UNIVERSIDADE FEDERAL FLUMINENSE

DANIEL RIBEIRO PIRES

Posicionamento de Câmeras por meio da Simulação Física

NITERÓI

2009

UNIVERSIDADE FEDERAL FLUMINENSE

DANIEL RIBEIRO PIRES

Posicionamento de Câmeras por meio da Simulação Física

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Computação Visual e Interfaces.

Orientador:

Prof. Esteban Walter Gonzalez Clua, D.Sc.

NITERÓI

2009

Posicionamento de Câmeras por meio da Simulação Física

DANIEL RIBEIRO PIRES

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Computação Visual e Interfaces.

Aprovada por:

_____

Prof. D.Sc. Esteban Walter Gonzalez Clua IC-UFF

_____

Prof. D.Sc. Bruno Feijó PUC-Rio

_____

Prof. D.Sc. Anselmo Antunes Montenegro IC-UFF

Niterói, 18 de Março de 2009

Agradecimentos

Primeiramente agradeço ao meu orientador, professor Esteban, pelas lições, esclarecimentos e apoio.

Ao amigo Erick por todas as dicas e contribuições.

Aos amigos do Media Lab pelas horas de conversas de nerd.

Aos professores Anselmo e Aura por estarem sempre presentes.

À minha namorada, a Cris, pelo apoio e compreensão.

Aos meus pais, que sempre me ajudaram.

À CAPES pelo apoio financeiro concedido.

Resumo

Nos ambientes virtuais 3D, tais como nos usados em jogos 3D ou ambientes de realidade virtual, a liberdade para o posicionamento de câmeras vêm permitindo que as aplicações utilizem técnicas de filmagem semelhantes às usadas nas apresentações cinematográficas. Porém, devido à interatividade nesses ambientes, não existe uma sequência predeterminada de acontecimentos que permita o posicionamento dinâmico ideal. Faz-se necessária, então, a utilização de técnicas de reconhecimento e previsão de ações para se posicionar a câmera de forma satisfatória. Este trabalho apresenta uma solução para o posicionamento automático de câmeras baseada em técnicas de reconhecimento e previsão de ações no ambiente virtual interativo, através de agentes de *software*, com base no estado da simulação física.

Palavras-chave: câmeras virtuais, previsão de ações, cinematografia

Abstract

In 3D virtual environments, such as those used for 3D Games or Virtual Reality, freedom for the positioning of cameras are enabling applications to use techniques similar to those used in live action film productions. However, because of the interactivity in these environments, there is no predetermined sequence of events that enables an ideal dynamic positioning. Therefore, it is necessary the usage of techniques for recognition and prediction of actions to position the camera in a satisfactory manner. This paper presents a solution for the automatic placement of cameras based on action recognition and prediction techniques in interactive virtual environment, through software agents, based on the physics simulation state.

Keywords: virtual cameras, actions prediction, cinematography

Siglas e Abreviações

DSL	:	*Domain-Specific Language*
PNF	:	*Past-Now-Future*
XNA	:	*XNA is Not Acronymed*
HUD	:	*Head-Up Display*
C#	:	C Sharp

Sumário

Lista de Figuras

Capítulo 1

Introdução

A câmera é o elemento responsável por captar as imagens de alguma ação em algum ambiente. Posteriormente, quando as imagens forem exibidas, a ação será mostrada de acordo com o ponto de vista daquela câmera. Segundo os profissionais do cinema, a câmera não é um simples instrumento de registro de imagens, mas um instrumento criador de imagens. Percebe-se, então, a subjetividade da produção cinematográfica. O cinema fez evoluir durante muito tempo as técnicas de filmagem que hoje em dia são aceitas naturalmente pelos espectadores. Segundo a cinematografia, a câmera não deve ser posicionada com a função de apenas transmitir objetivamente o que se passa no mundo, mas também visando ajudar o espectador a interpretar o desenvolvimento do roteiro. A cinematografia criou o que chamamos de linguagem cinematográfica, definida como uma série de regras para capturar ações específicas através de um conjunto de tomadas [Martin 1985]. Esta linguagem especifica posicionamentos e transições de tomadas com o objetivo de tornar a ação mais coerente ao espectador.

Com o advento da tecnologia para geração de gráficos 3D, o conceito de uma câmera virtual se tornou indispensável, estando intimamente ligado a outros conceitos como objeto 3D e ambiente 3D. Neste contexto, um objeto 3D fica

localizado em um ambiente 3D e é exibido ao mundo exterior através de uma janela 3D, segundo as propriedades da câmera virtual.

