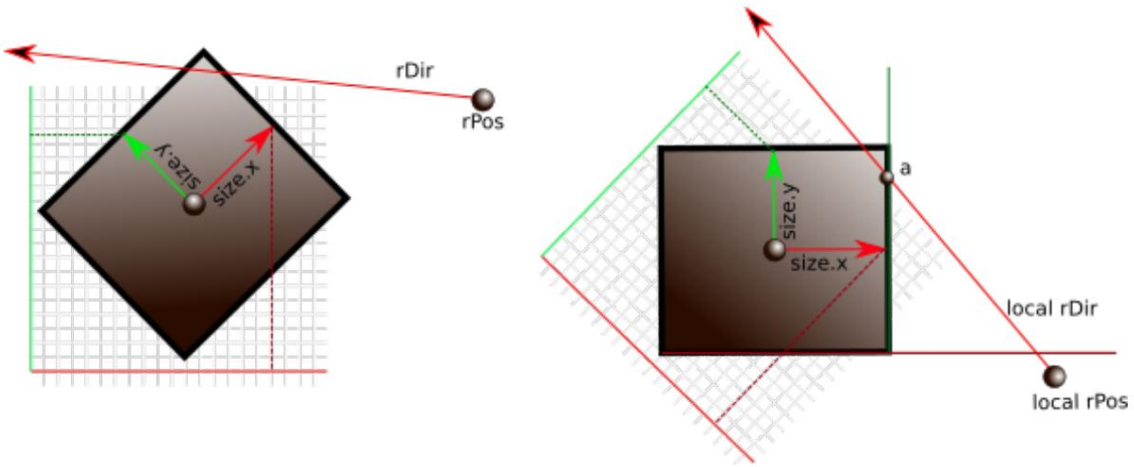


Podemos, portanto, usar isso apenas para testar contra 3 planos em vez de 6, usando os valores do vetor de direção normalizado do raio - para cada eixo, se a direção do raio for positiva, verificamos o plano 'negativo' (subtraímos o tamanho do origem), e se for negativo testamos o plano 'positivo' (adicionamos o tamanho à origem). Com o número de testes reduzido para três, podemos agora ver que o ponto de intersecção mais distante será a “melhor” escolha, já que estamos ignorando quaisquer planos que seriam atingidos quando o raio saísse da caixa. No exemplo acima, mesmo que o ponto 'a' esteja mais próximo, podemos pular a verificação, pois o ponto b está mais ao longo do raio. Contanto que a intersecção mais distante esteja na superfície da AABB (não está mais distante no eixo relevante do que o tamanho da caixa), então temos nosso ponto de colisão.

Interseção OBB

Caixas delimitadoras orientadas são mais complicadas. Embora possamos obter os deslocamentos do plano de um AABB por meio de simples adição e subtração de valores únicos, porque sabemos que os planos estarão alinhados ao eixo - o 'lado direito' de uma caixa sempre terá um deslocamento de [alguma distância,0,0] da sua posição. Porém, se estivermos lidando com uma caixa girada, isso não é mais verdade, e temos que determinar os vetores de direção corretos que apontam para frente, para cima e para a direita do objeto (ou em outros termos, o mapeamento do espaço mundial dos eixos do espaço local do objeto). Podemos extrair os eixos x, y e z da matriz de transformação do objeto conforme descrito anteriormente, mas agora teremos que testar 6 planos, pois agora é muito mais difícil determinar os 'melhores' 3 planos. Também é mais difícil testar se o 'melhor' ponto está dentro da caixa após a projeção pelas mesmas razões - não podemos mais verificar apenas um número por eixo!

