Machine Translated by Google

Física - Fase Ampla e Fase Estreita

Introdução

O mecanismo de física que foi construído ao longo da série de tutoriais está ficando mais poderoso - ele pode detectar colisões entre volumes de colisão simples, resolvê-los e aplicar forças a pontos na superfície de uma forma, produzindo velocidade linear e angular. Como determinamos no último tutorial, testar cada objeto em relação a todos os outros objetos em nossa cena levará rapidamente a um grande desperdício de computação. Para aliviar isso, os motores físicos dividirão o processo de determinação e resolução de colisões em duas fases separadas, a fase ampla e a fase estreita. A fase ampla determina aproximadamente quais corpos físicos podem se cruzar e, em seguida, a fase estreita realiza a detecção real e resolve quaisquer colisões que realmente tenham ocorrido. Neste tutorial, investigaremos como implementar estruturas de dados adicionais na fase ampla do nosso mecanismo de física para determinar rapidamente pares de objetos que podem estar colidindo e passar os resultados para a fase estreita para resolução.

A Fase Ampla e a Fase Estreita

Na fase ampla, o mecanismo de física determinará quais objetos podem possivelmente colidir entre si e, em seguida, armazenará a colisão potencial do objeto em uma lista - isso geralmente é conhecido como par de colisão.
Então, na fase estreita, o motor itera sobre a lista de potenciais pares de colisão e determina se eles realmente estão colidindo e, em caso afirmativo, resolve a colisão. Isso significa que se a fase ampla for de alguma forma “inteligente” o suficiente para saber quais objetos podem potencialmente estar colidindo (ou, inversamente, quais definitivamente não podem estar colidindo), então podemos reduzir o número de verificações de colisão que fazemos para menos do que n^2 pior cenário que vimos no tutorial anterior.

Vamos pensar em um exemplo de alto nível do que poderia ser uma fase ampla. Se tentássemos fazer uma simulação da população do mundo inteiro, teríamos que verificar se o objeto que representa cada pessoa está colidindo com alguma outra pessoa na Terra - são muitas verificações de colisão.
No entanto, sabemos intuitivamente que não há possibilidade de alguém em Londres, no Reino Unido, estar colidindo com alguém em Xangai, na China, então por que analisar todas as verificações de colisão de todas as 8,5 milhões de pessoas em Londres, versus todas as 24 milhões de pessoas em Xangai, quando sabemos que os resultados serão sempre falsos? Se pudermos dividir nosso mundo em áreas discretas, onde sabemos que as áreas não podem se sobrepor e, portanto, os objetos dentro dessas áreas não podem se sobrepor, poderemos potencialmente economizar muitos testes de detecção de colisão.

Estruturas de Aceleração Espacial

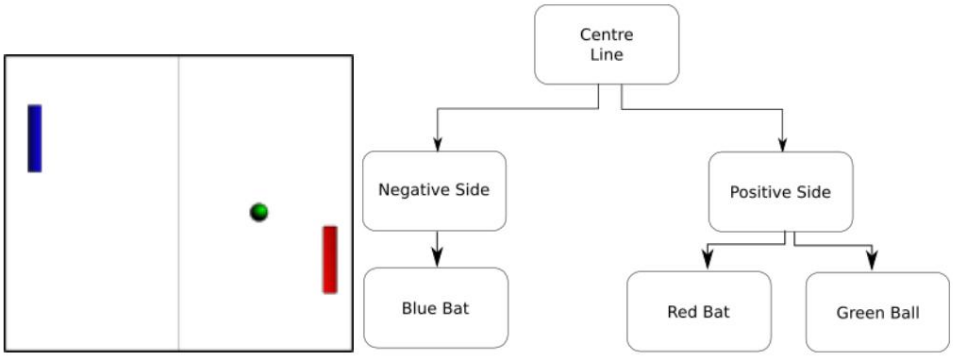
Para que a fase ampla desempenhe seu trabalho, ela precisa de algum método para determinar em quais áreas do mundo cada objeto está e descobrir se esses objetos em uma área específica podem colidir. Mas o que constitui exatamente uma “área”? E como vamos dividir o mundo neles? Existem várias maneiras pelas quais podemos dividir o mundo, amplamente categorizadas como estruturas de aceleração espacial - isto é, maneiras pelas quais podemos dividir o espaço para facilitar o raciocínio sobre ele.

Particionamento de Espaço Binário

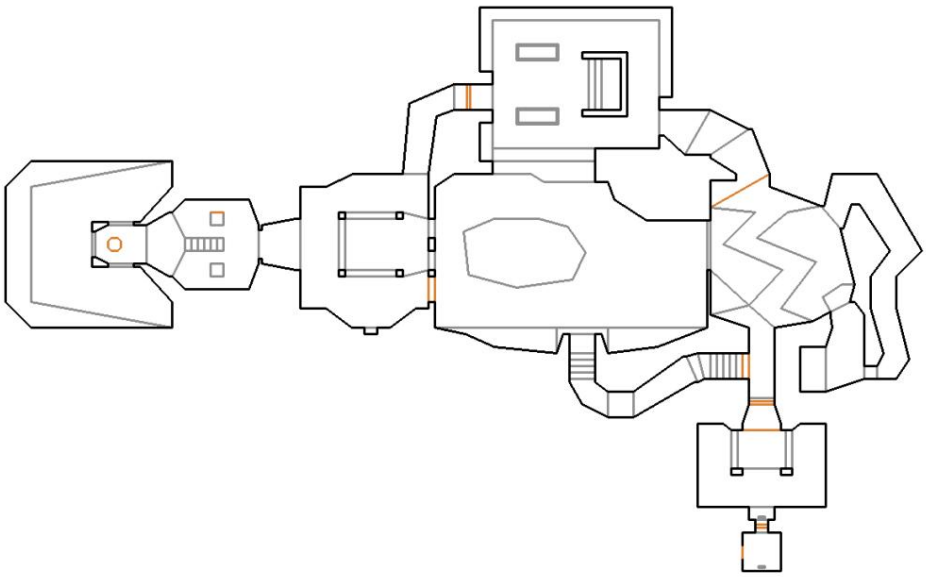
Em 1993, Doom, da id Software, demonstrou que era possível ter um mundo totalmente texturizado, aparentemente “3D”, renderizado de forma eficiente nos 386 computadores da época. Parte da magia da engenharia por trás de sua renderização foi o uso de particionamento de espaço binário para seu design de níveis. Este 'BSP' assume a forma de uma árvore, onde cada nó tem um plano divisório (ou linha infinita, já que Doom era realmente um jogo 2D com alguns algoritmos de renderização inteligentes para fazê-lo parecer 3D) e dois nós filhos, representando tudo.

Machine Translated by Google

atrás do avião e na frente do avião. Como um exemplo (muito) simples disso, aqui está um jogo simples de Pong, representado tanto visualmente quanto como um BSP muito simples que divide o mundo ao longo da linha de 'rede' do jogo:



Para testar colisões usando um BSP, percorremos recursivamente a árvore, determinando em que lado do plano está o objeto testado - se estiver atrás do plano do nó BSP, verificamos o nó esquerdo, e se estiver na frente, verificamos o nó direito - o 'binário' no BSP é que existem apenas dois nós. Quaisquer objetos que não estejam na mesma região BSP não podem estar colidindo e, portanto, não precisam ser verificados - em nosso exemplo simples acima, o taco azul não pode estar colidindo com a bola, pois não estão no mesmo meio espaço definido por o plano da raiz e, portanto, não precisamos verificar sua forma de colisão com o bastão vermelho ou com a bola. Poderíamos subdividir ainda mais o nó direito encontrando outro plano, criando outro par de nós filhos que representam subdivisões adicionais da área de jogo, mas estamos rapidamente superando nosso exemplo simples de Pong, já que não há outro 'plano' lógico que realmente faça senso. Em vez disso, aqui está o primeiro nível de Doom, visto de cima para baixo:



Cada uma dessas salas e corredores está cheia de demônios e itens colecionáveis, como saúde e munição - parte da eficiência de Doom estava em sua capacidade de determinar rapidamente quais itens, paredes e pisos desenhar. Para isso, a geometria do mundo foi dividida por meio de um BSP, para produzir diversos setores; regiões convexas do mundo colocadas nas folhas de um BSP, sendo cada plano divisório selecionado da geometria do mundo.

