Lucas Grassano Lattari

Gerenciamento e Visualização de Ambientes Virtuais

Orientador:
Marcelo Bernardes Vieira

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Juiz de Fora

Monografia submetida ao corpo docente do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como parte integrante dos requisitos necessários para obtenção do grau de bacharel em Ciência da Computação

Prof. Marcelo Bernardes Vieira, D. Sc.
Orientador

Prof. Raul Fonseca Neto, D. Sc.

Prof. Rodrigo Weber dos Santos, D. Sc.

Sumário

Lista de Figuras

Resumo

Este trabalho fundamenta-se na representação de ambientes virtuais, mostrando-os de maneira realista, através da simulação e visibilidade. São apresentados os fundamentos necessários para a compreensão da problemática, a análise de soluções eficientes computacionais e as implementações práticas desses conceitos de maneira independente, para os cenários e os personagens.

Finalizando, nesta simulação gera-se um ambiente virtual habitado por agentes, que devem interagir com o cenário.

1 Introdução

Gráficos proporcionam a forma mais natural de comunicação entre um usuário e um computador. A Computação Gráfica (CG) tornou-se o mecanismo mais importante para a geração e edição de imagens desde a fotografia e a televisão (FOLEY et al., 1993). A CG permite modificar elementos de uma imagem que anos atrás não eram possíveis, via sintetização de superfícies, colorização, efeitos de luz e sombreamento, correção de imperfeições etc. Esse conceito é facilmente estendido para imagens dinâmicas, que se modificam ao longo de um tempo ou espaço.

Essa ciência continuou a se desenvolver, ao longo do tempo, surgindo depois dela a Computação Gráfica Interativa (CGI). Essa última tem como foco principal permitir a interação de usuários com ambientes gráficos dinâmicos. Ela teria surgido em treinamentos militares, como através dos simuladores de vôo, tornando esses exercícios mais baratos, flexíveis e sem riscos de segurança. Atualmente, áreas de jogos e entretenimento digital são as maiores impulsionadoras da CG, que também definem novos panoramas e tendências. Cenários desenvolvidos para CGI são definidos como ambiente ou realidade virtual.

Ambientes virtuais tornam-se cada vez mais interessantes para desenvolvimento de interfaces realistas, com focos na área de educação, simulação, entretenimento etc. Com isso, pesquisas tem sugerido a integração de áreas como a Inteligência Artificial (IA), a Vida Artificial (VA), e a Animação Comportamental em trabalhos desse tipo. O objetivo é aprofundar a experiência do usuário com entidades inteligentes e meios efetivos de representação de ambientes.

1.1 Definição do Problema

O problema tratado nesta monografia é o da representação, simulação e visualização 3D em tempo real de um ambiente virtual interno e de personagens na forma de modelos animados. Para isso, deve-se considerar uma série de sub-problemas.

Um deles é o processo de otimização de cenas. A dificuldade em criar cenários realistas vem da exigência de processamento ao se gerar um número suficiente de cenas por segundo, para criar um efeito de continuidade. Cada imagem gerada é definida como um *quadro*. Para se perceber continuidade em tempo real são necessárias pelo menos 30 quadros por segundo. Assim, é necessário manter um equilíbrio entre o realismo de uma cena e a capacidade de criar quadros por segundo.

Um dos meios para se otimizar ambientes virtuais é removendo superfícies escondidas que não forem visíveis ao observador. Para isso, são feitos testes de interseção entre objetos gráficos de uma cena com uma câmera de visualização. Ainda assim, é pouco eficiente fazermos o teste com todos os objetos do cenário, sendo mais adequado agrupá-los hierarquicamente a partir da sua posição espacial. Para isso, utilizamos uma estrutura de dados do tipo árvore (EBERLY, 2000). Esse método é definido como *particionamento de espaços*.

O particionamento de espaços também se aplica para a determinação de superfícies visíveis. Ao projetar um ambiente 3D em um plano bidimensional, é preciso organizar a partir da câmera, polígonos de acordo com sua proximidade. Isso é necessário para desenhar objetos de forma correta à frente ou atrás de outros.

Outro problema comum em realidade virtual é a representação de modelos 3D, na qual a animação, via computador, é uma de suas principais instâncias. A animação tradicional é feita a partir de uma ilusão de movimento, ao filmar sucessivamente diversos quadros estáticos em um filme. No caso computacional, seria o mesmo efeito anterior aplicado sob uma cena dinâmica, interpolando movimentos quadro a quadro (THALMANN; THALMANN, 1996).

No caso de animação, é preciso determinar em tempo real quais são as posições e orientações de cada parte do corpo, levando-se em conta cada cena, a fim de criar um efeito natural de continuidade. Para isso, é adicionada uma malha de linhas deformáveis, interna ao personagem, semelhante a um esqueleto. Cada uma dessas linhas são denominadas ossos. Cada esqueleto é composto por um conjunto de vértices interligando ossos. Assim, cada animação é composta pela interpolação de novas posições entre vértices, respeitando essa hierarquia.

1.2 Objetivos

O objetivo primário desta monografia é pesquisar e formalizar os métodos de particionamento de espaço e de animação de personagens, e como objetivos secundários, advindos do objetivo primário, temos:

- apresentar matematicamente os conceitos aplicáveis na definição de um ambiente virtual;
- propor uma representação de classes concretas para um modelo computacional de cenários e personagens virtuais;
- estudar o formato de hierarquias utilizadas em softwares existentes;
- aplicar os conceitos e as representações propostas na implementação de um teatro virtual para simulação de agentes;

2 *Fundamentos para representação de cenas*

Para se construir um ambiente virtual, primeiro deve-se definir como serão representados os seus elementos. Esse estudo é, principalmente, geométrico pois considera a simulação fiel da superfície de seus objetos e da relação espacial entre eles. Sendo o mundo real composto por elementos em duas ou três dimensões, este trabalho utilizará os conceitos provindos das áreas de Geometria Sólida e Geometria Plana.

Serão utilizados basicamente duas noções geométricas para se representar cenas virtuais. A primeira delas é o conceito de *espaço euclidiano* tridimensional $\mathbb{R}^3$. Este será utilizado por permitir a composição e manipulação de vários objetos em duas ou três dimensões, de maneira intuitiva. Ele é o que melhor representa o espaço no qual está inserido o mundo real.

A outra noção necessária deve permitir a simulação dos objetos que estarão contidos nesse espaço, tais como: as salas, as paredes, o chão, as cadeiras etc. Este deve ser simples, versátil, e se restringir a representar, com exatidão, esses objetos em duas ou três dimensões. Para isso, será utilizada a noção de *complexo simplicial* para dimensões 1, 2 e 3.

Define-se um p-simplexo δ_T como o fecho convexo de T pontos linearmente independentes, em que $T \subset \mathbb{R}^n$ e p seja a dimensão do simplexo δ_T, de forma que $0 \leq p \leq n$.

Figura 1: Alguns p-simplexos de ordens 1, 2 e 3, respectivamente

A idéia de simplexo é denotar o menor polítopo possível para uma dada dimensão. Como todos os elementos do mundo real podem ser sintetizados por polítopos simples, ou composições dos mesmos em até 3 dimensões, limitaremos este estudo a eles.

De acordo com (MEDEIROS et al., 2007), um complexo simplicial K é uma coleção de p-simplexos δ que satisfazem as seguintes propriedades:

1. Se $\delta_T \in K$, então $\delta_U \in K$, $U \subset T$. Dizemos que δ_U é face de δ_T.
2. Se δ_U, $\delta_V \in K$, então $\delta_{T \cap V} = \delta_U \cap \delta_V$.

A segunda restrição determina que p-simplexos só podem se interceptar via vértices ou arestas comuns. Em uma ou duas dimensões, tem-se interligações entre pontos e fechos de segmentos de reta, respectivamente. Em três dimensões, essa interpretação afirma que todo o espaço simplicial pode ser composto por triangulações (divisão de superfícies ou planos em uma malha de triângulos, de forma que sempre compartilhem uma aresta). A próxima figura exemplifica isso de maneira intuitiva:

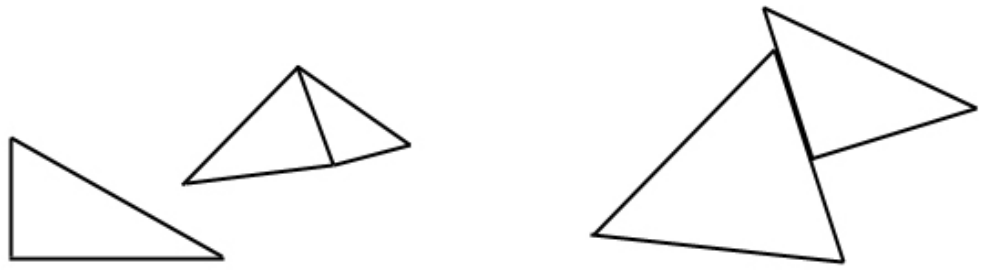

Figura 2: As figuras do lado esquerdo da foto são complexos simpliciais. A da direita não é um complexo simplicial.

Recapitulando, os ambientes virtuais neste trabalho serão representados por espaços euclidianos $\mathbb{R}^3$ e por complexos simpliciais de até terceira ordem. Ao longo deste trabalho, esses elementos serão referidos apenas como: *espaço* e *objetos*, respectivamente.

2.1 Particionamento de Espaços

Ainda que uma cena possa ser completamente representada em um espaço, essa abordagem continua sendo bem geral e limitada. Antes de se definir as soluções, é necessário ter certeza se o espaço será capaz de comportá-las, de forma independente e com o máximo de eficiência. Um dos meios para se fazer isso, é reduzindo a complexidade do espaço em classes menores, utilizando um método de divisão e conquista.

Se subdividir o problema principal em classes, a partir de algum critério, como a distribuição de objetos ao longo de certas áreas, essa estratégia dá maior poder de decisão e controle para o modelo computacional. Com isso, este modelo pode definir coerentemente que regiões necessitam de tratamento e de que forma tais soluções devem ser aplicadas. Esse é exatamente um dos princípios elementares do conceito de *particionamento de espaços*.

Uma partição P de um espaço X é uma coleção de subregiões $p_1, p_2, \cdots, p_n$, $n \in \mathbb{N}$, tais que:

1. seja p_i, $p_j \in P$, e $p_i \neq p_j$. Tem-se então que $p_i \cap p_j = \emptyset$;
2. a união de todos os elementos de P é um conjunto congruente a X. Ou seja, $X = p_1 \cup p_2 \cup \cdots \cup p_n$.

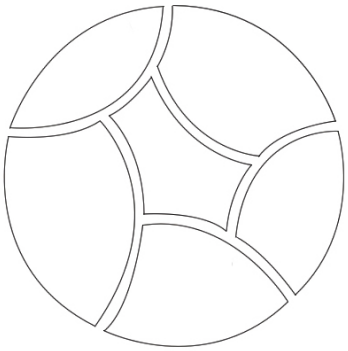

Figura 3: Exemplo de um espaço particionado em 6 subelementos.

No contexto de particionamento de espaços, cada elemento de uma partição será denominado como uma *célula* ou um *setor*. Em geral, as células são caixas convexas para facilitar os cálculos. Os critérios para se definir a geração das mesmas são inerentes ao tipo de esquema de particionamento empregado, e serão apresentados mais adiante, para cada esquema.