Câmeras virtuais possuem algumas propriedades próprias que as diferem das câmeras do mundo real. A princípio elas são desprovidas de atributos físicos como massa e volume, e podem ser posicionadas e orientadas livremente pelo ambiente 3D, podendo inclusive atravessar outros objetos. Além disso, câmeras virtuais possuem limites de visualização bem definidos, o que informará ao processador gráfico quais objetos ou polígonos serão exibidos.

Ao se manipular uma câmera virtual sem se considerar propriedades físicas, obtem-se uma apresentação de uma cena 3D um tanto quanto irreal e estilizada. Com o objetivo de tornar a simulação e a exibição mais próximas da realidade, desenvolvedores e designers passaram a modelar câmeras virtuais com atributos físicos, permitindo assim que esta possuísse uma aceleração quando se movimenta e que levasse em consideração colisões com outros objetos da cena.

Na cinematografia atual, a câmera é um dos principais elementos capazes de realçar o que se quer mostrar, além de introduzir subjetividade à cena dependendo do ângulo de visualização, do que está sendo focado e do tempo de tomada da cena [Martin 1985]. Com o passar do tempo, os desenvolvedores de jogos 3D se inspiraram nas produções cinematográficas para adicionar mais efeitos e emoção às suas produções [Hawkins 2005], gerando mais um atrativo para prender a atenção do jogador. Foram criadas vários tipos de comportamentos de câmera, cada uma imprimindo um significado à ação do ambiente. Termos técnicos como *zoom* e *pan* foram incorporados à interface de movimentação da câmera virtual. De fato, a programação de uma câmera virtual vem se tornando cada vez mais parecida com os comandos que um operador de câmera do mundo real receberia, com instruções de mais alto nível. Hoje já existem diversas DSL's (*Domain-Specific Language* – linguagem específica de domínio) com este fim. A experiência da indústria do cinema explica e justifica o uso de cada tipo de tomada de câmera, constituindo assim uma linguagem própria, que também é aproveitada na produção de apresentações e jogos 3D.

Com o recente aperfeiçoamento da tecnologia de redes e da capacidade de processamento dos computadores, tornou-se possível que partidas de jogos eletrônicos sejam transmitidas em massa para um grupo de telespectadores [Drucker 1994], exatamente como já acontece com os eventos esportivos. Com isso, este conjunto de usuários telespectadores fica impossibilitado de interagir com o ambiente virtual, mas ainda podem controlar de alguma forma que parte da ação ou do ambiente que querem assitir; alternativamente podem ter a mesma visão de um dos jogadores. Uma abordagem semelhante é utilizada em *replays* interativos, onde o usuário pode controlar sua câmera ou selecionar câmeras fixas ou comportamentos de câmera. Como exemplos pode-se citar jogos como Counter Strike [Valve 2009] ou Warcraft 3 [Blizzard 2009], onde os usuários se conectam e podem assistir à partida, mas nestes casos, cada usuário precisa controlar sua própria câmera, o que na maioria das vezes não traz um bom efeito cinematográfico.

No entanto, existem poucas alternativas para que os usuários telespectadores assistam a transmissão ou o *replay* de uma forma mais automática e organizada no que diz respeito à boa visualização e continuidade dos acontecimentos, exatamente como em um filme. Primeiramente, em aplicações 3D interativas, uma boa utilização de uma câmera virtual é bastante dificultada devido à imprevisibilidade dos acontecimentos. Em aplicações com seqüências pré-definidas, já se sabe exatamente o que vai acontecer com os elementos da cena, e isso torna possível o posicionamento e orientação ideais da câmera, de acordo com os resultados desejados. Porém, quando não se sabe previamente as transformações que podem acontecer com os objetos do ambiente 3D, é necessário se utilizar técnicas alternativas para a manipulação da câmera. É preciso, principalmente, reconhecer as ações e identificar os objetos relevantes para que a cena seja transmitida de forma apropriada, ainda considerando-se a probabilidade da ocorrência de algum outro evento. Após o reconhecimento das ações no ambiente interativo, o sistema precisa ainda escolher qual é a mais relevante a cada instante, exibindo-a de forma conveniente com a emoção esperada. Para isso é preciso escolher uma boa posição de câmera, definir o que será focado e determinar o tempo da tomada.

Neste trabalho é proposta e explorada uma técnica que, tendo escolhido uma ação para ser filmada, realiza-se um posiciomamento de câmera de baixo custo computacional baseado na simulação física, prevendo o que irá acontecer nos próximos momentos dentro do ambiente virtual. O processo é apoiado por agentes de software que ficam monitorando o ambiente para reconhecimento de ações e montagem da cena. Para exemplificar o uso da técnica foi desenvolvido um jogo, onde o usuário tem a sua visão de jogo, e após sua jogada é exibido um *replay* de toda a ação, o que também representaria a visão de um telespectador numa transmissão em tempo real.