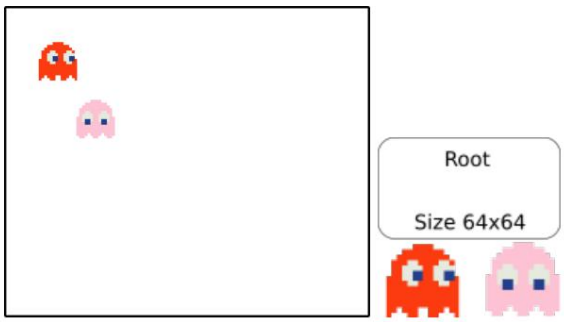
O processo de seleção de quais linhas geométricas usar para subdividir o mundo requer muito cuidado. Idealmente, cada divisão dividiria a geometria em seu nó pai exatamente pela metade, para fornecer a redução máxima no processamento ao determinar quais colisões podem acontecer. Os níveis em Doom (e na série posterior Quake) passariam por um longo processo de 'compilação' para produzir uma árvore perfeitamente balanceada, muitas vezes levando horas no hardware da época. Embora o hardware moderno possa calcular equações planas muito mais rapidamente do que em 1993, a quantidade de geometria num nível de jogo moderno também é muito maior, resultando num processo difícil de determinar um bom BSP, e que não

Machine Translated by Google

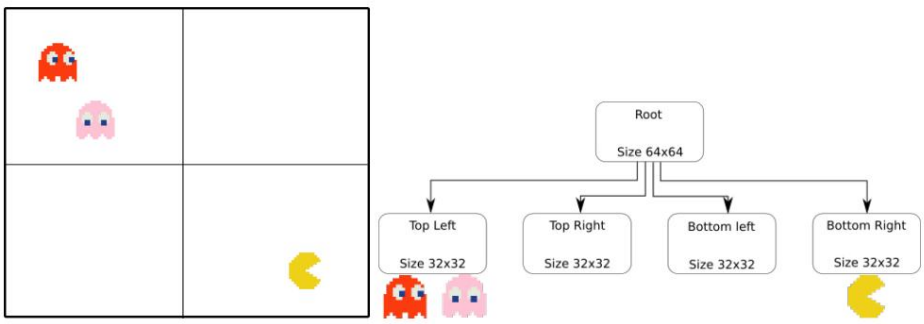
presta-se bem ao cálculo em tempo real. Compreender como um BSP funciona de maneira ampla ainda é benéfico, já que muitos jogos os usaram para geometria estática ao longo dos anos, tornando-os historicamente significativos.

Quadtrees Um

método popular de particionar o mundo do jogo é o Quadtree; assim chamado porque dividirá um nó pai em 4 nós filhos de tamanhos iguais em um determinado limite. Cada um dos nós da árvore tem um tamanho, e os filhos desse nó subdividem o espaço dos pais de forma que caibam dentro dele. Por esse motivo, você frequentemente verá quadtrees definidos com um tamanho que é uma potência de 2 em cada eixo - você pode pensar no processo de criação da árvore como sendo semelhante ao MIPMapping que vimos anteriormente, onde com subdivisões suficientes nós ' Acabaremos com uma estrutura de tamanho 1x1. Para ver isso em ação, vamos investigar um cenário de jogo pequeno, de um nível de jogo pequeno e restrito, contendo vários fantasmas e um personagem de jogador amarelo. Para criar a quadtree, cada fantasma/personagem é inserido na árvore, testando seu volume delimitador em relação ao tamanho do nó - somente se um personagem couber dentro de um nó ele desce naquele galho da árvore, semelhante a escolher o esquerdo /lado direito de uma árvore de pesquisa binária, dependendo se um valor é maior ou menor que o valor desse nó. Ao contrário da árvore binária, um nó geralmente pode conter vários valores, em vez de apenas um único valor - esses são os subconjuntos de objetos que devem ser testados quanto a colisões entre si, e somente se um objeto de jogo estiver no mesmo nó é que existe um chance de colisão. Se houver muitos objetos inseridos em um nó específico, esse nó será dividido e os objetos dentro dele serão redistribuídos entre seus nós filhos. Às vezes você verá nós quadtree chamados 'baldes', enquanto tentamos enchê-los até que comecem a transbordar. Vamos ver como funciona esse processo de preenchimento de baldes, com nosso exemplo de jogo fantasma e um tamanho máximo assumido de objeto de jogo por nó de 2. Inserir os dois primeiros objetos de jogo na quadtree é fácil, apenas ambos ficam no nó raiz:

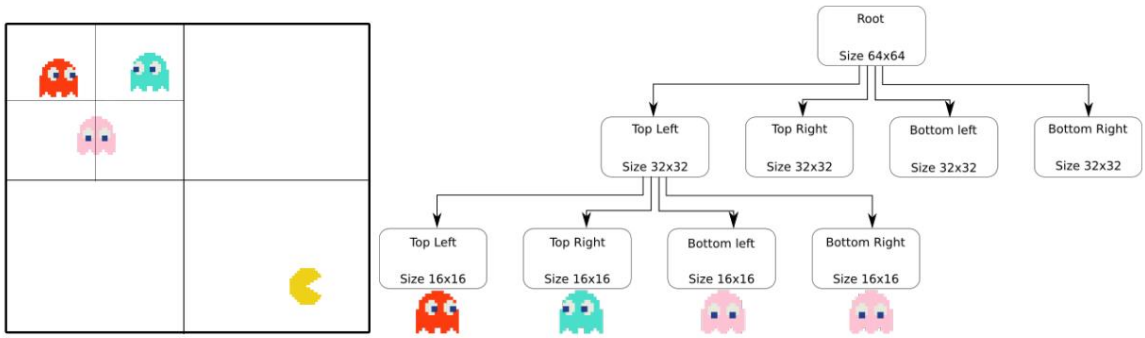


Quando chegamos à inserção do terceiro objeto, entretanto, ultrapassamos o limite do objeto da raiz e devemos dividi-lo em 4 segmentos, testando novamente cada objeto em relação aos tamanhos dos filhos para ver em qual nó eles se encontram:



Se executássemos uma etapa de detecção de colisão neste ponto, estaríamos testando apenas 1 colisão potencial - o fantasma vermelho e o fantasma rosa estão no mesmo segmento quadtree e poderiam estar colidindo.
Nosso personagem amarelo fica em seu próprio segmento quadtree, então não há nada com que ele possa colidir. Sem um quadtree, estaríamos realizando 3 verificações de colisão por quadro (vermelho vs rosa, vermelho vs jogador e rosa vs jogador). Se adicionarmos outro fantasma no canto superior esquerdo da área de jogo, atingiremos novamente o limite do que o nó pode conter e ele deverá ser dividido:

Machine Translated by Google

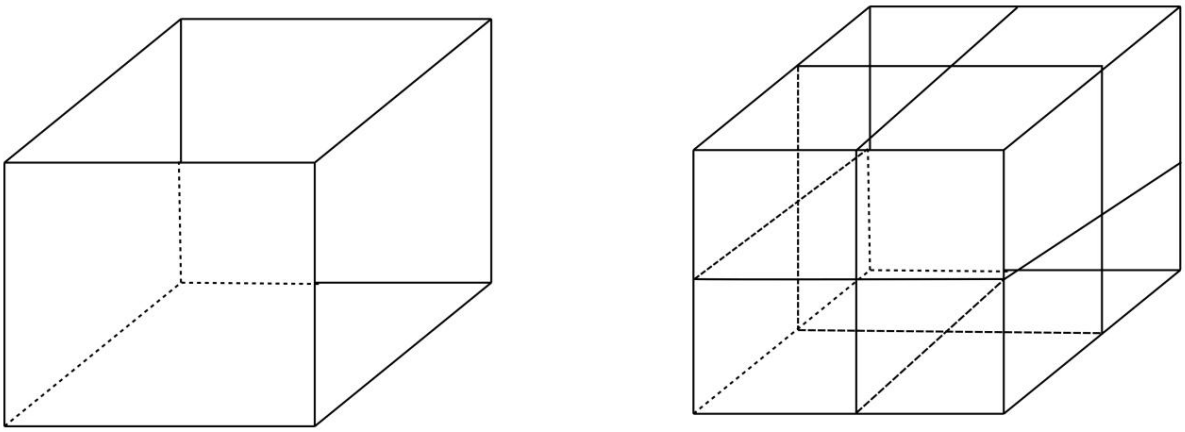


Com o fantasma azul inserido na quadtree, descobrimos um caso interessante para lidar com objetos com tamanho - depois que o nó superior esquerdo é subdividido mais uma vez, o fantasma rosa fica entre dois nós.
Como escolhemos a qual nó filho o fantasma rosa deve ser adicionado? A resposta é que ele deve ser adicionado a ambos os nós - se houvesse qualquer outro objeto em qualquer um dos nós, ele poderia estar colidindo com o fantasma rosa, portanto, a única maneira confiável de detectar possíveis colisões é adicioná-lo a ambos.

Depois que o fantasma rosa for adicionado à quadtree, quantas colisões nossa fase ampla terá que tentar detectar? Nenhum! Não há nenhum nó com vários objetos e, portanto, não há chance de colisões. Se não usássemos algum tipo de estrutura de aceleração espacial, estaríamos calculando 6 detecções de colisão por quadro (vermelho vs rosa, vermelho vs azul, rosa vs azul, vermelho vs personagem, rosa vs personagem e azul vs. personagem), portanto, isso pode ser uma economia significativa nos casos em que os objetos do jogo têm volumes de colisão complexos (AABBs ou cascos cóvexos arbitrários).

Octrees

O exemplo acima de quadtree leva em consideração apenas 2 dimensões. Isso é adequado para jogos que acontecem principalmente em uma arena plana; mesmo algo como Rocket League, embora seja '3D' no sentido de que os objetos podem se mover em três dimensões completas, é bastante 'plano', pois é mais provável que os objetos do jogo estejam na frente ou atrás de outro, em vez de acima ou abaixo, portanto é aceitável descontar o terceiro eixo para fins de verificação de colisão de fase ampla. Mas que tal algo mais espacialmente complexo, como MineCraft, onde o personagem A pode ter as mesmas coordenadas x e z do personagem B, mas estar separado por quilômetros de túneis subterrâneos e rochas no eixo y? Podemos estender o conceito de quadtree para 3 dimensões, formando uma octree. Como o nome sugere, em vez de um nó ser dividido em 4 filhos de tamanhos iguais, o nó octree será dividido em 8 filhos de tamanhos iguais, dividido da seguinte forma:



Isto permite que nós que estão acima ou abaixo uns dos outros, mas separados por uma distância significativa, sejam divididos em nós separados, de modo que o teste de colisão em fase ampla não os considere como uma colisão potencial. O processo real de manipulação e inserção de objetos na octree é exatamente o mesmo que na octree, exceto que testamos colisões contra um AABB completo por nó da octree, em vez de apenas uma caixa quadrada 2D plana.

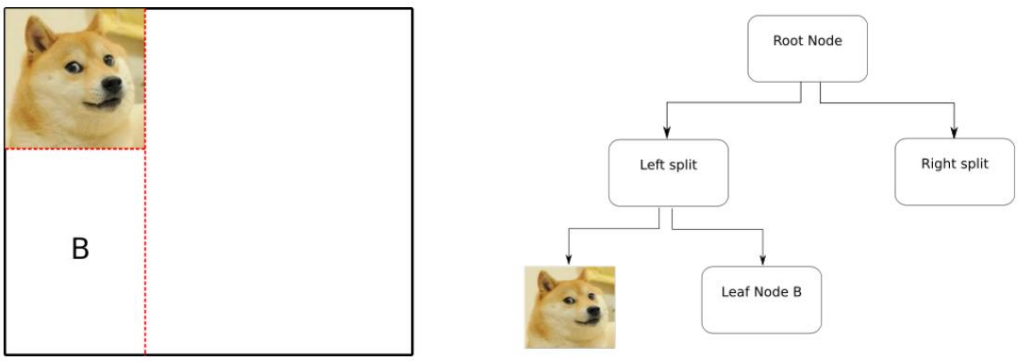
Machine Translated by Google

Árvores kD

Nas quadtree e octrees acima, um nó sempre foi dividido em volumes de tamanhos iguais. Existe uma estrutura semelhante conhecida como kDTree, onde a expansão do nó pode ser dividida para 'ajustar' melhor os nós à área dos objetos que estão sendo adicionados.

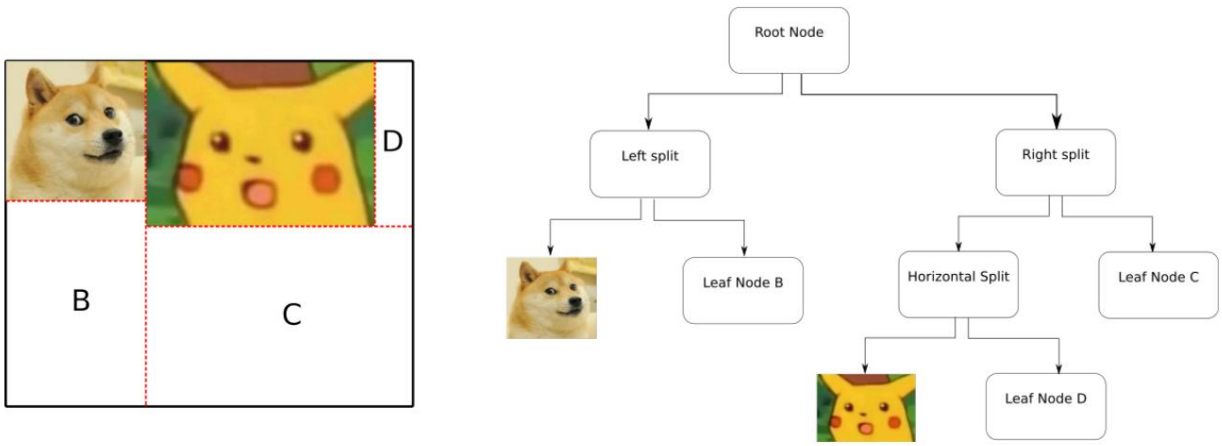
A principal desvantagem de um kDTree é que, ao ajustar os nós da árvore exatamente ao volume de um objeto, se esse objeto se mover de alguma forma, a árvore deve ser reconstruída, enquanto um quad ou octree só precisa de seus nós reconstruídos quando um objeto se move o suficiente. para entrar em outro nó filho. Isso torna o kDTrees mais difícil de usar e não é totalmente intuitivo como um nó deve ser dividido ou o que fazer se mais nós precisarem ser inseridos na árvore após ela ser construída. Por estas razões, os kDTrees não são frequentemente usados como uma estrutura de aceleração espacial para cálculos físicos.

Então, por que usaríamos um kDTree? Embora não sejam bons para a física, eles são bons para subdividir um volume em situações totalmente estáticas ou para determinar se um objeto de volume A pode caber em qualquer lugar dentro da árvore (existe um nó folha não utilizado de pelo menos tamanho A em algum lugar do estrutura de árvore?).
Isso é frequentemente usado na criação de um atlas de textura, uma textura 2D contendo uma série de subtexturas, frequentemente usada para jogos 2D da 'velha escola', ou para agrupar todas as texturas usadas na interface do jogo, de modo que o toda a tela da UI pode ser renderizada com apenas uma única ligação de textura. Nesse caso, estamos pegando uma textura de origem e, em seguida, copiando seu conteúdo para uma textura 2D nova e maior, e determinando a localização com base em uma pesquisa do kDTree. À medida que cada textura a ser adicionada é processada através do kDTree, é encontrado um nó que se ajustará à textura, onde descerá para o próximo nível, até que um nó folha seja encontrado - o nó será então dividido em dois, com o nova textura sendo adicionada a um nó e o outro permanecendo como um nó vazio para adicionar novas texturas posteriormente.