Espaços particionados podem ser representados como grafos acíclicos, ou árvores. Essa definição existe formalmente na CG como *grafo de cena*. Cada nodo interno representa uma célula, e seus filhos representam subespaços pertencentes ao nodo pai. As folhas são os menores elementos de todo o conjunto espacial. Essa estrutura hierárquica é a forma mais geral para se representar o particionamento de espaços de um ambiente.

Em geral, as partições são construídas por hiperplanos lineares, que são subespaços vetoriais de *codimensão* $n - 1$, dado um espaço euclidiano $\mathbb{R}^n$. Codimensão é o termo

usado para indicar a diferença entre a dimensão da maioria dos objetos desse espaço com a dimensão do menor objeto possível. Para $\mathbb{R}^3$, este elemento é o plano. Especificando essa análise para $\mathbb{R}^3$, os hiperplanos de particionamento sempre serão referidos neste trabalho como planos.

É importante definir a orientação dos particionamentos, que podem ser de dois tipos: os alinhados aos eixos ou os alinhados aos polígonos. No primeiro caso, os planos de particionamento são sempre paralelos a um dos eixos cartesianos e, no segundo, a posição de um polígono arbitrário de algum objeto da cena, define a passagem de um plano. A Figura 4 mostra essas duas representações, respectivamente.

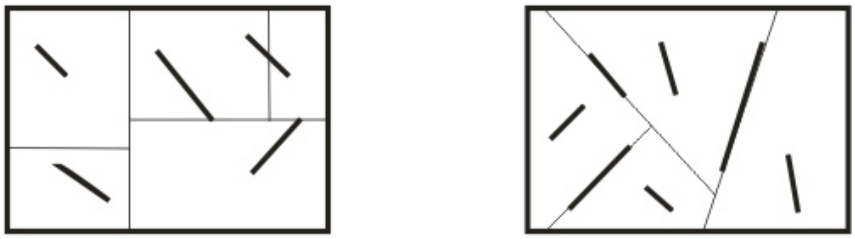

Figura 4: Espaços compostos por objetos, alinhados por eixos ou por polígonos.

A orientação de alinhamento aos eixos é mais simples de se manipular, pois são sempre construídos os setores no formato de caixas alinhadas, sendo assim fácil de se aplicar qualquer operação ou transformação geométrica baseada nos eixos cartesianos. Entretanto, leva pouco em consideração as dimensões e as posições dos objetos, podendo gerar células pouco otimizadas, ou que aproximem os objetos de maneira pouco confiável. É importante se otimizar e gerar setores que armazenem objetos da forma mais compacta possível, pois assim se garante a eficácia de métodos que serão vistos mais adiante, como os testes de visibilidade e de detecção de colisão.

A seguir, serão apresentados os esquemas mais comuns de particionamento de espaços para o problema de representação de cenas.

2.1.1 Octrees

A eficiência de um esquema de particionamento depende, principalmente, do espaço que se deseja representar. É preciso tirar proveito da quantidade de objetos inseridos, e da distribuição dos mesmos ao longo do ambiente.

Uma das estruturas mais utilizadas quando os objetos estão distribuídos uniforme-

mente ao longo do espaço são as *Octrees*. Seu esquema de representação é regular, baseado no alinhamento de eixos e construção de células a partir de octantes.

Para construí-la deve-se representar toda a cena sobre uma caixa envolvente. Para cada célula, sua partição é realizada por três planos distintos entre si, paralelos aos eixos x, y e z. Estes planos obrigatoriamente se interceptam no vértice central que define a célula. Com isso, são gerados octantes, semelhantes aos da Figura 5 (MOLLER; HAINES, 2002).

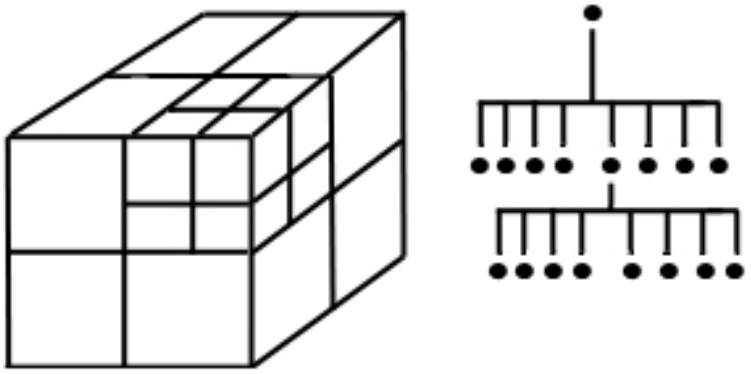

Figura 5: Espaço particionado com Octree, e seu grafo de cena.

Por serem gerados octantes, são criadas oito células para cada nó não-folha. Esse processo é repetido até que todos os setores ou estejam completamente preenchidos por objetos, ou totalmente vazios, ou seja, enquanto parte de uma célula for interceptada por um objeto, esta é subdividida.

Devido a quantidade de filhos gerada por célula, ela é ideal para se representar detalhadamente a modelagem de objetos sólidos ou espaços com objetos distribuídos homogeneamente (como uma cidade composta por dezenas de prédios próximos). Essa é uma das suas principais vantagens, assim como a simplicidade de se construir e gerenciar essas estruturas. Entretanto, essas características acabam sendo restritivas para outros tipos de espaços.

Em ambientes heterogêneos, essa simplicidade acaba sendo um limitador. Células podem ser geradas em demasia, sem necessidade, ocasionando o excesso de informações redundantes. Além disso, como não existem meios de se posicionar os planos de particionamento de acordo com a posição dos objetos, não há formas de se construir uma estrutura mais simples e otimizada. Isso a torna muito pouco adequada para esses exemplos.

A próxima estrutura se baseia nas vantagens da *octree*, que trazem consigo outras características que lhe garantam maior flexibilidade.

2.1.2 Kd-Trees

Ainda que a abordagem utilizada pelas *Octrees* seja eficiente para certos casos, ela não é adequada para espaços com objetos distribuídos heterogeneamente. Para este último, se faz necessário um meio de representação que se adapte melhor ao conjunto de objetos do cenário.

As *Kd-Trees* são uma generalização do conceito de *Octree*, podendo também ser facilmente transportada para qualquer dimensão euclidiana (originando assim seu nome). Apesar disso, este estudo discutirá apenas *Kd-Trees* tridimensionais, ou *3d-Trees*.

Assim como as *Octrees*, as *3d-Trees* também orientam seus planos de particionamento via alinhamento dos eixos, entretanto, se diferem ao longo do processo de subdividisão de regiões. Inicialmente, todo o espaço da cena é envolto por uma caixa, e esta é particionada por um único plano paralelo a um dos eixos cartesianos e que subvidide-a em duas células. Esse processo é repetido sucessivamente. Isso já simplifica em demasia a estrutura, sem perda de informação.

Durante o particionamento, os objetos podem ser totalmente inseridos em uma célula ou interceptarem duas células ao mesmo tempo. Para isso, existem duas abordagens possíveis: ou se define que o mesmo pertença aos dois conjuntos, ou o objeto é subdividido até que possa pertencer a um único conjunto. A última abordagem é considerada a melhor, por representar mais fielmente o espaço.

Outra característica desse esquema é sua liberdade para manipular planos de particionamento. Ao invés de fixá-los como em *Octrees*, pode-se definir arbitrariamente sua posição ao longo de um setor, desde que mantendo-os paralelos a um dos eixos cartesianos. Isso permite a esses elementos serem reposicionados e, assim, definir células de tamanhos variáveis mais convenientes, como na Figura 6.

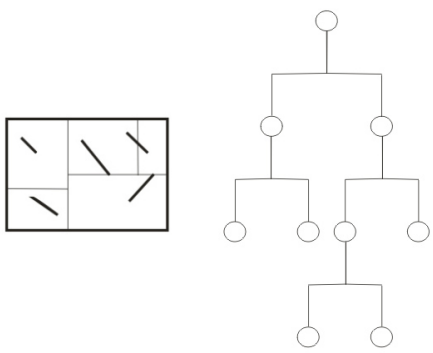

Figura 6: Espaço particionado com Kd-Tree, e seu grafo de cena.

Existem duas estratégias mais comumente empregadas para se definir os planos de particionamento para *3d-Trees*. A primeira delas é simplesmente realizar um ciclo em torno dos eixos, ou seja, realizar a primeira partição em alguma posição paralela ao eixo x, de suas células filhas ao longo do eixo y, das netas em torno de z, repetindo o ciclo. Outra estratégia, considerada mais eficiente, seria particionar a célula ao longo da direção da aresta de maior comprimento da sua caixa envolvente. Definido-a, se verifica qual é a melhor posição para se fazer o particionamento, verificando o total de objetos que cada sub-célula armazenará. Isso é feito para se balancear a *Kd-Tree* gerada.

Essa estrutura é mais adequada para modelos que não exijam tanto armazenamento de detalhes, sem perder sua principal característica, que é a regularidade. Como foi dito, é muito mais simples se elaborar técnicas aplicáveis a células que estejam alinhadas aos eixos, ainda que isso nem sempre seja garantia do melhor particionamento. Além disso, a liberdade de se manipular os planos de recorte lhe permite boas garantias de otimização.

Porém, essa estrutura não dispõe de todo o potencial necessário para se representar cenas. A próxima a ser apresentada possui particularidades extremamente vantajosas para os problemas mencionados até então.

2.1.3 Árvores BSP

Os espaços com seus objetos distribuídos em posições arbitrárias são melhores representados se seu esquema de particionamento for dependente dessas localizações. Para isso, se aplica o conceito de *árvores BSP* (*Binary Space Partitioning*, ou Particionamento Binário de Espaços), que é uma generalização de *Kd-Trees*.

Diferentemente das formas de representação apresentadas, nesse esquema a construção de partições é baseada na orientação por alinhamento de polígonos. Assim, para cada célula da cena, é escolhido um polígono que será posicionado por um plano de particionamento, que subdividirá seu espaço em dois conjuntos. Assim como nas *Kd-Trees*, sempre que um objeto interceptar este plano (ou seja, pertencer a mais de um setor), este poderá ser adicionado aos dois, ou ser dividido até que possa ser inserido a uma célula de forma única. Logo, a última abordagem é considerada melhor. Esse processo é repetido sucessivamente até que todos os objetos estejam representados na estrutura, como mostrado na Figura 7.

Entretanto, se os critérios para a escolha de polígonos não forem bem definidos, gera-se uma estrutura não balanceada, como por exemplo, um plano que subdivida um subespaço

em duas células: uma que esteja vazia e outra que contenha todos os objetos da cena. Uma das soluções para tal problema é definida como *Critério dos Menores Cruzamentos*. Primeiramente, um número de polígonos candidatos é selecionado randomicamente. O plano posicionado a partir desse polígono escolhido deve ser o que menos vezes intercepte outros polígonos. É empiricamente comprovado para uma cena com 1000 polígonos, que apenas 5 deles devem ser testados por célula particionada para se gerar uma árvore balanceada, proporcionalmente (MOLLER; HAINES, 2002).

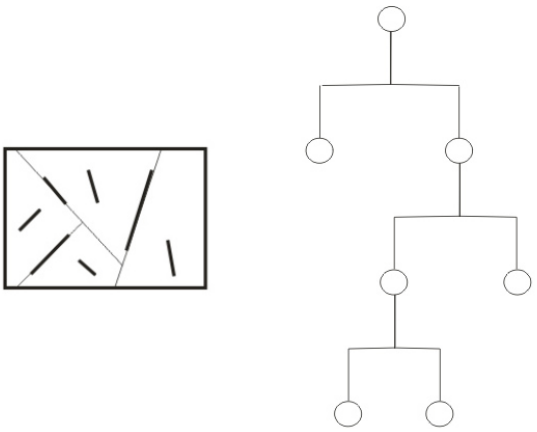

Figura 7: Espaço particionado com Bsp-Tree, e seu grafo de cena.