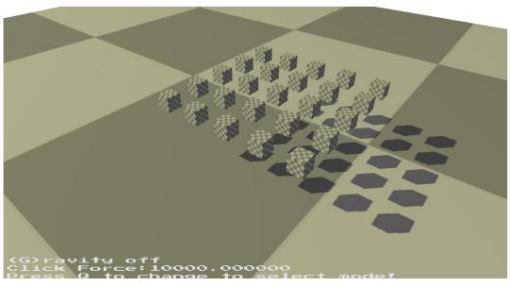
Neste caso, é melhor pensar nas coisas de uma maneira diferente - podemos transformar temporariamente a posição e a direção do raio usando o inverso da matriz de transformação do objeto, transformando o teste OBB no espaço mundial em um teste AABB na caixa. espaço local:



Isso nos dá um ponto de intersecção a no espaço local do objeto, então ele precisará ser transformado pela matriz do modelo do objeto novamente para recuperá-lo no 'espaço mundial' - isso somente se precisarmos da interseção ponto, se tudo o que precisamos saber é se o raio está se cruzando ou não, podemos simplesmente deixar o ponto em espaço locais. Isto lhe dá alguma indicação sobre a dificuldade em usar caixas delimitadoras orientadas, que só fica mais complexo à medida que começamos a observar as colisões de objetos mais tarde. Esta é também uma introdução uma operação comum que fazemos em cálculos de física - trazer alguma posição a para o 'espaço local' de algum objeto b. Se tudo o que nos importa é uma posição, podemos simplesmente subtrair a da posição de b para obtenha a posição relativa de a. Se, no entanto, precisarmos conhecer também a rotação, devemos usar o inverso de matriz do modelo do objeto, pois uma transformação por esta matriz nos trará de volta ao espaço local do objeto.

Código do Tutorial

Nosso programa de exemplo para demonstrar raycasting será bastante simples; apenas o suficiente para conseguir um uma ideia de como o raycasting pode ser usado e integrado a uma base de código de jogo. A classe TutorialGame nos fornece uma cena padrão que consiste em cubos e esferas - perfeita para testar alguns raios código com! Veja como fica o 'jogo' quando iniciamos o programa:



Já está tudo configurado para construir uma lista de objetos para renderizar na tela cada quadro. O que o código realmente não tem, porém, um mecanismo de física funcional ou qualquer forma de executar tarefas como fundição de raios. Ele tem uma classe bastante vazia chamada PhysicsEngine e um namespace chamado CollisionDetection, porém, e os tutoriais de física deste módulo irão preenchê-los, para dar nos dá uma ideia de como construir uma simulação física funcional em nossos jogos.

Volumes de colisão

Ser capaz de representar um objeto de colisão em nosso código é claramente muito importante - as técnicas de raycasting acima usam formas de colisão de vários tipos e, à medida que a série de tutoriais avança, veremos sobre como detectar colisões entre eles também. Na base de código fornecida, cada um dos volumes que iremos o teste derivará da classe CollisionVolume, conforme mostrado aqui:

```
# pragma uma vez
namespace NCL { // monitore seus namespaces !
	enum class VolumeType {
		AABB = 1 , OBB = 2 , Esfera = 4 , Malha = 8 ,
		Composto = 16 , Inválido = 256
	};
	class CollisionVolume {
	público :
		Volume de Colisão() {
			tipo = VolumeType :: Inválido;
		}
		~ CollisionVolume() {}
		Tipo de volume;
	};
} // fim do namespace !
```

Classe CollisionVolume

Não há muito a fazer além de armazenar um enum, que podemos usar para determinar que tipo uma classe derivada é, sem fazer nenhum método virtual ou conversão dinâmica. Para ver como isso é usado para representar um dos tipos reais de volume de colisão que abordamos anteriormente, veja como o fornecido A classe SphereVolume parece:

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

namespace NCL {
    class SphereVolume : CollisionVolume {
público :
        SphereVolume ( float esferaRadius = 1,0 f ) {
            digite = VolumeType :: Esfera ;
            raio = esferaRadius;
        }
        ~ SphereVolume () {}

        float GetRadius() const {
            raio de retorno ;
        }
    protegido :
        raio de flutuação ;
    };
}
```

Classe EsferaVolume

Possui um raio, mas nenhuma posição armazenada. Isso ocorre porque, em vez disso, usaremos a posição mundial de um matriz de transformação, armazenada dentro de uma classe 'Transform', que fará parte de um GameObject - essas classes foram abordadas mais detalhadamente no tutorial de introdução, portanto não iremos repassá-las aqui. Os outros volumes de colisão na base de código seguem praticamente o mesmo padrão, então familiarize-se com ele.

Classe Raio

Para representar o conceito de raio em C++, temos a classe Ray apropriadamente chamada. Não há demais para isso:

```
classe Raio {
público :
        Raio (posição Vector3, Direção do vetor3);
        ~ Ray ( vazio );

        Vector3 GetPosition() const { retorna posição; }
        Vector3 GetDirection () const { direção de retorno ; }

    protegido :
        Posição do vetor3; // Posição no espaço mundial
        Direção do vetor3; // Direção normalizada do espaço mundial
    };
```

Cabeçalho da classe Ray

Já fomos apresentados à classe Vector3 no módulo anterior e só precisamos de dois deles. aqueles para representar nosso raio; um para posição e outro para sua direção - este vetor é sempre assumido como ser normalizado. Além de getters, isso é tudo que precisamos para um raio. As coisas ficam um pouco mais interessantes para representam uma colisão de raios, no entanto. Para isso, você também recebeu um modelo RayCollision classe, mostrada aqui:

```
template < typename T>
class RayCollision {
    public:
        T* node;                                    //Node that was hit
        Vector3 colidiuAt ; // ESPAÇO MUNDIAL posição da colisão !
        RayCollision (T * node Vector3 colidiuAt) {
            this -> node = node;
            this -> colidedAt = colidiuAt;
        }
        RayCollision() {
            node = nullptr;
        }
};
```

Cabeçalho da classe Ray

A modelagem serve apenas para abstrair a ideia de um raio atingindo algo da implementação exata do nosso código (como um raio é uma construção bastante útil, podemos torná-lo mais reutilizável mantendo separado de como exatamente deve ser usado). Iremos usá-lo para armazenar ponteiros para GameObjects, junto com uma posição no espaço mundial onde ocorreu a colisão.

Classe de detecção de colisão

Para realmente usar a classe Ray e realizar testes de interseção entre os raios e nossos volumes de colisão, vamos adicionar algumas funcionalidades à classe CollisionDetection, o que atualmente não faz muito, mas nos fornece algumas funções para calcular a projeção inversa e visualizar matrizes, e 'desprojetar' as posições da tela no espaço mundial. Ele também possui um conjunto de funções de intersecção de raios que levam em um raio, transformação e volume delimitador, e retornar verdadeiro ou falso, dependendo se houve uma colisão ou não. Para começar, porém, cada uma dessas funções retorna apenas falso, então eles não fazem nada!

Para fazer o raycasting funcionar, teremos que preencher essas funções com algumas funcionalidades adequadas. Começaremos com a função RayIntersection, cujo objetivo é obter o tipo de a passado volume e chame a função de interseção de raio apropriada para ver se o raio passado colide com ele. Aqui está o código para a função RayIntersection:

```
1  bool CollisionDetection :: RayIntersection ( const Ray & r                                    ,
       GameObject e objeto, RayCollision e colisão) {
2  const Transformar e transformar                    = objeto. GetConstTransform();
3      const CollisionVolume * volume                 = objeto. GetBoundingVolume();
4      se (! volume) {
5          retorna falso ;
6      }
7      switch (volume -> tipo) {
8          case VolumeType :: AABB :          return RayAABBIntersection (r ( const          ,
9              transformação   ,   AABBVolume &)* colisão de volume);          ,
10         case VolumeType :: OBB : return RayOBBIntersection (r transform ( const OBBVolume &)*  ,
11             colisão de volume );                                                ,
12         case VolumeType :: Sphere : return RaySphereIntersection (r ( const SphereVolume &)* colisão de  ,
13             transformar     ,   volume );                                          ,
14     }
15     retorna falso ;
16 }
```

Detecção de Colisão::RayIntersection

Não é muito, mas nos ajuda a ver como a classe GameObject contém um Transform e um Colli-sionVolume (linhas 3 + 4). Presume-se que todo GameObject nesta base de código tenha uma transformação, portanto, o acessador deste objeto retornará uma referência, embora o objeto possa não ser colidível, e, portanto, pode não ter um CollisionVolume, então o acessador retorna um ponteiro, que podemos

verifique e retorne falso se não houver, pois não podemos colidir com ele (linhas 6 a 8). Se o GameObject passado tem um volume, podemos alternar contra sua variável de tipo e chamar a função de interseção apropriada. Isso envolve lançar o CollisionVolume para o sub correto class - contanto que não modifiquemos a variável de tipo definida nos construtores da subclasse, isso sempre acontecerá pressione a instrução switch 'correta' para o tipo de colisão real.

Código de intersecção raio/esfera

Ótimo! Agora vamos ver como a teoria por trás da fundição de raios pode ser implementada em C++. Começaremos com o mais fácil, e observe as colisões Ray/Sphere, que devem ser implementadas na função RaySphereInter-section, detalhada aqui:

```
1  bool CollisionDetection :: RaySphereIntersection ( const Ray &r 2 const SphereVolume e volume ,
       transformação const e transformação mundial ,                                              ,
3      RayColisão e colisão) {
4      Vetor3 esferaPos = worldTransform . GetWorldPosition();
5      float esferaRadius = volume. ObterRaio();
6
7      // Obtém a direção entre a origem do raio e a origem da esfera
8      Vector3 dir = (esferaPos - r.GetPosition());
9
10     // Em seguida, projetamos a origem da esfera em nosso vetor de direção de raio
11     float esferaProj = Vector3 :: Ponto ( dir , R. ObterDireção());
12
13     if( esferaProj <0,0 f) {
14         retorna falso ; // o ponto está atrás do raio!
15     }
16
17     // Obtém o ponto mais próximo da linha do raio para a esfera
18     Vetor3 ponto = r . GetPosition() + ( r . GetDirection () * esferaProj );
19
20     float esferaDist = (ponto - esferaPos). Comprimento ();
21
22     if (esferaDist > esferaRadius) {
23         retorna falso ;
24     }
25
26     deslocamento flutuante =
27         sqrt ((esferaRadius * esferaRadius) - (esferaDist * esferaDist));
28
29     colisão. rayDistance = esferaProj - (deslocamento);
30     colisão. colidiu em = r. ObterPosição() +
31                 (r.GetDirection()*colisão.rayDistance);
32     retornar verdadeiro ;
33 }
```

Detecção de Colisão::RaySphereIntersection

É uma implementação bastante direta da teoria - calculamos o vetor de direção entre o raio e o centro da esfera (linha 8) e, em seguida, use o operador de produto escalar para projetar este vetor contra o vetor de direção do raio (linha 11) - vamos ver até que ponto ao longo do vetor de direção podemos viajar antes de nos 'juntarmos' ao final do outro vetor (linha 18). Se esse ponto projetado for maior maior que o raio da esfera, o raio não pode estar colidindo (linha 19), caso contrário, determinamos a colisão ponto movendo o ponto de colisão de volta ao longo do vetor de direção, de modo que ele toque a superfície do a esfera, em vez de estar dentro dela. A linha 14 cobre o caso em que a esfera está atrás do raio - o produto escalar entre a direção do raio e o vetor de direção entre o raio e o objeto será este caso acaba sendo negativo e não devemos considerar mais o objeto.

Código de interseção Ray / AABB

A seguir, daremos uma olhada nas colisões de raios com caixas. Primeiro vamos escrever a função que realizará interseções entre uma caixa e um raio. Adicione este código ao RayBoxIntersection função no arquivo de classe CollisionDetection:

```
1  bool RayBoxIntersection ( const Ray &r                    ,    const Vector3 e boxPos                    ,
2                            const Vector3 e tamanho da caixa , RayColisão e colisão ) {
3          Vetor3 boxMin = boxPos - boxSize ;
4          Vetor3 boxMax = boxPos + boxSize ;
5
6          Vetor3 raioPos = r . ObterPosição();
7          Vetor3 raioDir = r . ObterDireção();
8
           Vetor3 tVals (-1                  ,    -1, -1);
9  10
11         for (int i = 0; i < 3; ++ i ) { // obtém as 3 melhores interseções
12                 if (rayDir [i] > 0) {
13                         tVals [i] = (boxMin [i] - rayPos [i]) / rayDir [i];
14                 }
15                 senão if( rayDir [i] < 0) {
16                         tVals [i] = (boxMax [i] - rayPos [i]) / rayDir [i];
17                 }
18         }
19         float melhorT = tVals. GetMaxElement();
20         if( melhorT < 0,0 f ) {
21                 retorna falso ; // sem raios invertidos !
22         }
```

Detecção de Colisão::RayBoxIntersection

Para realizar o teste da caixa, usaremos o caso 'reduzido' que marca apenas as 3 caixas mais próximas planos, em vez de todos os 6. Para fazer isso, vamos verificar a direção do raio - se ele estiver indo para a esquerda, verificamos apenas o lado direito da caixa, se estiver subindo, verificamos apenas o lado inferior da caixa, e se for daqui para frente, verificamos apenas o verso da caixa. Como a caixa está alinhada ao eixo, só precisamos para verificar cada eixo individual da direção do raio, juntamente com a extensão mínima ou máxima dessa caixa ao longo do eixo - é por isso que nas linhas 13 e 16 verificamos em [i], o que nos dará o eixo x, y ou z de um vetor. Os comprimentos resultantes ao longo do vetor são então armazenados em outro Vetor3, tVals. Isso nos permite usar o método membro GetMaxElement do vetor, que como seu name Suggest nos dará o float com a maior magnitude. Inicializamos os vetores tVals com valores negativos - se o elemento máximo dentro dele ainda for negativo após a conclusão do loop for, então a intersecção está realmente atrás do raio e deve ser ignorada (através do retorno na linha 22). A partir do valor máximo, podemos então determinar o melhor ponto de intersecção ao longo do vetor de raio, e armazene-o na variável de interseção e descubra se ele está realmente na superfície da caixa ou não:

```
23         Intersecção Vector3 = rayPos + ( rayDir * bestT );
24         const float épsilon = 0,0001 f; // uma quantidade de margem de manobra em nossos cálculos
25         for (int i = 0; i < 3; ++ i ) {
26                 if (intersecção [i] + épsilon <boxMin[i] ||
27                         interseção [i] - épsilon > boxMax [i]) {
28                 retorna falso ; // a melhor interseção não toca na caixa!
29                 }
30         }
31         colisão. colidiu em                                = interseção;
32         colisão. distância do raio = melhorT;
33         retornar verdadeiro ;
34 }
```

Detecção de Colisão::RayBoxIntersection

O loop for na linha 25 apenas passa por cada eixo e determina se o ponto de interseção é muito longe para um lado ou outro da caixa, determine pelas extensões mínima e máxima da caixa ao longo desse eixo. Observe que estamos usando um pequeno limite de erro (chamado épsilon) para acomodar pequenas variações na precisão do ponto flutuante - não queremos um ponto que esteja a 0,0001 unidades de distância de um cubo para contar como 'não se cruza' se esta distância for apenas devido à forma como os pontos flutuantes operam. Se isso colide, podemos preencher nossos detalhes de colisão diretamente e retornar verdadeiro.

Assim que tivermos a função RayBoxIntersection implementada, podemos usá-la para AABB e OBB colisões de raios. As colisões AABB são calculadas com a função RayAABBIntersection, que é praticamente apenas uma 'passagem' para a função que acabamos de escrever, pois tudo o que ela precisa fazer é obter o tamanho e posição da caixa da AABB e use-os como parâmetros:

```
1 bool CollisionDetection :: RayAABBIntersection ( const Ray &r 2 3 4 5 6 7 }                    ,
                    const Transform & worldTransform const
                    ,
                    AABBVolume e volume                    , RayColisão e colisão ) {
      Vector3 boxPos = worldTransform . GetWorldPosition();
      Vector3 boxSize = volume. GetHalfDimensions();
      retornar RayBoxIntersection (colisão r boxPos);    ,   tamanho da caixa  ,
```

Detecção de Colisão::RayAABBInterseção

As coisas são um pouco mais complicadas para um OBB, pois precisamos transformar o raio para que fique no local espaço da caixa (linha 12), e se colidir, transformar o ponto de colisão de volta no espaço mundial (linha 18). Para trazer o raio para o espaço local da caixa, subtraímos a posição da caixa (linha 10) e transformar a posição relativa recém-formada pelo inverso da orientação da caixa (formada usando o conjugado de seu quaternion de orientação - melhor do que inverter matrizes!), junto com a direção do raio, dando-nos um novo raio temporário, definido dentro do quadro de referência do OBB. Podemos então fazer raycast versus um 'AABB' que está na origem (a caixa fica em sua própria posição de origem) e, se estiver colidindo, trazemos o ponto de colisão de volta ao espaço mundial realizando as operações opostas no ponto de colisão (adicionamos a posição novamente e depois giramos pela transformação mundial, vista na linha 18).