Após uma iteração de inserção de uma área 2D no kDTree, obteríamos a seguinte árvore

Isso aproveita os pontos fortes do kDTree, já que cada 'nó' é ajustado exatamente às dimensões do objeto que contém (uma textura 2D imutável) e pode colocá-lo conforme necessário dentro de seu volume para evitar deixar qualquer espaço livre ao redor de um nó isso pode ser difícil de preencher.

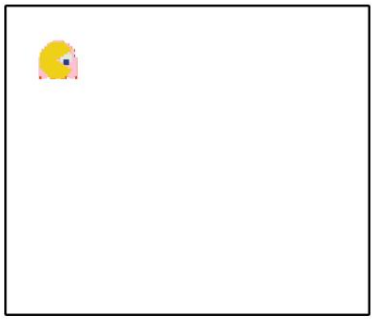


A segunda área 2D a ser adicionada é muito grande para a área de divisão esquerda e, em vez disso, é adicionada ao nó filho direito, que é dividido ainda mais, dando-nos o nó C

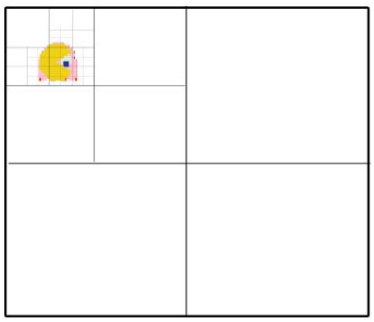
Machine Translated by Google

Considerações sobre árvores

Os exemplos de árvores acima ajudam a mostrar os benefícios das quadtrees e octrees, mas escondem algumas coisas importantes que devem ser consideradas ao gerar estruturas de aceleração para nossos sistemas físicos. Pense em um caso em que, seja por azar com a integração física da posição, ou por um erro no código de geração de objetos, acabamos em uma situação em que vários objetos terminam na mesma posição:



Neste exemplo, um fantasma rosa, um fantasma vermelho e um personagem amarelo estão todos na mesma posição. Se assumirmos que há no máximo 2 objetos contidos em um nó, devemos dividir o nó em 4 filhos:



O problema é que, como existe essa sobreposição, cada nó filho dividido ainda contém 3 objetos e será dividido novamente. Embora essa divisão e redução dos tamanhos pela metade signifique que eventualmente os nós filhos ao redor da borda dos objetos de colisão pararão de se dividir, cada nó 'dentro' dos objetos será subdividido para sempre! Por esta razão, geralmente definimos uma profundidade máxima de uma octree ou quadtree; um nó folha na profundidade máxima abaixo da árvore não será mais subdividido e, em vez disso, apenas armazenará mais objetos dentro dele do que um nó folha 'mais acima' na árvore. Isso ainda faz com que muitos nós sejam criados no pior cenário de sobreposição total de objetos, mas coloca um limite máximo no tempo necessário para gerar uma árvore, permitindo que a próxima atualização física detecte a colisão e separe os objetos.

Outra consideração a ser feita é quantos nós filhos realmente instanciar ao dividir um nó pai. No exemplo acima, os nós filhos superior direito, inferior esquerdo e inferior direito da raiz foram instanciados junto com o canto superior esquerdo, onde estavam os 3 objetos. São possivelmente 3 construções de objetos e, dependendo do modelo de memória utilizado, talvez 3 chamadas para alocar memória, que são completamente desperdiçadas neste caso, pois não há nada nelas. Em vez disso, pode ser melhor deixar os nós 'não preenchidos' não alocados até que algo os exija, e somente se houver uma colisão com esse nó, instancie-o.

Tamanho e posição do nó

Na discussão acima sobre adiar a alocação de nós até que seja necessário, você pode estar pensando como um teste de colisão poderia ser feito em um nó que não existe - certamente precisamos armazenar uma posição e tamanho do nosso nó em uma estrutura ou classe em algum lugar antes de podermos realizar testes de interseção nele? Talvez não! A estrutura de quad e octrees é tal que ganhamos duas propriedades: primeiro, podemos usar funções recursivas para percorrer a árvore a partir da raiz para realizar operações e, segundo, podemos derivar as propriedades de um nó filho de seu pai, como desde que saibamos 'qual' criança é (canto superior esquerdo

Machine Translated by Google

ou nó superior direito, para um exemplo de quadtree). Juntos, isso significa que podemos calcular as propriedades de um nó dinamicamente enquanto caminhamos pela árvore! Se soubermos que o tamanho da raiz é 64, podemos calcular rapidamente que cada filho 3 níveis abaixo tem tamanho 8. A posição só é mais difícil de determinar e pode ser mais fácil de determinar usando parâmetros em funções recursivas para calculá-la conforme necessário .

Uso de memória

Você também deve estar se perguntando como armazenar os objetos em um nó quadtree. Uma solução simples pode ser usar um std::vector para cada nó e colocar ponteiros para os objetos nele à medida que são inseridos nos nós folha. Isso certamente funciona, mas pode ser ineficiente em termos de memória. Nos bastidores, um std::vector usa uma alocação de memória para armazenar seus objetos e, sempre que essa alocação de memória é preenchida, ele deve solicitar ao sistema operacional para realocar um pedaço maior de memória. Pedir memória ao sistema operacional é lento, portanto, ter que solicitá-la em cada construção de nó e toda vez que a alocação é preenchida leva tempo e retarda a criação de nossa estrutura de 'aceleração'! pode ser melhor usar uma única alocação grande para a árvore inteira, que pode ser passada como um parâmetro nas funções recursivas, onde cada nó folha que armazena um objeto apenas é colocado na estrutura única e armazena apenas um índice para o primeiro elemento dentro dele e uma contagem de quantos elementos estão dentro dele. Isso, combinado com a ideia de que talvez nem precisemos armazenar a posição e o tamanho de um nó, pode tornar um nó quadtree ou octree uma estrutura de dados muito eficiente.

Objetos estáticos e dinâmicos

Grandes mundos de jogo podem ser construídos com centenas de objetos estáticos, que não necessariamente se movem (ou seja, têm massa inversa de 0), mas têm volumes de colisão para impedir o progresso do jogador. Pense em todas as paredes e plataformas que compõem os níveis de um jogo FPS como Call of Duty ou Overwatch - elas nunca se movem ou são destruídas, mas impedem que as balas e os jogadores se movam através delas. Então, esses jogos constroem toda uma estrutura de aceleração espacial para centenas de objetos a cada quadro? Provavelmente não - levaria muito tempo em cada quadro para calcular uma resposta que seria a mesma em todos os quadros, exceto para os jogadores e projéteis em movimento. Por esta razão, pode ser que você tenha não uma, mas duas estruturas de aceleração espacial - uma para objetos estáticos (as paredes e pisos do mundo) e outra para objetos dinâmicos (os jogadores, seus projéteis, os NPCs, e assim por diante). Então, a estrutura de aceleração para objetos estáticos só precisa ser reconstruída se algo mudar fundamentalmente no mundo e em seus objetos estáticos (talvez o objetivo seja explodir uma parede, que deve ser removida da estrutura estática após a conclusão). Para testar um objeto dinâmico versus a estrutura de aceleração estática, em vez de inseri-lo diretamente em uma lista, simplesmente precisamos de uma função para retornar os nós nos quais o objeto seria inserido e, em seguida, realizar os testes de colisão nos objetos dentro dele dessa forma. - como já sabemos que os objetos estáticos possivelmente não estão colidindo, então esta é apenas uma operação O(n), em vez de O(n para cada área com vários objetos dentro dela. ^{2}) teste que normalmente realizaríamos