As árvores BSP são bem flexíveis, no sentido de permitirem a representação de um espaço baseado na posição de seus objetos e amplo controle do funcionamento de seu particionamento. Entretanto, é a estrutura mais difícil de se gerenciar, devido a sua irregularidade geométrica ao longo da construção de células e de processos complexos para sua geração, como a otimização dos dados e os critérios que definem seu encerramento.

Apesar disso, as características das árvores BSP lhes garantem particularidades únicas. As duas principais são os tratamentos de visibilidade e de detecção de colisão. Ainda que todas as estruturas citadas possam tratar esses problemas, as melhores soluções são dadas por árvores BSP. Esses fatos serão esclarecidos nos subtópicos posteriores.

2.1.3.1 Solução para o problema de Visibilidade

As características das árvores BSP são muito convenientes para se solucionar problemas de visualização. Recapitulando, busca-se determinar a visibilidade de uma cena 3D, ou seja, ordenar corretamente seus objetos, considerando a profundidade dos mesmos em projeções bidimensionais e selecionar apenas o que está visível para um observador.

É óbvio perceber que em um ambiente virtual, a posição do observador é alterada

constantemente. Isso exige o reposicionamento de todos os objetos da cena. Entretanto, os objetos dos cenários não se modificam ao longo do tempo. Com isso, é bem conveniente se utilizar uma estrutura pré-processada e estática, que seja utilizada pelo observador para cálculos de visibilidade (SAMET, 1990).

Para se determinar a profundidade e projetá-la, basta fazer um caminhamento sobre sua estrutura de árvore. Como dito, se uma célula não for folha, então ela é particionada em dois conjuntos. Já que esses subconjuntos são construídos a partir de um plano, pode-se verificar rapidamente em qual das duas partições uma posição se encontra, apenas verificando se esta localiza-se à frente ou atrás de um plano, para cada célula.

Sendo fácil de verificar uma posição dentro de uma árvore BSP, basta percorrê-la de trás para frente. Assim, devido aos planos, pode-se garantir quais são as folhas mais afastadas de um determinado observador.

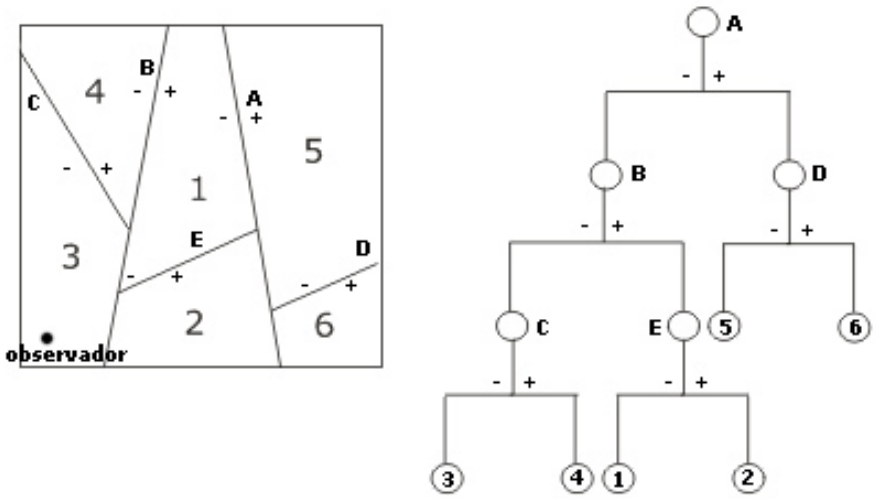

Figura 8: Verificação de visibilidade com árvores BSP.

Através do espaço representado pela Figura 8, com um observador na região 3, deseja-se determinar uma lista ordenada de objetos, do mais afastado ao mais próximo, de acordo com o observador.

Acessando a árvore de cena, começa-se pela raiz A. Como o observador está na direção negativa ao plano de A, então sabemos que a região positiva é a mais distante, que é o nó D. Verificando a posição em relação a D, observa-se que ele está na direção negativa novamente. Com isso, é óbvio que 6 é a região mais afastada. Acessa-se então as regiões 6 e 5, e as selecionam como as mais distantes.

Continuando o percurso, acessa-se B. Novamente o observador está na região negativa ao plano, o que leva ao nó E. Dessa vez, o observador está na direção positiva a E, o que faz percorrer a direção negativa, ou seja, a região 1. Assim sabemos que 1 e 2 devem ser desenhados nessa ordem. Por fim, verifica-se C, em que deve-se desenhar 3 e 4. Assim, a projeção das regiões deve ser feita na ordem: 6, 5, 2, 1, 4, 3.

Outro emprego útil desse esquema de particionamento é na verificação de células visíveis. Obviamente nem todo o espaço estará visível para um observador em um dado momento. A divisão do espaço em setores minimiza o total de verificações para se saber se uma dada região está visível. À medida que o caminhamento é realizado, verifica-se, caso o observador consiga visualizar, uma célula geral. Se o observador não puder, não é preciso nem verificar seus filhos, nem verificar a projeção de profundidade dos mesmos.

Faz parte da arbitrariedade na construção de células em árvores BSP essa verificação de visibilidade, sendo seu maior diferencial na renderização em tempo real, para outras estruturas. Entretanto, vantagens também podem ser aplicadas em detecção de colisão, como será visto no próximo tópico.

2.1.3.2 Solução para o problema de Colisão

Deseja-se implementar um ambiente virtual realista que permita a interação entre um cenário e alguns personagens 3D. A detecção de colisão é importantíssima nesse quesito, pois por meio dela, pode-se planejar ações para um personagem a partir de sua interação com algum elemento, seja para simular realista a reação de um personagem contra uma parede, seja para verificar se ele está mantendo contato com uma porta para abri-la, dentre outras aplicações. No entanto, esse estudo se aterá apenas em determinar colisões.

Tal tratamento é semelhante ao abordado no tópico anterior, pois também é realizado um percurso pela árvore. No entanto, busca-se determinar em qual célula se localiza um objeto que se deseja testar a colisão. Se a mesma ocorrer, ela será restrita a esse setor, sendo necessário apenas fazer a verificação em seu interior.

Iniciando da raiz A, verifica-se a posição do objeto em relação ao plano A. Como ele está na direção negativa, percorre-se a árvore até o nodo B. Em B, ele está na região positiva, o que leva a E. Como ele está na posição negativa de E, ele acessa a região 1.

Entretanto, como essa análise é para um objeto dinâmico, deve-se representar o seu trajeto como um segmento de reta. Percorre-se então a árvore BSP buscando qual a folha onde se localiza esse segmento. Caso o mesmo esteja contido em mais de uma célula,

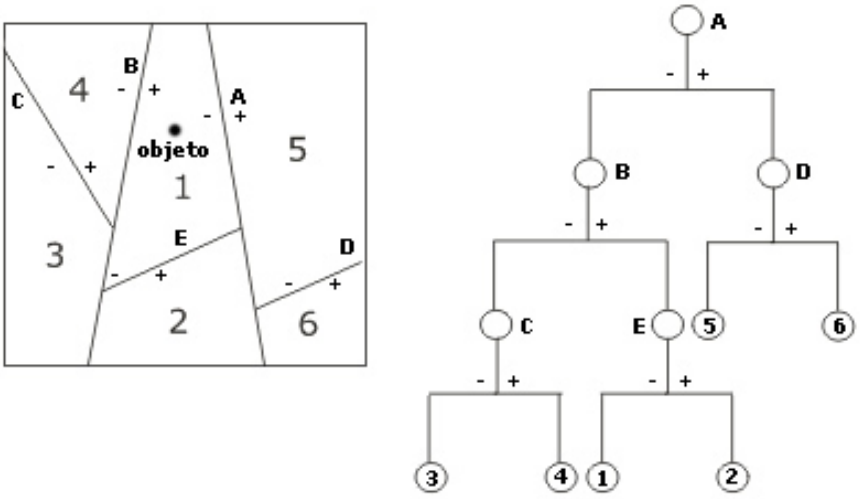

Figura 9: Determinando colisão em Árvores BSP.

esse deve ser subdividido até que se insira unicamente em um setor (BERGEN, 2003). Determinado esse trajeto em uma célula, basta verificar se o objeto intercepta algum elemento geométrico interno a esse setor.

Os testes de interseção costumam ser realizados sobre caixas ou superfícies envolventes aos objetos. Isso é feito para simplificar o total de testes realizados, principalmente sobre objetos muito complexos. Com isso, é mais interessante se construir células que estejam alinhadas aos objetos do que aos eixos, pois a aproximação é feita da melhor forma possível. Essa é outra vantagem do esquema de árvores BSP sobre os outros listados.

2.2 Fundamentos de Animação de Personagens

É preciso formalizar os elementos necessários para se representar e simular a ação de personagens 3D. Sua representação geométrica também é composta por complexos simpliciais em $\mathbb{R}^3$. Um exemplo está na Figura 10, sendo que esse e os outros modelos pertencem ao jogo *Half-Life*, propriedade da Valve Corporation.

Para se obter realismo na representação de um personagem virtual, é preciso animá-lo, ou seja, demonstrar sua movimentação e a variação de suas ações ao longo do tempo, de maneira natural. Diferentemente de modelos rígidos, a animação de personagens é mais complexa, pois deve levar em conta a ação de cada articulação de maneira independente, e de forma que influencie nos outros membros de maneira hierárquica. Um exemplo é a movimentação de uma mão, que influencia em um braço, e por sua vez em um antebraço

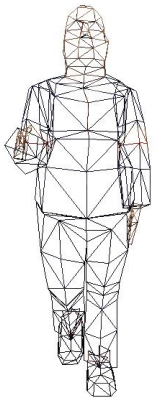

Figura 10: Exemplo de personagem representado por complexos simpliciais.

etc.

A ação de um personagem ao longo do tempo é realizada via transformações geométricas sobre sua geometria. Nesse escopo, são fundamentais as seguintes operações: a translação e a rotação. Aplicando-as em cada cena, cria-se a ilusão de movimento. No entanto, essa abordagem por si só é falha, pois se o animador não tiver cuidado sobre as operações aplicadas, algumas partes do modelo podem ser transladadas ou orientadas de maneira impossível ou nem um pouco realista em intervalos de tempo. É preciso delimitar os limites possíveis do corpo.

Para fazer isso é bem comum se representar as articulações de um personagem utilizando uma malha deformável interna a sua superfície, a qual interfere diretamente sobre cada ação. Essa malha é definida como *esqueleto*. Um esqueleto é uma coleção de ossos, os quais se movimentam de maneira independente, desde que respeitando seus limites geométricos locais.

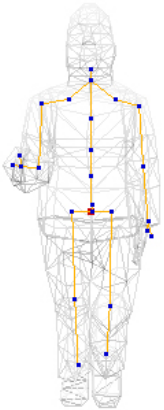

Figura 11: Esqueleto interno a malha de um personagem.