Este trabalho está organizado da seguinte forma. Primeiramente é apresentado um panorama sobre cinematografia e como ela vem sendo aplicada nos games. Logo após, no capítulo 3, são discutidos os trabalhos mais relevantes relacionados ao problema de posicionamento de câmeras em ambientes virtuais. A partir do capítulo 4 são apresentados as contribuições deste trabalho: técnicas que auxiliam na construção de uma boa seqüência cinematográfica envolvendo uma partida de um *game*. São também abordados o problema, as soluções propostas e resultados obtidos, explicando todas as particularidades do jogo produzido como exemplo e das ferramentas utilizadas. Por fim, no capítulo 7, são apresentadas as conclusões e referências bibliográficas.

Capítulo 2

Cinematografia

Cinematografia é uma ciência que vem se desenvolvendo há muitas décadas, e criou com o tempo o que chamamos de linguagem cinematográfica [Martin 1985]. Essa linguagem define um conjunto de regras para transmitir ao telespectador uma certa emoção, dependendo da organização das tomadas. Isso inclui posicionamento da câmera, iluminação, tempo dos cortes e seqüenciamento das cenas. Por exemplo, em uma filmagem de um diálogo, os posicionamentos, efeitos e transições de câmera indicam que tipo de emoção se quer transmitir. Uma câmera filmando uma pessoa de baixo para cima, por exemplo, exibindo também o teto ou o céu, traz a impressão de poder ou domínio da situação para esta pessoa [Hawkins 2005].

Devido à popularidade do cinema e ao desenvolvimento da linguagem cinematográfica, as pessoas têm esta forma de comunicação como algo natural. Quando aplicada de forma incorreta ou diferente da usual, porém, pode deixar o telespectador confuso ou levá-lo a uma perda de imersão. Às vezes a intenção do diretor em um certo momento pode ser mesmo instaurar a confusão, mas caso contrário, pode trazer uma interpretação errada da cena ou da produção como um todo. O objetivo principal da cinematografia é, por meio das imagens exibidas, deixar

as ações mais coerentes ao telespectador, tornando sua experiência mais agradável e interessante.

Um entendimento de posição de câmera define os conceitos necessários para filmar ação em uma área fixa [Hawkins 2005]. Através dos cortes e movimentos de câmera consegue-se ampliar a área de visualização que a câmera pode atingir. Os movimentos de câmera em si também possuem sua influência na forma de contar estórias. É uma importante ferramenta para criação de filmes, mas é necessário um uso correto.

As principais questões sobre o posicionamento de câmeras nas produções cinematográficas são relacionados ao enquadramento da imagem, ao ângulo e ao movimento da câmera. Juntas, estas características definem o que deve ser filmado. Através do enquadramento, escolhem-se os componentes da cena que serão captados. Com ele estudam-se os arranjos dos elementos dentro da cena, os tamanhos relativos das pessoas e objetos em relação à imagem gerada, e o foco dos componentes.

O ângulo da câmera diz respeito à inclinação da câmera em relação aos elementos de cena. Por exemplo, uma pessoa filmada em *plongée* (de cima para baixo) perde sua força e parece mais vulnerável. Ao contrário, com uma imagem em *contra-plongée* ressalta-se a imponência da figura em cena [Filho 2007]. Outra angulação notável da câmera é o *tilt*, que faz as imagens parecerem ligeiramente inclinadas, ressaltando assim uma inquietação no personagem focado.

Praticamente uma necessidade nas produções atuais, a movimentação de câmera enriquece os trabalhos cinematográficos. Movimentando a câmera consegue-se filmar uma área maior do que se a câmera estivesse estática. O movimento permite também prender a atenção do telespectador em um objeto que se movimenta, bastando para isso enquadrá-lo enquanto se movimenta. De fato, a motivação mais comum para mover a câmera é o movimento do objeto focado [Hawkins 2005]. Entre os movimentos de câmera mais comuns tem-se:

- *Pan*: Movimento que fixa a câmera em um ponto e a rotaciona para mostrar uma área. É muito usado para mostrar o que há em volta, tal como o cenário. Também conecta dois objetos na cena, iniciando o movimento focando um objeto e terminando focando em outro. Se a audiência já sabe sobre a conexão, ajudará a fixar o conhecimento. Também é usado para seguir a ação de objetos que se deslocam.
- *Dolly*: Também conhecido como *track*, é o movimento de mover a câmera em linha reta, para frente ou para trás, para a esquerda ou direita, focando ou não um objeto específico. Com isso obtém-se um efeito de *parallax* e profundidade. É usado para dar uma firme impressão do posicionamento do ator na cena. Também é usado para seguir um objeto que se movimenta.
- *Zoom*: Não se trata de um movimento, mas uma mudança no foco da câmera. É importante quando se quer filmar o que há a uma certa distância quando o ator principal está longe.
- Caminhos: Normalmente definem a trajetória da câmera por meio de aparelhos como trilhos e braços mecânicos. Servem para dar um efeito mais dramático à cena, normalmente quando há muitos objetos se movimentando. Esta forma de movimentação ajuda a manter a informação de localidade dos objetos sem que sejam usados cortes.

Movimentos de câmera, tal como no cinema se tornaram parte integrante dos jogos desde que se tornaram tridimensionais. A câmera age como uma interface para o mundo do jogo, mostrando o que o jogador precisa ver enquanto joga. Dada a dinâmica do ambiente, isto pode não acontecer perfeitamente todo o tempo, principalmente em jogos com visão em primeira pessoa.

Posicionar uma câmera real no local em que o diretor deseja pode envolver o uso de braços mecânicos, trilhos e outros equipamentos especializados [Martin 1985]. Dependendo do tempo e do orçamento, pode acontecer que o diretor use uma posição diferente da ideal. Felizmente esse tipo de problema não existe em ambientes 3D, mas mesmo assim algumas questões devem ser consideradas. Dentro de um ambiente 3D é possível posicionar e mover a câmera livremente. Mas vale lembrar que em

ambientes virtuais de tempo real um planejamento das cenas e dos acontecimentos pode ser impossível, e isso pode acarretar alguns problemas. A existência de um obstáculo entre a câmera e o alvo pode representar um problema difícil de detectar e de prevenir. Toda a computação consumida para enquadrar o objeto na tomada pode ter sido em vão se existirem obstáculos que bloqueiam a visão. Deixar os obstáculos invisíveis pode ser uma solução, mas pode se tornar uma armadilha, pois afeta a consistência espacial da cena, devendo este recurso portanto ser usado com cautela. Outro problema que pode surgir é em relação à movimentação da câmera. Se ela passa por dentro de objetos, os polígonos são vistos com suas normais invertidas, torna o ambiente fisicamente irreal, e conseqüentemente a cena se torna incoerente para o telespectador.

2.1 Cinematografia nos Games

A evolução dos *games* pode ser entendida como um aumento de sua complexidade. Com o passar dos anos, o jogo propriamente dito foi acrescido de estórias, narrativas e significados. Com a evolução tecnológica, os *games* passaram a contar com novos recursos e mídias para contar estórias, entre elas o uso de seqüências cinematográficas e interfaces de jogo que lembram cenas de filmes.

Nos *games* a cinematografia pode ser usada com os mesmos propósitos que no cinema, tal como para as apresentações e passagens pré-definidas. Mas para o ambiente de jogo é preciso fazer algumas adaptações devido à sua natureza dinâmica e para não prejudicar a jogabilidade. Hoje em dia os posicionamentos de câmera, definindo como será a visão do jogador, são uma importante questão de usabilidade a ser considerada. Dependendo do grau de interatividade, os conceitos cinematográficos podem ser suprimidos ou adaptados em um certo grau. Por exemplo, nas primeiras versões do game Resident Evil [Capcom 2009], a câmera era fixa mas colocada em pontos que já traziam alguma expectativa de ação ou dificuldade. Em outros jogos, como God of War [Sony 2009.a], a experiência cinematográfica se dá através da movimentação da câmera seguindo uma trajetória pré-definida dependendo da localização do personagem principal no cenário. Mas em jogos como

Super Mario 64 [Nintendo 2009], maximizou-se a interatividade e a cinematografia foi ligeiramente sacrificada. Neste jogo é possível controlar a câmera livremente, além do personagem. A Figura 1 ilustra os jogos citados.

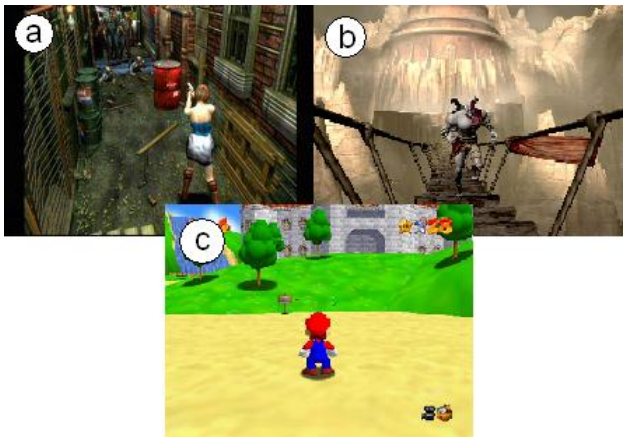

Figura 1: Resident Evil (a); God of War (b); Super Mario 64 (c).