Objetos adormecidos

Também podemos usar nosso conhecimento sobre o estado dos objetos físicos em nossa simulação para reduzir o número de testes de colisão que ocorrem a cada quadro. Em muitos jogos, o nível de jogo em que o jogador se encontra é preenchido com um número predefinido de inimigos que ele deve enfrentar - um designer de níveis poderia ter adicionado 10 inimigos atrás de uma porta, que ficam parados e não fazem nada até que o jogador abra a porta. porta, momento em que eles se tornam 'ativos' e atacam o jogador. Esses objetos não se enquadram exatamente na definição de objeto 'estático' definido anteriormente; embora eles fiquem parados até o jogador entrar na sala, eles eventualmente se moverão e, portanto, não devem ser 'incorporados' na mesma estrutura de aceleração das paredes e do chão da sala em que estão. deve inserir todos os 10 inimigos em uma estrutura 'dinâmica' e, em seguida, testar as colisões entre eles em cada quadro? Talvez não! Neste exemplo, os inimigos não estão se movendo, nem girando, nem fazendo nada que possa causar uma colisão entre eles. Podemos afirmar que tal objeto está 'em repouso' e, em seguida, estender isso ainda mais para dizer que dois objetos que estão em repouso não podem estar colidindo e, portanto, não faz sentido executar a detecção de colisão entre eles. Se, para qualquer área específica da nossa estrutura de aceleração, não houver objetos em movimento nela, então não faz sentido executar a detecção, pois ela nunca encontrará um par de objetos em colisão!

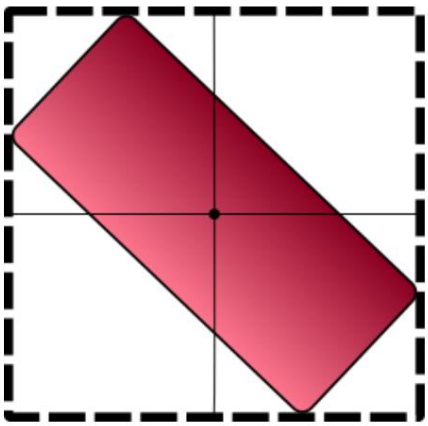
Machine Translated by Google

Ao inserir um objeto em uma estrutura de aceleração, podemos verificar se o objeto está ‘adormecido’ ou não – ou seja, se ele irá se mover de alguma forma neste referencial. Podemos então também armazenar um estado adicional no nó da estrutura de aceleração ou não - seja atualmente um nó 'acordado' ou não. Podemos assumir que por padrão ele não está 'acordado' e só muda para esse estado se um objeto acordado for inserido nele. Se, após a inserção de todos os objetos, não houver nenhum objeto acordado em um nó específico, ele poderá ser ignorado ao determinar possíveis pares de colisão, pois não pode haver nenhum no nó. Podemos estender isso ainda mais e definir o estado 'acordado' de um nó como verdadeiro quando um nó filho fica acordado devido à inserção de um projeto ativo nele. Isso significaria então que, se um nó não estiver acordado, nenhum processamento recursivo adicional deverá ocorrer em nenhum de seus filhos, pois eles também não estarão acordados, e ramos inteiros da árvore poderão ser ignorados ao determinar pares de colisão.

Inserindo Objetos

Nesta série de tutoriais até agora, vimos vários tipos de volumes de colisão - esferas, caixas alinhadas aos eixos e caixas delimitadoras orientadas. Como determinamos em quais nós da nossa estrutura de árvore estão nossos volumes de colisão? Uma coisa que você provavelmente notou é que os nós em uma quadtree e uma octree estão alinhados ao eixo, o que significa que podemos determinar sobreposições entre a forma da nossa árvore e nossos objetos usando um teste de caixa delimitadora alinhada ao eixo. Podemos ir mais longe e dizer que, no nível amplo da fase, tudo é uma caixa delimitadora alinhada ao eixo; vimos que os testes de sobreposição AABB podem ser bastante baratos (o método AABBTest contém apenas algumas linhas de código!), e como o objetivo da fase ampla é determinar rapidamente possíveis colisões, podemos gerar nossa estrutura de aceleração mais rapidamente se usamos um teste rápido.

Você pode estar se perguntando como transformar nossas outras formas de colisão em AABBs para o teste de colisão de fase ampla. Fazer um AABB rápido a partir de uma esfera delimitadora é simples - assumimos que cada um dos eixos de metade do tamanho do teste AABB são iguais ao raio da esfera e usamos a posição do objeto normalmente. Que tal uma caixa delimitadora orientada? Como sempre acontece com esse tipo de volume, é complicado. Ou melhor, vamos usar um truque para calculá-lo. Se tivermos a orientação de um objeto como uma matriz 3x3 M, podemos calcular os meios tamanhos de um AABB que irá encapsular o OBB transformando os meios tamanhos do objeto pelo absoluto da matriz M - isto é, definindo cada valor nele como positivo .



Usando esses cálculos, podemos sempre ter um conjunto de tamanhos AABB adequados para uso na fase ampla para cada um dos corpos físicos em nossa simulação, que pode então usar o volume real de colisão na fase estreita, uma vez encontrados os prováveis pares de colisão. .

Escolhendo uma estrutura

Lendo essas estruturas comuns, você provavelmente notou que todas elas são estruturas em árvore, que dividem o mundo recursivamente. Assim como podemos usar uma árvore binária para criar uma estrutura heap que nos ajuda a classificar e pesquisar arrays de maneira eficiente, podemos usar exatamente os mesmos conceitos para nos ajudar a determinar quais objetos estão relacionados espacialmente - eles teriam um pai comum. Mas qual estrutura usar?
A escolha realmente se resume ao formato geral do seu mundo de jogo e à frequência com que ele muda.
Os testes de avião são rápidos, então inicialmente pode parecer que os BSPs são a melhor escolha. No entanto, a sua eficiência é altamente dependente de quais planos são usados para dividir o mundo - escolhas erradas levarão a uma árvore muito "desequilibrada" que não reduz muito o número de casos de teste por nível de árvore. Escolhendo estes

Machine Translated by Google

planejar e construir a árvore não é particularmente rápido e não é adequado para executar todos os quadros, e assim BSPs geralmente são usados apenas para dividir geometria 'estática' que nunca se move ou muda durante o curso de um jogo - a série de jogos Quake da id Software usava árvores BSP para representar a geometria mundial, aumentada com uma matriz de visibilidade adicional que armazenava quais nós de árvore podiam ser vistos de outros nós, permitindo acesso muito rápido a quais partes do nível desenhar ou testar colisões.

Para cenas mais dinâmicas, como a cena de teste que construímos ao longo desses tutoriais, quadtrees e octrees são candidatos mais adequados - quads e boxes ainda são testes fáceis, mas sem alguns das desvantagens do uso de aviões - mover um avião poderia mudar drasticamente a forma de um BSP, pois a geometria é movida para um lado ou para outro, mas mover a caixa delimitadora de uma forma para alguma nova é improvável que os nós mudem mais do que um pequeno número de nós folha. Escolhendo entre quadtrees e octrees depende principalmente da forma geral do nível do jogo - em um mundo razoavelmente 'plano' como no estádio da Rocket League, um quadtree seria bom (a única coisa que provavelmente estará no ar é um único bola), enquanto algo que acontece em um bloco de torre seria melhor usando uma octree - o a capacidade de dividir o eixo y torna-se útil quando a geometria é empilhada uma sobre a outra.

Código do Tutorial

Para demonstrar a fase ampla e a fase estreita, substituiremos o método BasicCollision-Detection anterior da classe PhysicsSystem por dois novos métodos, apropriadamente chamados Fase Ampla e Fase Estreita. Para atuar como nossa estrutura de aceleração espacial, vamos implementar uma quadtree simples - isso será suficiente para demonstrar como a inserção de objetos baseada em árvore poderia funcionar, e a implementação que utilizamos aqui pode ser melhorada e desenvolvida em muitas áreas.