Esse limite geométrico é definido pelas juntas, que são os vértices localizados nas

intermediações entre cada osso, conectando-os dos mais específicos aos mais gerais. Essa hierarquia se baseia na idéia de que, caso uma articulação A sofra um movimento, toda articulação B conectada a ela também seja reposicionada, mantendo a malha coesa. A regra fundamental dessa representação é que as juntas sempre se mantenham conectadas aos mesmos ossos.

As vantagens desse esquema são diversas. Não apenas pelo fato de ser mais realista e considerar a ação de todos os membros do corpo, mas também por permitir criar novas animações ou modificá-las em sua execução. Por exemplo, ainda que um personagem possa estar correndo, ele ainda pode girar sua cabeça em outra direção, ainda que isso não ocorra da maneira padrão.

A seguir será apresentado um esquema tradicional de animação, que exemplifica o processo de gerenciamento de poses ao longo do tempo.

2.2.1 Quadros-Chave

Um dos principais problemas na representação de uma animação, é elucidar uma forma de organizar e movimentar a geometria que compõe um personagem. Para criar esse efeito, são realizadas algumas transformações geométricas variando ao longo do tempo. Assim, é preciso escolher um esquema que armazene a menor quantidade de informação possível, suficiente para realizar todas essas operações geométricas, sem deixar de exibir um aspecto flexivel e facilmente adaptável.

Um dos meios mais convencionais para se representar uma animação é utilizar *quadros-chave* para um personagem. Um quadro-chave representa a descrição de uma pose intermediária do modelo em um instante de tempo, de maneira que a composição de todos esses quadros represente a animação completa. Essa descrição determina quais são as posições dos vértices de um personagem em cada intervalo.

Todavia, utilizar quadros-chave leva a uma série de descontinuidades na animação, dando a mesma um aspecto robótico e com uma série de movimentações bruscas. Uma solução para isso, pouco eficiente, seria utilizar um número grande de quadros-chave. É mais interessante utilizar a menor quantidade possível de quadros-chave e interpolar tanto a posição quanto a orientação de todas as poses intermediárias entre um quadro-chave e outro, em sua execução.

Ainda que essa solução possa suprir diversas necessidades, ela continua trazendo possíveis problemas. Esse esquema, convencionalmente, não utiliza o conceito de esque-

leto, mas apenas aplica transformações geométricas locais sobre cada parte da geometria. Como cada transformação é feita de maneira independente, não se pode garantir que sua superfície estará correta, já que o reposicionamento de vértices é arbitrário, mas o comprimento dos membros é constante. Isso pode levar a uma série de erros e artefatos (EBERLY, 2000).

Uma forma de solucionar esse problema, é utilizar uma abordagem híbrida, selecionando as vantagens do esquema de esqueleto, em conjunto com os quadros-chave. Isso é feito no tópico a seguir.

2.2.2 Esqueletos-Chave

Uma das principais vantagens de se utilizar quadros-chave, é minimizar o total de informações necessárias para se compôr uma animação. Entretanto, para esse esquema funcionar corretamente, é necessário impôr a ele certas restrições geométricas. Isso é feito utilizando-se um esqueleto.

Basicamente, esse esquema funciona da mesma maneira que o exemplificado no tópico anterior. Entretanto, sua diferença é que, ao invés de calcular a interpolação entre um quadro-chave e outro diretamente nos vértices de sua superfície, esse processo é realizado unicamente sobre as articulações. Isso implica em muitas mudanças no seu funcionamento, ainda que a idéia básica seja a mesma.

Como as transformações geométricas são aplicadas sobre as articulações ao invés de sua superfície, não se realizam translações independentes, a fim de solucionar os erros e artefatos anteriores. Ou seja, ao manipular um esqueleto, só são realizadas interpolações angulares em torno de suas juntas. Isso simplifica em demasia o processo, que se limita a um único tipo de interpolação.

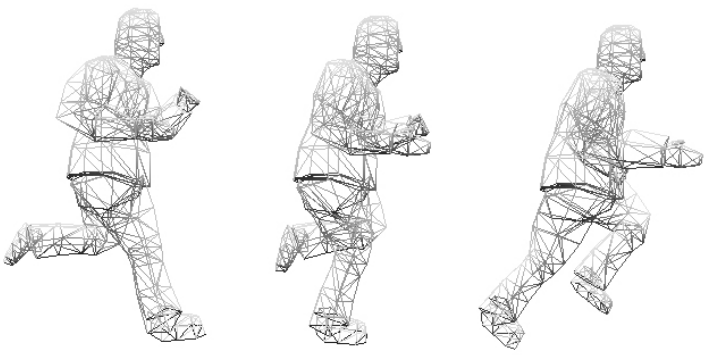

Figura 12: Animação via Quadros-Chave.

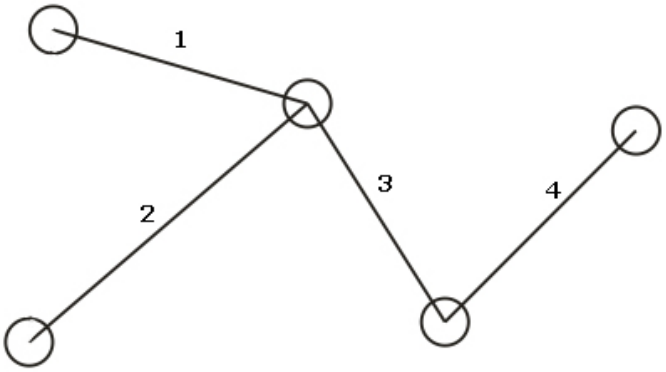

Figura 13: Representação de um esqueleto hierárquico.

Esse processo é bem mais eficiente, pois além de simplificar a movimentação para um único tipo de interpolação, esta leva em consideração os limites físicos do modelo e a ação de uma articulação sobre a outra. Ainda assim, é preciso definir como a movimentação do modelo é realizada.

As posições e orientações de um modelo podem ser animadas via uma interpolação linear (PIPHO, 2002). Entretanto, como o objetivo desse tópico é determinar uma função que possa representar fielmente uma movimentação em torno de um ponto de articulação fixo, o trajeto mais fiel que pode ser utilizado é via uma *interpolação linear esférica*. Uma interpolação linear esférica é uma função $\mathbb{R}^n$ que calcula o menor caminho entre dois pontos P_0 e P_1, tais que pertençam ao comprimento de uma esfera. Ela é usada por que ela é o melhor meio de descrever o trajeto da ação de uma articulação sobre outra, como visto na Figura 13.

Em CG, a maioria das operações geométricas entre objetos, como a translação e a rotação, é realizada via multiplicação de vértices sobre matrizes de transformação. Para realizar essa interpolação é preciso determinar uma matriz de rotação que realize esse movimento para cada articulação envolvida. Calcular essa matriz diretamente é um processo complexo, devido ao número de constantes empregadas, pouca robustez e erros numéricos que possam ocorrer.

Por causa disso, é uma abordagem bem comum, representar uma matriz ou vetores que determinam uma rotação, via uma estrutura denominada *Quaternion* como será visto a seguir.

2.2.2.1 Quaternions

Uma alternativa bem comum em simulações, é utilizar *Quaternions* para fazer cálculos complicados, ao invés de matrizes de rotação. Um *Quaternion* é um par ordenado de um escalar e um vetor 3D. É representado em quatro dimensões, não comutativo, associativo e é um exemplo da classe mais geral de números hipercomplexos.

Tem-se um quaternion como:

$$H = a + bi + cj + dk. \tag{2.1}$$

De tal maneira que a, b, c e $d \in \mathbb{R}$ e $i^2 = j^2 = k^2 = ijk = -1$, em que esses últimos são números complexos.

Os *Quaternions* são utilizados para se calcular operações complexas de forma mais simples, eficiente e robusta. É bem simples fazer a conversão entre eles e elementos em três dimensões, como matrizes e vetores, pois são realizadas um número menor de contas e o total de informação a ser armazenada é menor. Ao invés de multiplicar matrizes, basta realizar a soma de *Quaternions* para se obter o resultado de uma rotação. Da mesma forma, é bem melhor representar uma rotação via 4 componentes do que 16, caso de uma matriz 4x4.

A seguir será apresentado como essas estruturas são utilizadas no cálculo de uma interpolação linear esférica.

2.2.2.2 Interpolação Linear Esférica

Como foi dito, essa interpolação é utilizada no cálculo de articulações, pois o trajeto o qual percorre é semelhante ao arco de uma esfera. Deseja-se obter uma matriz que represente essa interpolação, ao longo de uma constante $\delta \in [0, 1]$. Para isso, é necessário que sejam definidos P_0 e P_1 os vértices que definem as juntas interpoladas, buscando-se interpolar P_1 a partir de P_0.

A partir desses pontos, define-se dois Quaternions q_m e q_i, que representam respectivamente: uma matriz M que aplicada sobre P_1 realiza uma rotação até a posição final P_1', e outro *Quaternion* que representa a posição P_0. Definida-as, aplica-se a seguinte função (FERGUSON, 2001):

$$q_k = \frac{sen(1-\delta)\Theta}{sen(\Theta)} q_0 + \frac{sen(\delta\Theta)}{sen(\Theta)} q_1. \tag{2.2}$$

Assim, para determinar o trajeto de uma articulação conectada a outra, basta transformar q_k em uma matriz K que irá representar a próxima matriz a ser aplicada em um dado frame. Realizando esse procedimento sucessivamente ao longo da constante δ, é feita a interpolação.

3 Representação Computacional

No capítulo anterior, foram apresentados os conceitos básicos que definem os ambientes virtuais, de forma independente para cenários e personagens animados. A partir dessa compreensão, serão propostos meios para se representar e processar essas informações em computadores, via apresentação de classes concretas.

3.1 Classe para Cenários Virtuais

Como dito, um dos objetivos primários desse trabalho é formalizar e demonstrar os métodos referentes a simulação e visibilidade de cenários estáticos internos. Esses processos serão representados e solucionados nessa classe, utilizando o conceito de Particionamento de Espaços.

Como esse cenário virtual é estático e dispõe de objetos geométricos dispostos heterogeneamente em sua superfície, ele é melhor representado por Árvores BSP. As soluções relacionadas com a determinação de visibilidade e a detecção de colisão são melhor determinadas por esse tipo de estrutura.

Uma árvore BSP é uma estrutura binária que subdivide o espaço em conjuntos menores (RANTA-ESKOLA, 2001). Essas partições são armazenadas hierarquicamente em nodos de uma estrutura de dados formando um grafo acíclico ou árvore. Esta foi criada com o principal intuito de classificar polígonos em três dimensões projetados sobre uma superfície bidimensional, como a tela de um monitor.

Anteriormente para se calcular essa projeção, utilizava-se buffers que mapeavam via matrizes, a profundidade de todos os pixels da tela. Para se renderizar um novo polígono, bastava consultar essa matriz e verificar se ele poderia ou não ser desenhado em uma posição. Esse método é denominado *Z-Buffer*. Obviamente, essa solução era muito custosa e difícil de se calcular computacionalmente.

O passo inicial para a construção de uma solução mais eficiente foi a criação do

Algoritmo do Pintor. Este se baseia num procedimento semelhante ao de um pintor elaborando um quadro, que pinta os elementos dos mais afastados da cena aos mais próximos, como por exemplo: o céu, as montanhas, as nuvens, uma casa etc, nessa ordem. Entretanto, essa solução ainda não é adequada, pois além de necessitar ser calculada em tempo real, ainda renderiza muitos objetos que potencialmente seriam encobertos.