```
1  bool CollisionDetection :: RayOBBIntersection ( const Ray &r 2 const Transformar e transformar ,
                                                    ,
3              const OBBVolume e volume          , RayColisão e colisão ) {
4       Orientação Quaternion = worldTransform . GetWorldOrientation();
5       Posição do vetor3                  =mundoTransformar. GetWorldPosition();
6
7       Transformada Matrix3               = Matriz3 (orientação);
8       Matrix3 invTransform = Matrix3 (orientação. Conjugado ());
9
10      Vetor3 localRayPos = r . GetPosition() - posição;
11
12      Ray tempRay (invTransform * localRayPos, invTransform * r. GetDirection ());
13
14      bool colidido = RayBoxIntersection (tempRay, volume. GetHalfDimensions  Vetor3 ()      ,
15                      () colisão);                                              ,
16
17      if (colidiu) {
18            colisão. colididoAt = transformação * colisão. colidiuAt + posição;
19      }
20      retorno colidiu;
21 }
```

Detecção de Colisão::RayOBBIntersecção

Arquivo principal

O programa foi configurado para já usar as funções de raycasting - é por isso que elas já estavam lá, mas estavam apenas retornando false. O raycasting real é executado na função SelectObject na classe TutorialGame - observe que ele está chamando o método Raycast da classe GameWorld, que irá iterar sobre cada objeto de jogo na cena, e chamar o método RayIntersection que preenchemos anteriormente para ver se é colidindo ou não. Há muitas funções cujo conteúdo não abordamos aqui, então você pode querer investigar como elas funcionam e como tudo está interligado na classe TutorialGame, que possui um método UpdateGame chamado pela função principal, em um loop while que deve parecer bastante familiar ao que você viu no módulo anterior.

Conclusão

Embora o programa de demonstração pareça um tanto simples, aprendemos bastante ao fazê-lo. Em primeiro lugar, vimos como clicar em coisas no mundo - não existem muitos jogos em que você não clica em algo ou pelo menos determina o que está sob a mira ou o ponteiro do mouse, então isso por si só é importante. Em segundo lugar, vimos como esse mesmo mecanismo pode permitir aos programadores determinar se um objeto pode ver outro. A IA costuma usar esses testes para simular a visão e descobrir para onde ir ou o que atacar, então isso será útil quando entrarmos nos algoritmos de IA mais tarde. Por fim, também aprendemos os fundamentos por trás de algumas formas básicas de colisão e vimos como o cálculo de interseções com raios não exige muito esforço computacional. Usaremos essas formas com mais frequência à medida que entrarmos nos algoritmos de detecção e resolução de colisões.

Nos próximos dois tutoriais, vamos dar uma olhada em como simular movimentos lineares e angulares fisicamente precisos em nossos objetos, para que eles possam se mover de maneira realista e começar a construir a parte física real da nossa física. motor. Estaremos usando raycasting para empurrar objetos, então foi muito útil adicionar o para formar raios da câmera e selecionar objetos.

Trabalho adicional

1) Às vezes, queremos ignorar seletivamente certos tipos de objetos ao realizar raycasting - se os fantasmas no Pac-Man usassem raycasts para determinar se poderiam ver o jogador, não seria muito útil quase sempre devolver um pellet colecionável como o objeto mais próximo! No motor de jogo Unity, esse problema geralmente é atenuado através do uso de camadas de colisão, nas quais cada objeto é marcado com uma propriedade de 'camada', onde apenas certas combinações de camadas podem colidir ou sofrer raios.
Investigue a 'detecção de colisão baseada em camadas' do Unity e considere como você poderia implementar um sistema semelhante como uma pequena adição às classes GameObject e GameWorld.

2) É uma operação comum na IA do jogo determinar se o objeto A pode ver o objeto B. Tente usar o novo código de raycasting para ver qual GameObject no mundo pode ser visto na direção direta do GameObject selecionado.

3) Agora é um bom momento para se familiarizar com o recurso de desenho de linha de depuração - tente adicionar algumas chamadas a Debug::DrawLine para visualizar os raios sendo lançados pelo mouse/objetos.