Classe QuadTreeNode

Para tornar as coisas interessantes, implementaremos nosso quadtree como um tipo de modelo, permitindo-nos reutilizar o código quadtree para qualquer outra coisa que possamos imaginar. Para fazer isso, precisamos de três modelos classes - QuadTreeNode para representar uma parte individual da nossa árvore (que será preenchida com um número de QuadTreeEntries, cada um com uma posição e tamanho) e QuadTree para conter a própria árvore. Como todos os modelos devem ser definidos dentro de um cabeçalho, as coisas são um pouco mais complexas do que o normal para descrever, mas começaremos com um esboço da classe QuadTreeNode:

```
1  espaço para nome NCL {
2  usando o namespace NCL :: Maths ;
3          espaço para nome CSC8503 {
4                  modelo < classe T>
5                  classeQuadTree ;
6
7                  modelo < classe T>
8                  estrutura QuadTreeEntry {
9                          Vetor3 pos;
10                         Tamanho do vetor3;
11                         Objeto T;
12
13                         QuadTreeEntry ( Tobj ,          Vetor3 pos, objeto      Tamanho do vetor3) {
14                                 = obj;
15                                 isto -> pos = pos;
16                                 este -> tamanho = tamanho;
17                         }
18                 };
19
20                 modelo < classe T>
21                 classeQuadTreeNode {
22                 público :
23                         typedef std :: função <
24                                 void (std :: list < QuadTreeEntry <T > >&) > QuadTreeFunc ;
```

Machine Translated by Google

```
25              protegido :
26                  classe amiga QuadTree <T >;
27
28                  QuadTreeNode() {}
29                  QuadTreeNode (Vetor2 pos, = nullptr;          Tamanho do vetor2) {
30                      crianças
31                      isto -> posição = pos;
32                      este -> tamanho = tamanho;
33                  }
34                  ~QuadTreeNode() { excluir []filhos;}
35              protegido :
36                  std :: lista < QuadTreeEntry <T >> conteúdo ;
37
38                  Posição do vetor2;
39                  Tamanho do vetor2;
40
41                  QuadTreeNode <T >* filhos;
42              };
43          }
44}
```

Cabeçalho da classe QuadTreeNode

Cada nó em nossa quadtree conterá uma lista de algum modelo do tipo T (vamos usá-lo para armazenar GameObjects, mas pode ser qualquer coisa!), junto com as posições e tamanhos do AABB que representa o objeto inserido (na estrutura QuadTreeEntry), e terá uma posição e um tamanho descritos em 2D. Como cada nó pode em algum momento ser dividido, também teremos um ponteiro para alguns nós filhos - estes devem seriam ponteiros, caso contrário eles sempre seriam instanciados, e nossa quadtree construiria recursivamente novos nós para sempre assim que instanciamos um. Até agora, a única outra coisa importante a notar é outro typedef - isso nos dá uma forma abreviada para um ponteiro de função, que nos permitirá operar no conteúdo do nosso quadtree. Faremos isso através do método OperateOnContents, que deve ser adicionado à parte pública da classe QuadTreeNode:

```
1 void OperateOnContents (QuadTreeFunc & func) {
2       se (filhos) {
3           for ( int i = 0; i < 4; ++ i ) {
4               crianças [eu]. OperateOnContents ( func );
5           }
6       }
7       outro {
8           if (! conteúdo. vazio ()) {
9               função (conteúdo);
10          }
11      }
12}
```

Método QuadTreeNode::OperateOnContents

Nosso quadtree está sendo projetado com a suposição de que o único conteúdo real dele estará na folha nós, então precisamos verificar se algum nó da árvore quádrupla em particular tem filhos e, se tiver, recursivamente chame o método até cairmos em alguns nós folha - para evitar a chamada de qualquer função passada em listas vazias, também verificamos se há realmente algum conteúdo antes de chamar o passado ponteiro de função também (linha 41).

Em algum momento, um nó quadtree terá que ser dividido em 4 nós filhos que, quando combinados, preencherão a mesma área (estamos, de certa forma, tesselando nossa área). Para lidar com isso, precisamos adicionar outro novo método para a classe QuadTreeNode, Split:

```
1 divisão nula () {
```

Machine Translated by Google

```
    Vector2 halfSize = tamanho / 2,0 f ;
    filhos = novo QuadTreeNode <T >[4];
    filhos [0] = QuadTreeNode <T >( posição +
        Vector2 (- halfSize .x , metade do tamanho . y ) halfSize );
    filhos [1] = QuadTreeNode <T >( posição +
        Vector2 ( halfSize .x halfSize ); , metade do tamanho . você) ,
    filhos [2] = QuadTreeNode <T >( posição +
        Vector2 (- halfSize .x , - metade do tamanho . y ) halfSize );
    filhos [3] = QuadTreeNode <T >( posição +
        Vector2 (halfSize.x - halfSize.y) halfSize); ,
}
```

Método QuadTreeNode::Split

Também estamos assumindo que cada quadtree terá 4 ou 0 filhos, todos instanciados de uma vez - usando alguma adição e subtração, podemos definir o deslocamento de 4 pontos do centro de um nó que seus nós filhos ficarão posicionados para preencher a área corretamente.

A única outra coisa que nosso QuadTreeNode precisa fazer é permitir que objetos sejam inseridos nele. Eram assumindo que cada nó tem uma capacidade máxima, que, se atingida, fará com que o nó se divida, e mova seu conteúdo para seus novos nós filhos. Tudo isso é tratado com uma função final, Insert, que pegará algum objeto, em uma posição do mundo, com um determinado tamanho, e tentará inserir em sua própria lista ou nas listas de seus nós filhos. Adicione este método à parte pública do Classe QuadTreeNode:

```
inserção vazia (T e objeto, 2 3 4 5 6 7 const Vector3 e objectPos,
    const Vector3 & objectSize , int profundidadeLeft , if (! int tamanhomáximo) {
    CollisionDetection :: AABBTest ( objectPos ,
        Vector3 (posição .x 0, posição .y), objectSize,
        Vetor3 ( tamanho .x 1000,0 f tamanho . y, ))) {
        retornar ;
    }
    if (filhos) { // não é um nó folha, apenas desce na árvore
        for ( int i = 0; i < 4; ++ i ) {
            crianças [eu]. Inserir (objeto, objectPos, objectSize,
                profundidadeEsquerda - 1 , tamanho máximo );
        }
    }
    else { // atualmente um nó folha , pode apenas expandir
        conteúdo. push_back (QuadTreeEntry <T >(objeto, objectPos, objectSize));
        if (( int ) conteúdo. size () > maxSize && profundidadeLeft > 0) {
            se (! filhos) {
                Dividir ();
                // precisamos reinserir o conteúdo até agora!
                for ( const auto & i: conteúdo) {
                    for (int j = 0; j < 4; ++ j ) {
                        entrada automática = eu;
                        crianças [ j ]. Inserir (entrada.objeto,entrada.pos,
                            entrada. tamanho , profundidadeLeft - 1 maxSize );
                    }
                }
                conteúdo. claro (); // conteúdo agora distribuído !
            }
        }
    }
}
```

Método QuadTreeNode::Inserir

Machine Translated by Google

A primeira coisa que um nó de árvore quádrupla deve fazer é um teste de interseção (linha 3), para ver se o objeto que está sendo adicionado tem um AABB (representado pelas variáveis objectPos e objectSize) que se sobrepõe ao AABB por si só - embora nossa quadtree seja realmente 'plana', vamos estendê-la artificialmente no eixo y, e use x e y da posição e tamanho do nó como as coordenadas x e z do mundo (isso é bom para um mundo 3D de cima para baixo, como o código do tutorial, mas para um jogo paralelo, podemos apenas usar o x e coordenadas y como estão). Se os AABBs não se sobrepuserem (usando o código de teste de interseção, adicionamos alguns tutoriais atrás), podemos simplesmente retornar - se um objeto não se sobrepõe a um nó, ele também não pode se sobrepor a nenhum dos nós. os filhos desse nó (linha 6). Se o objeto se sobrepuser, o nó precisará tentar recursivamente e adicione o objeto aos seus filhos (se não for um nó folha, na linha 8), ou adicione o objeto à sua própria lista de objetos armazenados (linha 15). Ainda não terminamos - um nó de árvore quádrupla tem um número máximo de objetos que pode conter e, uma vez excedido, o nó deve se dividir (linha 18), a menos que cheguemos à profundidade máxima da árvore, estamos permitindo a quadtree (linha 16). Se o nó divide, ele deve chamar Insert recursivamente para cada objeto contido anteriormente, para tentar adicioná-lo lista de cada criança. Feito isso, o nó pode remover seu próprio conteúdo, pois eles estarão em algum lugar nos nós filhos (linha 27).