Outro problema relevante é a incapacidade de tratar casos de sobreposição em ciclo, como na Figura 14. Isso ocorre porque não se pode definir se o objeto está atrás ou à frente do próximo, condição vital para execução do algoritmo.

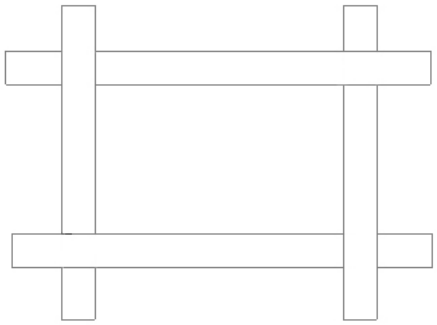

Figura 14: Exemplo de ciclo de sobreposição.

As árvores BSP são uma extensão do Algoritmo do Pintor, pois realizam procedimento semelhante e solucionam essas deficiências, além de ser uma solução mais simples e eficiente, pois a classificação de objetos só exige um único caminhamento em árvore. Além disso, permite a aplicação de esquemas úteis para otimização, a fim de se reduzir o número de objetos não visíveis que serão renderizados. Por fim, como o método de Particionamento de Espaços só pode ser aplicado em elementos convexos, isso impossibilita o surgimento de ciclos de sobreposição.

A fim de se ilustrar o funcionamento do processo de visibilidade e simulação de um cenário virtual, que utiliza árvores BSP, será apresentada um diagrama de classes concretas contendo o mínimo necessário para sua implementação, na Figura 15.

Toda a árvore BSP é constituída de nodos intermediários e folhas. Os nodos representados por *nodes* no diagrama, são setores representados por registros que contêm pelo menos os seguintes atributos:

- o hiperplano em duas dimensões que realiza a subdivisão de células;
- os dois identificadores que definem seus filhos;
- os vértices mínimo e máximo da célula, que definem a região da célula;

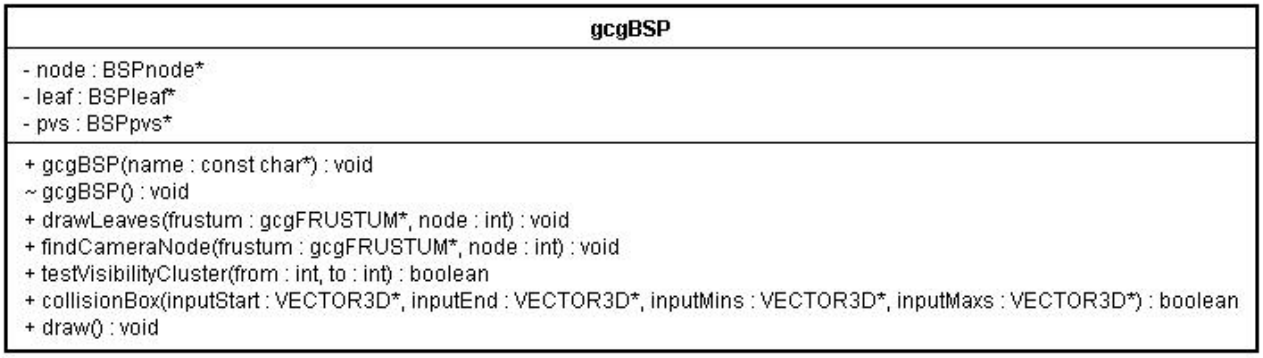

Figura 15: Diagrama de classes que representa a visibilidade de um cenário.

Durante um acesso a uma árvore, são percorridos os nodos e seus filhos sucessivamente, até a localização de uma folha. Ao localizá-la, deve-se renderizar os objetos que estiverem contidos em seu interior. A estrutura folha possui, de forma geral, apenas os campos referentes a sua área geométrica e as informações necessárias para a renderização de objetos, como seus vértices, as faces, as texturas etc.

No construtor da classe, recebe-se o nome do arquivo físico no qual são lidos os dados pré-processados que compõe uma árvore já construída. A partir dessa fundamentação, inicia-se o processo de determinação de superfícies visíveis.

Para determinar as superfícies visíveis, é preciso chamar a função *drawLeaves()* a cada instante o qual o cenário for renderizado. Esse método realiza o exato procedimento da solução de visibilidade apresentada no capítulo anterior, mas aplicando simultaneamente testes para a remoção de objetos que não forem visíveis sobre os nodos verificados. Para cada nodo acessado, são aplicados todos os testes de visibilidade e, se este for visível, continua-se o percorrimento a partir dos seus filhos.

3.1.1 Remoção de Superfícies Escondidas

Anteriormente foi dito que testes computacionais são realizados para se determinar superfícies visíveis. A fim de se minimizar o custo computacional referente a essas verificações, o espaço da cena é particionado em várias células, simplificando o procedimento a um número pequeno de cálculos. Se um setor considerável da cena não for visível a partir de um desses testes, suas células filha não precisarão ser verificadas, o que reduz em muito a complexidade computacional.

O primeiro teste aplicado envolve a visibilidade de um objeto em relação a câmera. Em CG, define-se o *frustum* como o volume de visualização que envolve todo o conteúdo

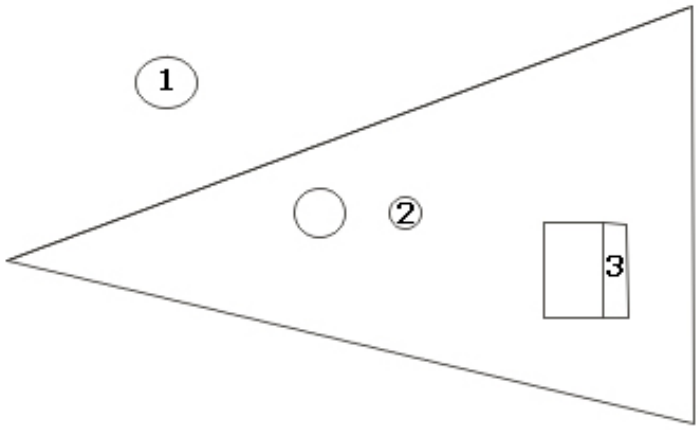

Figura 16: Exemplos de otimizações realizadas em árvores BSP.

visível para um observador. Este é representado via um volume piramidal, composto por 6 planos que o circundam, como na Figura 17.

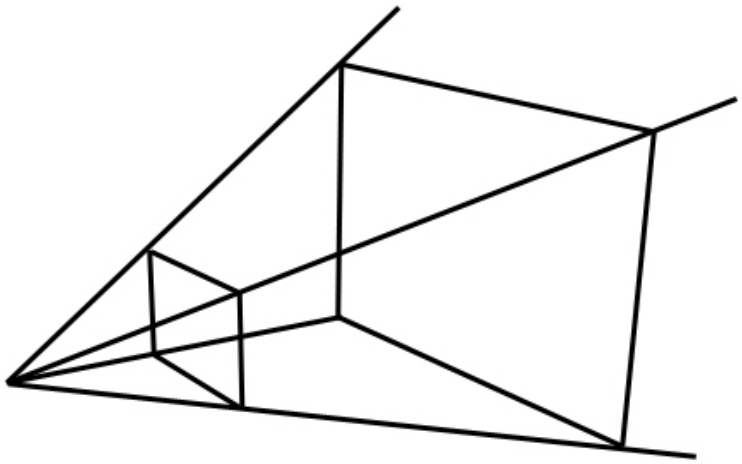

Figura 17: Volume de visualização.

Para que um objeto seja visível a um observador, este ou parte dele deve simultaneamente ser interno a todos os planos mostrados, como se verifica no objeto 1, na Figura 16.

Esse primeiro teste de visibilidade resume-se a verificar se um objeto intercepta o volume da câmera de um observador. No entanto, não é eficiente testar se cada polígono de uma cena pertence ao *frustum*, até porque algumas cenas podem possuir mais de 10.000 polígonos (ABRASH, 1997). Para se fazer isso mais eficientemente, é aplicada a solução de árvores BSP.

Como dito, todo o espaço da cena é particionado em células hierárquicas recursivas.

Ao invés de se realizar o teste de geometria sobre os objetos, isso é feito sobre as células, normalmente representadas sob uma caixa envolvente. Se esta não interceptar a câmera, um número significativo de objetos ou regiões espaciais já não precisa ser verificado para se determinar a visibilidade.

O segundo teste mais empregado é a remoção de polígonos que, embora estejam visíveis a câmera de um observador, não estão com a face posicionada para o mesmo. Está representado pelo objeto 3 da Figura 16. Isso é comum, por exemplo, com uma parede em que o ponto de visão só lhe permite enxergar um dos lados da mesma, não sendo possível visualizar o que está atrás. Para se realizar esse teste, é feita uma análise vetorial.

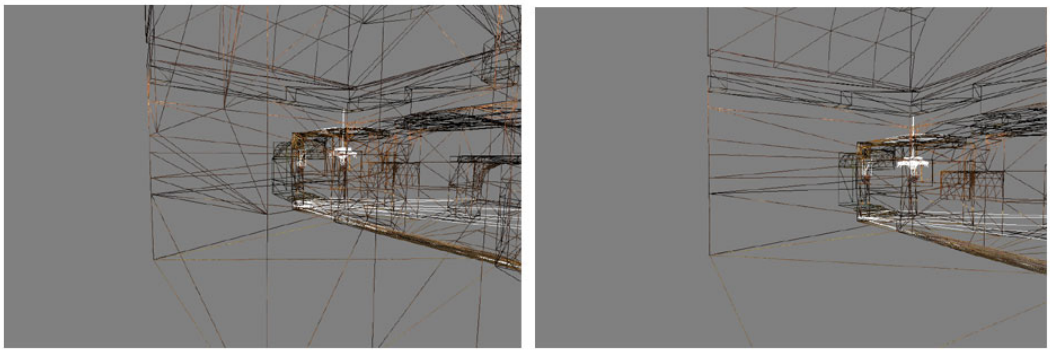

Figura 18: Exemplo de uma face que pode ser removida ou não, via teste das faces apontadas na direção visível ao observador.

Definindo o vetor normal de uma face e um vetor que indica a direção do ponto de visão de seu observador, calcula-se o produto vetorial dos mesmos. Caso o ângulo resultante seja superior a 90º, essa face não estará visível e retornará um valor negativo. Essa face só será renderizada caso o resultado retornado seja positivo.

Por fim, a última otimização realizada é um pouco mais complexa e sem soluções bem definidas até hoje. Como os cenários aplicados neste trabalho são ambientes fechados, muitas regiões podem pertencer ao frustum e estar com suas faces na direção do observador e não necessariamente estarem visíveis. Um claro exemplo disso são os objetos que são obstruídos por outros, como paredes ocludindo salas. Para isso, pode-se aplicar uma solução denominada por *Potentially Visible Sets* (PVS), ou *Conjuntos Potencialmente Visíveis*.

3.1.1.1 Conjuntos Potencialmente Visíveis

O problema da oclusão é complexo de se solucionar, pois não existem meios eficientes de se fazer esse cálculo em tempo real. Este é representado pelo objeto 2, na Figura 16. Algumas soluções envolvem a computação de estruturas como *Z-Buffers* ou árvores BSP com percorrimento de frente para trás, mas que são bastante custosas. É preciso tratar esse problema de maneira simples, eficiente e mais próxima possível da realidade (COHEN-OR et al., 2003).

Já que os cenários representados neste trabalho são estáticos, pode se utilizar uma estrutura previamente processada que já contenha informações relativas a visibilidade de uma região a partir de outra. Isso é feito na criação do mapa, em que se calcula a partir de um ponto de visão ou de uma célula, quais são os possíveis setores ou regiões visíveis. Realizado isso, escreve-se em um conjunto de bits quais são as células visíveis a partir de outras. Esse elemento é representado pelo atributo *BSPpvs* no diagrama de classes.

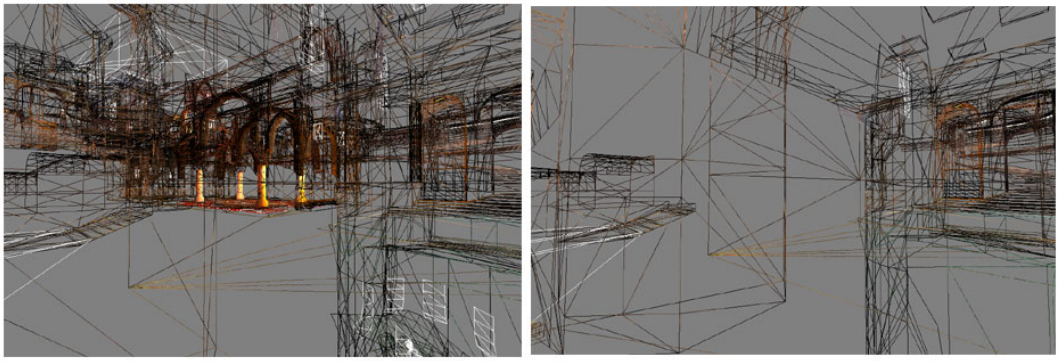

Figura 19: Exemplo de utilização do teste de oclusão.

O teste de oclusão realizado na função *drawLeaves()* é realizado pelo método *testVisibilityCluster()*. Neste, são passados dois identificadores que definem quais são as células a serem testadas, e a função retorna a um valor verdadeiro se um setor for potencialmente visível em relação a outro.

Esse teste de oclusão é realizado verificando se um setor é visível a partir da célula da câmera. Para isso se implementa a função *findCameraNode()*, que percorre a árvore de forma inversa ao *drawLeaves()*, buscando o setor o qual localiza-se o *frustum*. Após isso, ao longo das verificações do método *drawLeaves()*, são realizados os testes de oclusão.

3.1.2 Detecção de Colisão

Como exemplificado no capítulo anterior, também é necessário realizar um percorrimento em árvores BSP para detecção de colisão. A principal vantagem dessa abordagem é que, ao invés de se fazer testes de interseção por toda a geometria do mapa, esses são realizados apenas nos objetos localizados na célula a qual ocorreu a colisão.

Sabe-se que um personagem é composto por dezenas de complexos simpliciais em três dimensões, compondo uma malha triangulada. É extremamente ineficiente realizar a detecção de colisão para todos os vértices dessa malha, o que se faz necessário realizar uma aproximação desses elementos sobre uma superfície envolvente, normalmente uma caixa. A vantagem de se utilizar elementos envolventes é reduzir drasticamente o total de pontos a serem testados, sem perder a eficiência e o realismo.

Entretanto, esses objetos são dinâmicos e se movimentam constantemente ao longo do mapa, por isso, se faz necessário detectar a exata posição de uma colisão e tratar a reação da mesma, a fim de tornar a simulação realista. Para se fazer isso, deve-se determinar o ponto exato da colisão e ao atingí-lo finalizar sua animação, interrompendo sua movimentação.

A função do modelo computacional responsável pela detecção de colisão é a *collisionBox()*. Esta recebe justamente como parâmetros: os vértices que compõe a caixa envolvente ao personagem e os vértices inicial e final que definem o segmento de reta do trajeto do mesmo, em um dado instante. A função retorna um valor booleano que indica se ocorreu alguma colisão dentro de uma célula ou não.

Essa função realiza o mesmo procedimento exemplificado na solução de colisão do capítulo anterior. Iniciando-se do nodo raiz, os vértices inicial e final são testados em relação ao seu plano de particionamento, e determinado a qual célula filha esses pontos pertencem, continua-se o percorrimento. Caso ocorra de o trajeto interceptar um dos planos de particionamento ao longo do processo, esse segmento é subdividido no local onde ocorreu a interseção. Esse processo é repetido até que um nodo folha seja encontrado, e toda a geometria de seu interior seja testada com o objeto o qual se deseja averiguar a colisão, a partir de seu trajeto.

3.2 Classe para Personagens Animados

Deseja-se representar computacionalmente o esquema apresentado que utiliza esqueletos-chave. Essa abordagem é a melhor a ser utilizada, pois considera o movimento de cada articulação do modelo de maneira independente, e também pelo fato de seus cálculos serem mais simples por se limitarem a uma única interpolação angular sobre as juntas.

Entretanto, essa abordagem ainda exige um certo custo computacional, principalmente se aplicada sobre todos os vértices desse personagem. Por isso, essa interpolação é aplicada somente sobre o esqueleto interno a superfície do modelo, que é representado como um conjunto de segmentos de reta. Todos os vértices que compõe a malha do personagem são transformados utilizando esse esqueleto como referencial, via uma matriz de transformação de ossos.

É representada a seguir um esquema de representação por classes concretas, contendo o mínimo necessário para seu funcionamento. É baseada no formato de arquivo MDL, utilizada nos modelos animados do jogo *Half-Life*:

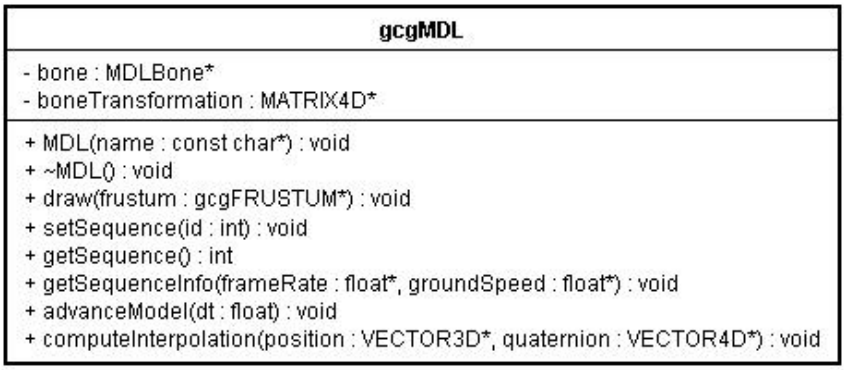

Figura 20: Diagrama de classes que representa um modelo animado.

A primeira estrutura importante a ser definida é um osso do esqueleto, representado no diagrama como *bone*. É primordial representá-lo computacionalmente ao menos pelos seguintes atributos: um identificador que defina o parentesco de ossos em relação a outros, para que as transformações dos pais possam ser aplicadas nos filhos, e dois vetores que representem o grau de liberdade dos mesmos. Esses vetores armazenam a posição e a orientação de um osso, respeitando a hierarquia.

De maneira análoga ao cenário estático, o construtor é utilizado para carregar os dados previamente armazenados no arquivo físico do modelo, como as texturas, a malha de triângulos, as animações etc. É repassado a ele o nome do arquivo em disco, e o

destrutor é utilizado para remover essas informações após o uso ou destruição do objeto *gcgMDL*.

Após armazenar todas as informações necessárias para a renderização do modelo, é necessário definir qual animação deve ser executada. Essa necessidade é satisfeita pela função *setSequence()*, que recebe o identificador da animação a ser executada. Para isso, todos os modelos devem possuir as mesmas animações e os mesmos identificadores, fato que ocorre em modelos baseados no mesmo formato de arquivo.

Logo em seguida, é preciso animar o modelo. Isso é primariamente realizado pela função *advanceModel()*, que a partir de um instante de tempo passado, calcula qual o próximo quadro do modelo. Esse quadro é de vital importância, pois a partir dele será lido do arquivo, quais são as transformações geométricas necessárias para a interpolação de animações.

A última etapa antes da renderização propriamente dita, seria calcular as interpolações referentes aos ossos do modelo, variando ao longo do tempo. Notadamente, isso pode ser realizado de duas formas: calculando de maneira independente a posição e a orientação de ossos, via uma interpolação linear e outra esférica, respectivamente, ou apenas por uma única interpolação linear esférica. Na classe apresentada, é realizado o primeiro procedimento. Nesse método é repassado para cada osso, sua posição e sua orientação e, assim, calculada sua interpolação.

Para facilitar os cálculos relativos a interpolação e as transformações geométricas, essas estruturas são sempre convertidas em *quaternions*. Obtém-se a partir delas um *quaternion* resultante, que é convertido novamente em uma matriz de ângulos de Euler. Esta, por fim, é aplicada sobre todos os vértices do modelo.

Entretanto, essa abordagem ainda é muito simplificada. É comum calcular mais de um *quaternion* para um mesmo modelo em uma etapa da animação e aplicá-los sobre um número bem específico de ossos, para criar um efeito realista e que enfatize a independência dos ossos. Essa implementação é realizada pela função *computeInterpolation()*.

O método *getSequence()* retorna via um identificador inteiro, qual a animação executada em um dado momento na aplicação. Essa informação é necessária para se determinar qual ação o personagem está realizando e assim poder ser decidido qual ação realizar.

Já o método *getSequenceInfo()* recebe dois atributos reais e atribui a eles qual a taxa de quadros de uma determinada animação do personagem e qual a velocidade do mesmo sobre o chão. Essas informações são necessárias para se simular a movimentação

do modelo.

A cada instante de tempo, da mesma maneira que no cenário, a função *draw()* é chamada para renderizar o modelo e o *frustum* utilizado para testar sua visibilidade a um usuário ou observador.

4 Aplicação

A partir da exemplificação das técnicas anteriores, é possível construir um ambiente virtual composto de um cenário interno estático e um grupo de personagens animados. Será apresentada uma aplicação desses conceitos por meio de um software de teatro virtual no qual ocorre a interação de agentes inteligentes, representados por personagens animados.

Essa aplicação, que será apresentada ao longo desse capítulo, engloba conceitos de Inteligência Artificial (IA) para a solução do problema de busca de caminhos. Esse software foi desenvolvido conjuntamente com o trabalho final de bacharelado do formando Maurício Archanjo, da Universidade Federal de Juiz de Fora (COELHO, 2007). O objetivo dessa implementação é demonstrar, via um exemplo prático, como todo esse estudo pode ser aplicado e quais as vantagens de integrá-lo com uma grande área da computação, como a IA.

O objetivo dessa implementação é representar um teatro virtual habitado por agentes. Esse cenário é representado pela classe concreta de árvores BSP proposta e por personagens animados que realizem ações e possam interagir utilizando os objetos gerados pela classe MDL apresentada. Nessa, foi utilizada a linguagem de programação C/C++ e o sistema gráfico OpenGL.

4.1 Contexto da Aplicação

A representação de ambientes virtuais que contenham um grupo de agentes possuem uma série de aplicações, dentre elas, desenvolver entidades capazes de interagir com um usuário, ou com demais agentes, com o intuito de se simular ou representar um evento específico, ou algum tipo de estudo. Esse tipo de trabalho é muito empregado em indústrias de entretenimento, simulações como representações biológicas etc.

Na aplicação descrita, pretende-se simular um comportamento entre um personagem

perseguidor e outro perseguido, e verificar quais são os resultados.