Classe QuadTree

A classe QuadTree em si é bastante simples, pois é principalmente um wrapper em torno de um QuadTreeNode que será a raiz da nossa árvore. Definimos a classe na mesma configuração de namespace do QuadTreeNode e, assim como aquela classe, devemos declarar que nosso QuadTree é um tipo de modelo (linha 4). Precisamos apenas de 3 variáveis de membro para nossa demonstração simples de quadtree - o nó raiz e inteiros para representar o profundidade máxima que a árvore poderá estender e um tamanho máximo para limitar quantos objetos deve estar contido em cada nó. O construtor do QuadTree receberá um Vector2 que representa as dimensões da árvore e números inteiros padrão para o tamanho e profundidade. O resto de os métodos públicos que precisaremos são mais ou menos métodos de 'passagem' para inserir objetos em a raiz e receber uma função para chamar qualquer conteúdo da nossa árvore - isso fará mais sentido quando veremos esse recurso em ação em breve.

```
1 espaço para nome NCL {
2 usando o namespace NCL :: Maths ;
3 espaço para nome CSC8503 {
4 modelo < classe T>
5 classeQuadTree {
6 público :
7 QuadTree (tamanho do vetor2 int maxSize = 7, int maxDepth = 6 root = ,
8 QuadTreeNode <T >( Vector2 () tamanho ); ,
9 isto -> maxDepth = maxDepth ;
10 this -> maxSize = maxSize ;
11 }
12 ~QuadTree(){}
13 void Insert (objeto T, const Vector3 & const Vector3 & pos , raiz .
14 Inserir (objeto, pos, maxSize); tamanho, profundidade máxima,
15 }
16 vazio OperateOnContents (
17 nome do tipo QuadTreeNode <T >:: QuadTreeFunc func) {
18 raiz . OperateOnContents ( func );
19 }
20 protegido :
21 QuadTreeNode <T >* raiz;
22 int maxDepth;
23 int maxSize;
24 };
25 }
26}
```

Cabeçalho da classe QuadTree

Machine Translated by Google

Classe GameObject

Isso é tudo que precisamos para construir uma quadtree, mas usá-la exigirá um pouco mais de trabalho. Precisamos ser capaz de transformar qualquer um dos volumes de colisão que usamos para nossos GameObjects em um AABB para inserção na quadtree - a posição AABB é fácil (é apenas a posição do objeto), mas obter os meios tamanhos da caixa exigem um pouco mais de esforço. Para fazer isso, precisamos adicionar alguns novos métodos e uma variável para a classe GameObject - UpdateBroadphaseAABB preencherá os meios tamanhos corretos em a variável broadphaseAABB, e GetBroadphaseAABB a retornará (como referência, permitindo-nos também obter uma resposta verdadeira ou falsa sobre se temos um volume de colisão).

```
1 público :
2 bool GetBroadphaseAABB (Vetor3 e outsize) const ;
3 nulo UpdateBroadphaseAABB();
4 protegido :
5 Vetor3 fase largaAABB;
```

Cabeçalho da classe GameObject

Aqui está o conteúdo das duas funções. Observe que poderíamos, em vez disso, adicionar um método virtual a a classe CollisionVolume para reduzir bastante a quantidade de código que precisaríamos aqui, mas isso fará com que cada volume ganha um ponteiro vtable, o que podemos querer evitar. Obter as extensões de uma AABB é claramente trivial (linha 14) e não muito mais complicado para uma esfera (linha 18), mas os OBBs exigem que usemos o valores absolutos de uma matriz de rotação (linha 22+23), que transformará os meios tamanhos do espaço local em maiores que envolverão qualquer rotação no espaço mundial.

```
1 bool GameObject :: GetBroadphaseAABB ( Vector3 e outSize ) const {
2         if (!boundingVolume) {
3                 retorna falso ;
4         }
5         outSize = fase amplaAABB;
6         retornar verdadeiro ;
7}
8
9 void GameObject :: UpdateBroadphaseAABB () {
10 if (!boundingVolume) {
11                retornar ;
12        }
13        if (boundingVolume -> type == VolumeType :: AABB) {
14                fase largaAABB =
15                        (( AABBVolume &)*boundingVolume ). GetHalfDimensions();
16        }
17        else if (boundingVolume -> type == VolumeType :: Sphere) {
18                float r = (( SphereVolume &)*boundingVolume ). ObterRaio();
19                fase largaAABB = Vetor3 (r                    ,   R ,   r);
20        }
21        senão if (boundingVolume -> type == VolumeType :: OBB) {
22                Matriz3 mat = transformação. GetWorldOrientation(). ParaMatriz3();
23                tapete = tapete. Absoluto ();
24                Vetor3 halfSizes =
25                        (( OBBVolume &)* limitanteVolume ). GetHalfDimensions();
26                broadphaseAABB = mat * halfSizes ;
27        }
28}
```

Arquivo de classe GameObject

Machine Translated by Google

Classe de Sistema de Física

Para usar nossa quadtree como uma estrutura de fase ampla, precisamos encontrar alguns métodos atualmente vazios dentro da classe PhysicsSystem, BroadPhase e NarrowPhase, e crie um novo método UpdateOb-jectAABBs. A classe foi definida para usá-los em vez de BasicCollisionDetection se a variável useBroad-Phase tiver sido definida como true no construtor, então você pode querer definir isso agora ou adicionar um
tecla para alterná-lo à vontade.

Começaremos com o método BroadPhase. Nele, faremos duas coisas: primeiro, criar um
QuadTree de todos os objetos da cena e, em segundo lugar, atravesse a árvore para chegar a cada folha
nó e adicione quaisquer colisões exclusivas a uma lista de colisões globais para processar posteriormente na fase estreita. Lembre-se, os objetos podem acabar em vários nós de árvore se o seu volume delimitador cruzar vários nós.
volumes de nós quadtree, então há uma chance de que o mesmo par de objetos esteja em vários nós, então
temos que detectar isso para garantir que os objetos não tenham resolução de colisão calculada para eles
várias vezes.

Para iniciar a função BroadPhase, vamos simplesmente construir um Quadtree com alguns parâmetros padrão e, em seguida, iterar por todos os objetos no mundo do jogo, inserindo-os em
a árvore um após o outro:

```
1  void PhysicsSystem :: BroadPhase () {
2          broadphaseColisões . claro ();
3  QuadTree < GameObject * > árvore ( Vector2 (1024                    ,    1024) ,7 , 6);
4
5          std :: vector < GameObject * > :: const_iterator primeiro ;
6          std :: vector < GameObject * > :: const_iterator last ;
7          mundo dos jogos . GetObjectIterators (primeiro último);          ,
8          for ( auto i = primeiro; i! = último; ++ i) {
9                  Vector3 halfSizes ;
10                 if (!(* i ) -> GetBroadphaseAABB ( halfSizes )) {
11                         continuar ;
12                 }
13                 Vetor3 pos = (* i ) -> GetConstTransform (). GetWorldPosition();
14                 árvore . Inserir (* i halfSizes, pos ,
15         }
```

Método PhysicsSystem::BroadPhase

Para armazenar os pares de colisão para processamento posterior, usaremos um std::set broadphaseColli-sions, armazenado como uma variável de membro do PhysicsSystem. Um conjunto conterá apenas valores únicos, e
não adicionará nenhuma duplicata, então podemos continuar chamando insert com segurança mesmo em pares de objetos que já podem estar no contêiner (lembre-se, nossos corpos físicos podem acabar cobrindo vários nós de quad tree, mas
queremos verificar suas colisões reais apenas uma vez).