Para se representar e visualizar o cenário, foi utilizado o formato de arquivo BSP do jogo *Quake 3*, da ID Software. A leitura de dados do cenário é realizada via um arquivo físico pré-processado, em que é feito o parser de suas informações com o intuito de se construir uma árvore BSP. Monta-se então uma estrutura de dados baseada no diagrama de classes apresentado anteriormente.

Nessa aplicação, todos os objetos geométricos são classificados em três tipos: os polígonos, as malhas e as superfícies de Bézier. A primeira é representada como um objeto com número variável de vértices, o segundo é um objeto constituído de uma malha de triângulos, e o último representa superfícies curvas. Esta última é interpolada a partir de equações lineares.

Outro aspecto importante a ser levado em consideração para se gerar um cenário realista é a aplicação de textura. Neste trabalho foi utilizado o conceito de *multitextura* que refere-se a capacidade de se mapear e aplicar mais de uma imagem sobre um mesmo fragmento, simultaneamente. Estas se complementam, de forma que é gerado um efeito visual bem mais realista.

Nesse ambiente foram utilizadas dois tipos de texturas com multitexturização, que são: a fotografia a ser mapeada sobre um objeto e um mapa de luzes que define as cores aplicadas sobre o mesmo.

Realizados esses procedimentos iniciais, toda o restante da implementação se baseia no modelo computacional apresentado no capítulo anterior.

Para a implementação dos modelos representando personagens animados, foi utilizado o formato de arquivo aberto mdl, utilizado em diversas versões de jogos como *Quake* e *Half-Life*. Para o carregamento e a exibição do modelo animado, utiliza-se o formato de arquivo do *Half-Life* e parte do código-fonte da *Source SDK Engine*, utilizada na produção do jogo.

De maneira análoga ao cenário, antes da renderização, são carregados os dados referentes a textura e geometria do modelo.

Esse formato de arquivo possibilita algumas manipulações, como por exemplo a, de controlador de ossos. Essa estrutura dá a possibilidade de se definir a movimentação de um grupo de ossos, flexibilizando e modificando durante sua execução.

Por fim, outra estrutura bem específica desse formato são as caixas envolventes in-

corporadas por todo o personagem. Estas são úteis para se calcular a colisão de uma determinada parte do corpo do modelo, como uma mão, o que lhe dá maior realismo.

4.2 Abstração de Agentes

Para se utilizar uma abordagem que envolva agentes, é preciso que os mesmos interpretem dados geométricos advindos do teatro virtual e gerenciem o comportamento de um personagem. Esse processo é realizado por uma estrutura denominada *agente*, que independa do cenário ou do modelo animado. A representação de um agente sob forma de classes abstratas é vista na Figura 21.

Um agente deve realizar distintamente duas tarefas: interpretar os dados geométricos recebidos para decidir uma ação e executá-la fisicamente. Já que essas tarefas tem características e soluções bem específicas e, o comportamento dessas podem variar de acordo com o agente utilizado, é conveniente tratá-las independentemente. Para isso são empregadas duas abstrações computacionais denominadas *Mente* e *Corpo*.

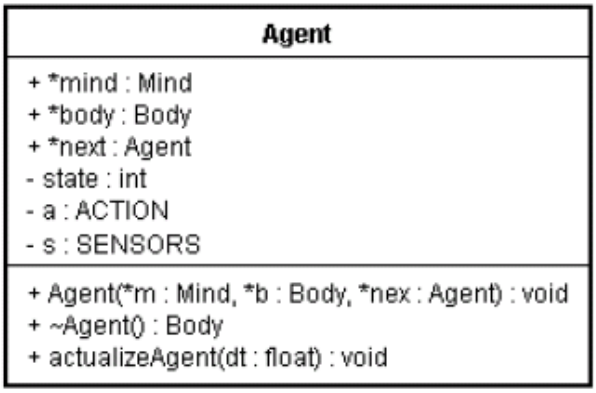

Figura 21: Diagrama da classe abstrata do Agente.

Uma das principais vantagens em se utilizar classes abstratas para representá-las é o fato de que isso lhe permite escalabilidade (COELHO, 2007). Assim, várias mentes podem ser implementadas e utilizadas por corpos diferentes e vice-versa, o que constituiria em dezenas de possíveis agentes.

Além de possuir uma mente e um corpo, um agente também armazena internamente qual ação que está sendo realizada. Essas podem ser:

- *STOPPED_STAND*: o agente deve parar;
- *RUNNING_STAND*: o agente deve correr;

- *WALKING_STAND*: o agente deve andar;
- *STOPPED_CRAW*: o agente deve agachar;
- *WALKING_CRAW*: o agente deve andar agachado;
- *JUMP_UP*: o agente deve saltar para cima;
- *JUMP_FRONT*: o agente deve saltar para a frente;
- *DYING*: o agente deve morrer;
- *DEAD*: o agente está morto e não pode realizar nenhuma ação.

Por fim são armazenados os sensores, os quais são analisados e usados para a tomada de decisões e o conjunto de ações propriamente ditas.

Para definir o comportamento do agente, utiliza-se o método *actualizeAgent()*. Esse método decide a ação a qual o mesmo deve realizar, verificando o estado atual do ambiente e a passagem de tempo via uma máquina de estados.

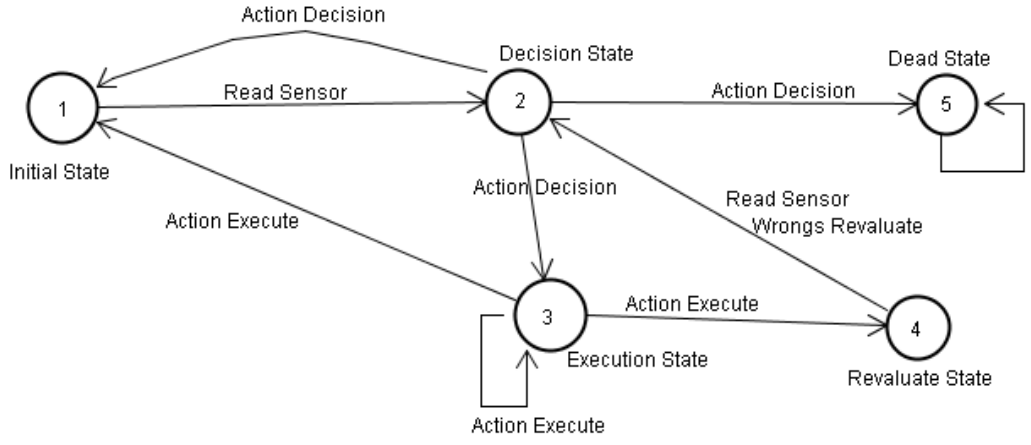

Figura 22: Máquina de estados que representa o comportamento do Agente.

Os estados possíveis do Agente são:

- *INITIAL_STATE*: inicializa a máquina de estados;
- *DECISION_STATE*: momento o qual é escolhida uma ação;
- *EXECUTION_STATE*: a execução da ação;

- *DEAD_STATE*: caso esse estado seja alcançado, o agente morre e não realiza mais nenhuma ação;
- *REVALUATE_STATE*: caso alguma ação não possa ser realizada, é avaliado o que aconteceu e como esse problema é tratado;

Por fim, representa-se o seu estado atual, tal qual o andamento de suas ações da seguinte maneira:

- *EXECUTED_STATUS*: ativada quando uma ação acaba de ser executada;
- *CURRENT_STATUS*: permanece ativada enquanto a ação não for finalizada;
- *COLLISIONERROR_STATUS*: chamada quando ocorre colisão entre o agente e o ambiente;
- *TIRED_STATUS*: determina um instante o qual o agente possa descansar, depois de um certo intervalo de tempo;

Definido um agente, é preciso obter quais são os objetivos e o que as abstrações, mente e corpo, devem realizar. A tarefa da mente é, a partir de informações geométricas obtidas, decidir qual ação o agente deve realizar, seja ela uma sequência como parar, correr ou morrer, e calcular quais as posições geométricas o mesmo deve usar a fim de chegar em seu destino. A execução dessas ações é realizada pelo corpo. Agentes são representados como na Figura 23.

Figura 23: Exemplos de agentes.

Já o corpo deve ser capaz de repassar para a mente toda a geometria que é possível percorrer, e realizar as ações como uma sequência de animação, ou o percurso propriamente dito no cenário. Também é função do corpo verificar se é possível concretizar uma ação ou não. Uma abstração para corpo e mente é apresentada como a seguir:

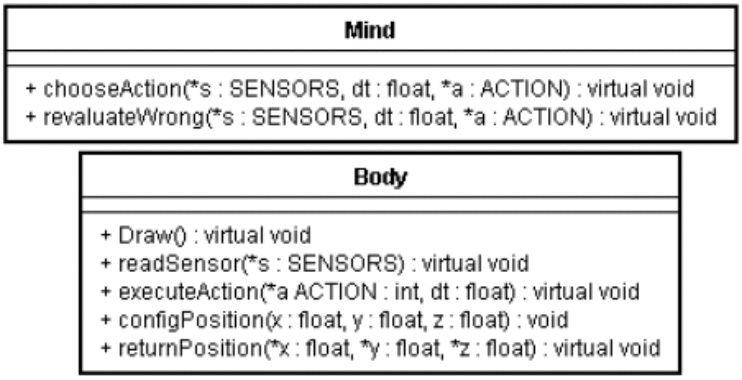

Figura 24: Diagrama das classes abstratas da Mente e do Corpo.

Para permitir o diálogo entre essas estruturas, são definidos os sensores e os conjuntos de ações, representados como *sensors* e *action* no diagrama. Um sensor é uma estrutura que armazena informação geométrica refinada a partir da árvore BSP, e por meio dela são realizados os cálculos da mente. Já um conjunto de ações detém dados ligados a ação corrente de um agente, a velocidade a qual ele deve movimentar-se para realizá-la, sua posição final etc.

A abstração "'Mente"' implementa dois métodos. O primeiro deles é o *chooseAction()*, que determina qual ação o agente deve realizar a partir dos sensores e do conjunto de ações realizados até então. O segundo é o *revaluateWrongs()*, que só é chamado quando uma ação não for possível de ser realizada, por exemplo, quando o personagem tenta caminhar até uma determinada posição e o mesmo é impedido por uma parede. Esse deve determinar uma nova tarefa.

A abstração "'Corpo"' executa os comandos determinados pela mente. Para isso são lidos os sensores que são direcionados a mente, via o método *readSensor()*. Quando a Mente retornar a ação a qual o corpo deve realizar, a mesma é executada pelo método *executeAction()*, que verifica se a mesma pode realizar a ação ou não, e a executa. As outras funções são relativas a retornar ou definir uma nova posição ao agente, e desenhar o modelo de modo satisfatório.

4.2.1 Implementações da Mente

Neste trabalho são representadas três classes concretas a partir da Mente apresentada, que são: a Ameba, o Predador e a Presa. Na implementação, elas são definidas como: *Amoeba*, *Predator* e *Prey*.

A classe *Amoeba* realiza uma busca de caminhos a partir de pontos inicial e final decididos aleatóriamente.

As classes *Predator* e *Prey* são dependentes. O objetivo da primeira é alcançar a posição da segunda, enquanto que a última deve buscar um caminho tal qual seja o mais afastado possível da primeira. Essas classes herdam de *Amoeba* e são mais complexas, pois a mudança da posição final ao longo da busca exige a reinicialização do algoritmo de busca algumas vezes.

4.2.2 Implementação do Corpo

Será proposta uma abstração de um Corpo que utilize todos os conceitos envolvidos no modelo computacional de um personagem apresentado.

4.2.2.1 Classe MDLBody

Uma implementação de Corpo é uma entidade que deve ser capaz de processar geometria no formato de sensores e realizar ações, sendo capaz de tratar casos onde não é possível realizá-la. Isso é representado nessa classe, por meio de modelos animados como sugerido no tópico anterior. Seu diagrama de classes é definido na Figura 25.

Na criação de um objeto *MDLBody*, é repassado ao seu construtor o nome do arquivo físico do modelo ao qual se deseja aplicar o Corpo. Também é passado como parâmetro uma lista de triângulos necessária para que o agente possa realizar sua busca e caminhar pelo ambiente, independente do espaço geométrico ou do esquema de representação. O último parâmetro é o numero de triângulos contidos nessa lista. Os outros parâmetros definem a posição do corpo e o seu ângulo de orientação.

A estrutura *BSPEnvironment* é responsável por fazer a interface entre as informações geométricas do ambiente, que no caso advém de uma árvore BSP e o algoritmo de busca de caminhos empregado. Neste são calculadas as heurísticas, o nó alvo, dentre as outras informações relativas a busca, independente de qual seja. Por isso, essa é representada na forma de classe abstrata.

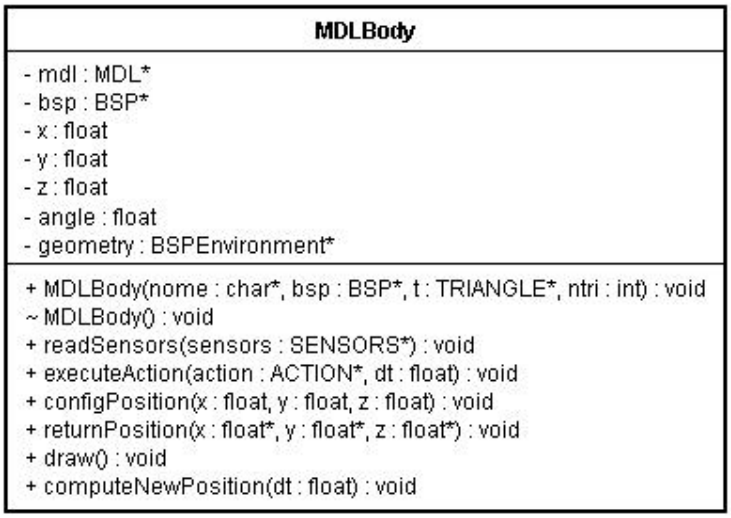

Figura 25: Diagrama de classe do MDLBody.

Logo em seguida, é chamada a função *executeAction()*. Nesta, é analisada a ação que o corpo está realizando para, então, verificar se a mesma é possível ou não. Também é feito o cálculo dos parâmetros de movimentação do personagem, como por exemplo o ângulo que ele deve se dirigir ao seu destino e sua velocidade. Caso a ação possa ser realizada, o método *computeNewPosition()* é invocado, o que determina a nova posição do personagem.

O método *computeNewPosition()* retorna verdadeiro ou falso se a nova posição a qual o corpo deve percorrer for possível ou não de ser alcançada, em um dado momento. A partir de sua velocidade e intervalo de tempo, a nova posição é calculada. Um exemplo de impossibilidade de realizar a ação é, por exemplo, o que deve ser realizado caso o corpo colida com algum objeto, como uma parede.

Para se detectar uma colisão, o método referente a ela no modelo computacional de cenários é invocado. É passado, por parâmetros, a caixa envolvente ao personagem em um dado momento da animação, e os pontos iniciais e finais que definem seu trajeto. Se for detectada uma colisão, o método *computeNewPosition()* retorna falso, o que implica em retornar a máquina de estados do agente e reavaliar suas ações.

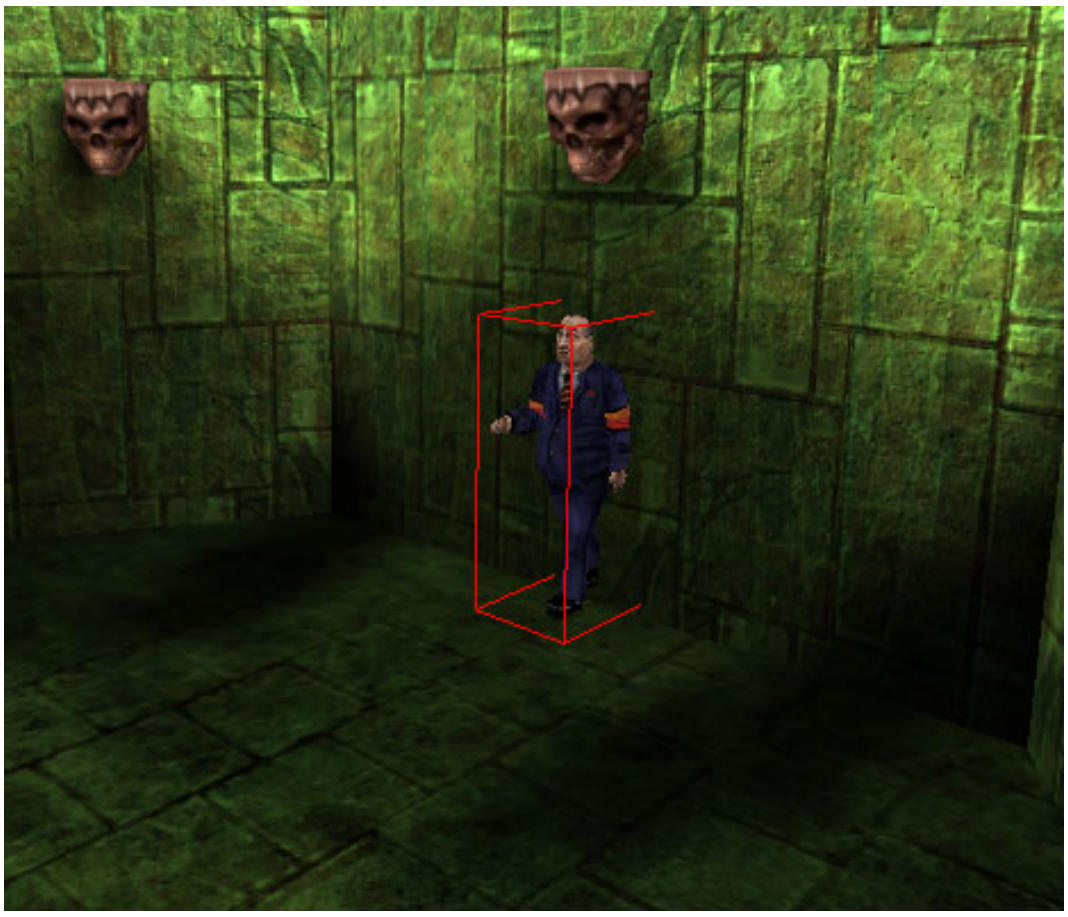

Figura 26: Teste de colisão, ao qual deseja verificar se a geometria envolvente ao objeto intercepta parte do cenário.

5 Conclusão

Neste trabalho foram vistos os conceitos necessários para a visualização e a manutenção de um ambiente virtual, desde o panorama inicial que apresenta as motivações para a utilização dos mesmos, até uma demonstração prática de seu funcionamento.

A apresentação dos métodos que envolvem realidade virtual em 3D foram divididos e explicados a partir de seus dois elementos principais: os cenários virtuais e os personagens animados. Foram utilizadas as abordagens mais utilizadas na literatura para a resolução de problemas envolvendo visibilidade, representação e simulação dos mesmos.

Para os cenários, foi descrito o método de Particionamento de Espaços, que é vantajoso para ser utilizado, pois subdivide o problema em pequenas classes, facilitando sua solução. Para os personagens, foi apresentado um esquema que utiliza esqueletos e quadros-chave. Esse é mais interessante de ser utilizado pois a animação se torna mais flexível e realista, já que cada articulação do corpo age de maneira independente. O armazenamento prévio de quadros também permite armazenar a menor quantidade possível de animações.

Após essas conceituações, para esclarecer a eficácia dessas abordagens, foi realizada uma implementação computacional. A partir dos diagramas de classes, foi descrito passo a passo como é o funcionamento de uma aplicação e como são verificadas as suas soluções. Como se sabe, é necessário determinar quais são as superfícies visíveis de um cenário para minimizar o total de operações realizadas na placa gráfica, otimizando, e muito, o desempenho de aplicações em tempo real. Ao invés de realizar esses cálculos ao longo de todo o cenário, isso é feito em regiões restritas, graças ao método de Particionamento de Espaços.

Em animação de personagens, o processo computacional se resume a definir a posição e a orientação de cada articulação do corpo, via equações de interpolação. Como o esqueleto é hierarquicamente dependente de seus ossos, as transformações geométricas realizadas sobre cada articulação devem interferir nas outras.

Para exemplificar como os métodos descritos podem ser utilizados, esses foram in-

corporados em um trabalho de outra natureza, para a busca de caminhos em espaços tridimensionais. A união dos dois estudos tornou possível a criação de um ambiente rico, no qual foi realizada a simulação de agentes inteligentes sobre um ambiente virtual. Aplicações em que implementações desse porte podem ser utilizadas são: educacionais, entretenimento, simulação de eventos ou comportamento específicos etc. Essa implementação foi desenvolvida em inglês, com o intuito de deixá-la livre para utilização.

É sugerido como trabalho futuro estender os conceitos exemplificados para ambientes abertos, como terrenos e cidades. Nestes, outras otimizações devem ser levadas em consideração, como nível de detalhe e outras estruturas, que podem ser utilizadas para representar os espaços, tais como *octrees*. Também podem ser verificados o funcionamento desses e outros métodos em ambientes dinâmicos, utilizando abordagens que envolvam portais.

Referências

ABRASH, M. *Michael Abrash's Graphics Programming Black Book*. [S.l.]: Coriolis Group Books, 1997. 1342 p.

BERGEN, G. van den. *Collision Detection in Interactive 3D Environments*. [S.l.]: Morgan Kaufmann, 2003. 277 p.

COELHO, M. A. N. *Busca de Caminhos em Espaços 3D*. Dissertação (Mestrado) — Universidade Federal de Juiz de Fora, 2007.

COHEN-OR, D. et al. A survey of visibility for walkthrough applications. 2003.

EBERLY, D. H. *3D Game Engine Design : A Practical Approach to Real-Time Computer Graphics*. [S.l.]: Morgan Kaufmann, 2000. 561 p.

FERGUSON, R. S. *Pratical Algorithms for 3D Computer Graphics*. [S.l.]: A K Peters, 2001. 537 p.

FOLEY et al. *Introduction to Computer Graphics*. [S.l.]: Addison Wesley, 1993. 632 p.

MEDEIROS, E. et al. *Uma Abordagem Estocastica para Objetos Solidos*. 2007.

MOLLER, T.; HAINES, E. *Rendering Real Time*. [S.l.]: AK Peters, 2002. 900 p.

PIPHO, E. *Focus on 3D Models*. [S.l.]: Course Technology PTR, 2002. 232 p.

RANTA-ESKOLA, S. *Binary Space Partioning Trees and Polygon Removal in Real Time 3D Rendering*. 2001.

SAMET, H. *Applications of Spatial Data Structures*. [S.l.]: Addison Wesley, 1990. 516 p.

THALMANN, N. M.; THALMANN, D. Computer animation in future technologies. 1996.