Com a árvore concluída, podemos determinar quais conjuntos de objetos podem estar colidindo. Para fazer isso,
vamos usar o método OperateOnContents do QuadTree, que se você se lembra, irá
percorra a árvore para encontrar todos os nós folha e, em seguida, chame uma função que recebe a lista de nós do nó dados como seu parâmetro. Vamos definir a função a ser passada para o método ali mesmo no
lista de parâmetros, usando uma função lambda:

```
16         árvore . OperateOnContents ([&]( std :: list < QuadTreeEntry < GameObject * > >& dados ) {
17                 CollisionDetection :: Informações de CollisionInfo ;
18
19                 for ( auto i = dados. início (); i! = dados. fim (); ++ i) {
20                         for ( auto j = std :: next ( i ); j != dados . end (); ++ j ) {
21                                 // este par de itens já está no conjunto de colisão -
22                                 // se o mesmo par estiver em outro nó quadtree junto etc.
23                                 informações. a = min ((* i). objeto, (* j). objeto);
24                                 informações. b = max ((* i). objeto, (* j). objeto);
```

Machine Translated by Google

```
25                          broadphaseColisões . inserir (informações);
26                  }
27              }
28          });
29}
```

Método PhysicsSystem::BroadPhase

Como o lambda irá inserir pares de colisão potenciais na variável broadphaseCollisions do PhysicsSystem que fizemos anteriormente, devemos capturá-lo, permitindo-nos utilizá-lo sem ter que enviá-lo explicitamente como um parâmetro e tornar nosso código QuadTree mais complexo. Para automaticamente capturar todas as variáveis por referência, precisamos simplesmente adicionar um & à lista de captura do lambda (o colchetes na frente da definição da função). Dentro da própria função, teremos um loop for aninhado para iterar por todos os objetos dentro do nó da árvore quádrupla, semelhante a como fizemos no método BasicCollisionDetection anteriormente. Desta vez, porém, não estamos tentando executar diretamente detecção de colisão, mas em vez disso adicionar as colisões potenciais ao conjunto (linha 26). Para tornar a vida um pouco mais fácil, vamos definir uma ordem estrita de objetos no par de colisão, para que possamos evitar adicionando uma colisão entre objetos A e B, e também B e A, como colisões separadas. Para fazer isso, vamos simplesmente sempre fazer com que o ponteiro do objeto com o valor numérico mais baixo seja o objeto 'a' em o par de colisão e 'b' o mais alto. O std::set sendo usado para o contêiner de colisão de fase ampla depende do operador menor que definido para nossa estrutura CollisionInfo. Aqui está o código para o operador, definido no arquivo CollisionDetection.h:

```
30 operador bool <( const CollisionInfo e outro) const {
31          tamanho_t outroHash = (tamanho_t) outro. a + ((tamanho_t) outro. b << 8);
32          tamanho_t thisHash = (tamanho_t) a + ((tamanho_t) b << 8);
33
34          return (esteHash <outroHash);
35}
```

Método PhysicsSystem::BroadPhase

Para descobrir se um par de colisão é 'menor que' outro, o código executa um simples 'hashing' função, para transformar os ponteiros armazenados dentro de cada um (a e b são os ponteiros para GameObjects preenchemos a chamada da função OperateOnContents acima) em um único valor inteiro que pode ser comparado contra. Idealmente, tal função hash sempre formaria um número único para cada par de objetos, então você pode querer estender a classe GameObject para ter um ID inteiro exclusivo para cada objeto no mundo, pois seria mais confiável trabalhar com ele do que com um ponteiro.

Isso é tudo que precisamos para nosso exemplo de fase ampla, agora para a fase estreita. Esta função é muito mais curto - simplesmente iteramos por todas as colisões adicionadas à lista allCollisions, e se os dois volumes de colisão estão realmente se cruzando (isso verifica o volume 'verdadeiro' de cada forma, não o AABB que usamos para preencher a estrutura de fase ampla), chame o método ImpulseResolveCollision que feito anteriormente na série de tutoriais para aplicar as forças corretas para separá-los.

```
1 void PhysicsSystem :: NarrowPhase () {
2          for (std :: set < CollisionDetection :: CollisionInfo >:: iterador
3                  i = fase amplaColisões. começar ();
4                  i != broadphaseCollisions . fim (); ++eu) {
5                  CollisionDetection :: CollisionInfo info = * i ;
6                  if (CollisionDetection :: ObjectIntersection ( info .a info .framesLeft =                    , informações .b , informações)){
7                          numCollisionFrames ;
8                          ImpulseResolveCollision (* info .a info . point );          ,    * informações .b ,
9                          todasColisões. inserir (informações); // insere em nosso conjunto principal
10                 }
11         }
12}
```

Método PhysicsSystem::NarrowPhase

Machine Translated by Google

Por último, precisamos criar um novo método UpdateObjectAABBs, preenchê-lo com o seguinte código e, em seguida, chamá-lo a cada frame do jogo do método PhysicsSystem Update:

```
1 void PhysicsSystem :: UpdateObjectAABBs () { 2
        std :: vector < GameObject * > :: const_iterator primeiro ; std :: vector <
3       GameObject * > :: const_iterator last ; mundo dos jogos . GetObjectIterators
4       (primeiro último); for ( auto i = primeiro; i! = último; ++ i) {  ,
5
6               (*i) -> UpdateBroadphaseAABB();
7       }
8}
```

Método PhysicsSystem::UpdateObjectAABBs

Isso permitirá que o PhysicsSystem veja as últimas caixas delimitadoras corretas para os objetos do jogo em cada quadro - se você nunca alterar os tamanhos, poderá apenas chamar UpdateBroadphaseAABB uma vez quando um objeto for construído.

Conclusão

Com essas mudanças, implementamos uma fase ampla básica e uma fase estreita em nosso mecanismo de física. A fase ampla é projetada para reduzir globalmente o número de cálculos de detecção de colisão executados em cada quadro. Em vez de testar cada objeto em relação a todos os outros objetos, podemos, com o uso cuidadoso de uma estrutura de particionamento espacial, verificar apenas objetos próximos uns dos outros e, em vez de realizar 2 grandes n testes.[2] Uma vez que isso nos dá um conjunto reduzido de casos de teste, a fase estreita pode então aplicar as forças corretas para resolver nossas colisões e atualizar qualquer um de nossos retornos de chamada de função.

Trabalho adicional

1) As estruturas de aceleração espacial são úteis para mais do que apenas a fase ampla do seu mecanismo de física. Eles também podem ser usados para acelerar seu raycasting - se um raio não cruzar com a caixa delimitadora que representa um nó octree, então é lógico que ele não pode cruzar nenhum dos objetos dentro dele, potencialmente salvando muitos testes de interseção de raios . Tente modificar os métodos Raycast introduzidos no primeiro tutorial para construir uma lista de objetos a serem testados com base em se o raio intercepta os nós de sua quadtree ou octree.

2) As estruturas de aceleração espacial também são úteis fora do âmbito da física. Aprendemos anteriormente que podemos acelerar a renderização da cena realizando a seleção de tronco para construir uma lista de nós de cena que estão dentro do ponto de vista da câmera do jogo. Considere como você usaria testes de interseção de planos nos volumes de colisão de seus nós quadtree ou octree para obter uma lista de objetos de jogo para renderizar.

3) O exemplo quadtree descrito aqui é funcional, mas ingênuo – há muito espaço para melhorias de desempenho! Considere armazenar um std::vector de nós quadtree descartados entre quadros e usá-los em vez de chamar new e excluir cada quadro.

4) Talvez ter uma única estrutura de aceleração construída em cada quadro não seja a melhor maneira de acelerar os cálculos físicos - será repetir o mesmo trabalho em cada quadro para a inclusão de todos os seus objetos estáticos (ou seja, aqueles com massa infinita que nunca se moverá, como paredes e chão).
Considere ter duas estruturas, uma para objetos estáticos e outra para objetos dinâmicos. Talvez seja necessário criar um método na classe QuadTree que tente determinar com quais nós um objeto dinâmico cruza, mas em vez de inseri-lo na árvore, apenas então chama uma função no conteúdo do nó - permitindo que você execute código de interseção apenas com as formas na área do objeto.