Outras duas aplicações com alto apelo cinematográfico são a exibição de *replays* e a exibição dos acontecimentos do jogo para uma platéia puramente espectadora, em tempo real. Para estas aplicações, os conceitos da cinematografia podem ser aplicados em totalidade, visando mostar a ação da forma mais atraente possível. Porém, a maioria dos jogos que exibem *replays* ou que transmitem o jogo em tempo real não se preocupam em posicionar a câmera de forma a exibir a ação como num filme. Muitos disponibilizam a opção de manipular a câmera manualmente, o que às vezes sacrifica a cinematografia em função da interação do jogador. Para os jogos que fazem uma apresentação cinematográfica da partida, nota-se que as imagens geradas para o telespectador são diferentes das exibidas para o jogador. Com as imagens do *replay* a jogabilidade seria muito afetada, chegando ao ponto de impossibilitar o jogo. Por exemplo, o jogo Gran Turismo 3 [Sony 2009.b], mostrado na figura 2, tem um modo de exibição de *replays* com alguns modos de câmera. Se uma das posições de câmera usadas no *replay* fosse usada para o jogo, traria uma grande dificuldade para o jogador, além de diminuir a imersão, visto que não se tem uma boa noção do que há à volta do carro, imprescindível para um piloto de carros.

Figura 2: Gran Turismo 3; tela de jogo à esquerda e replay *da corrida à direita.*

Capítulo 3

Trabalhos Relacionados

Desde o surgimento da cinematografia, no século XIX, diversos autores vêm estudando novas técnicas para deixar as produções mais atraentes para o telespectador, e as escolas de cinematografia também se multiplicaram. Hoje existem boas publicações sobre as técnicas cinematográficas, como o trabalho de Martin [1985], além de *websites* especializados, como o Mnemocine [2008]. Mais recentemente, as mesmas técnicas vêm sendo amplamente adotadas pelos jogos, para aumentar a imersão do jogador no ambiente 3D. Além disso, começou-se a pensar na possibilidade de se exibir uma partida de um jogo 3D exatamente como nas transmissões esportivas. O trabalho de Drucker [1994] analisa a viabilidade dessa aplicação das técnicas cinematográficas, mostrando um panorama muito positivo.

Existem diversos trabalhos, tais como o de He et al. [1996] e Amerson & Kime [2001], que visam facilitar o problema de posicionar a câmera no ambiente virtual. Alguns deles utilizam uma DSL (*Domain-Specific Language* – Linguagem Específica de Domínio), que são extremamente úteis por posicionar a câmera através de comandos de mais alto nível do que as linguagens de programação convencionais, porém têm a desvantagem de deixar o posicionamento das câmeras predefinido; apenas ativam uma câmera ou outra dependendo dos eventos que ocorrem no ambiente.

Outros trabalhos, como os de Drucker [1994] e Hermann & Celes [2005], posicionam a câmera de forma mais automática, sem a intervenção do usuário. Porém estes métodos oferecem um resultado cinematográfico mais pobre se comparado ao uso de DSLs. O sistema de Hermann e Celes [2005] ainda divide o trabalho de posicionar a câmera em módulos roteirista, diretor e cinegrafista. Porém, com o *framework* utilizado neste trabalho, o XNA, o módulo cinegrafista proposto por eles se confunde com o próprio mecanismo de geração de imagens. Com isso, o módulo diretor pode dar a sua saída de dados diretamente para o *pipeline* gráfico da aplicação.

Na linha de reconhecimento de ações no ambiente virtual, Pinhanez [1999] propõe uma representação formal para ações no contexto de sistemas interativos envolvendo atores reais e virtuais. Essa representaçao baseia-se na decomposição em sub-ações elementares, sendo sensível a contexto, com as possíveis relações temporais entre as sub-ações e a ação principal representadas por uma rede de restrição com três possíveis valores: passado, presente e futuro (do inglês PNF, *past-now-future*). O reconhecimento de ações dentro dos ambientes virtuais é feito através da propagação do estado temporal dessas redes PNF levando-se em consideração o estado de sensores elementares. Porém, em seu trabalho não foi feito nenhum experimento em sistemas que incluam motores de simulação física, que podem incluir uma maior riqueza de informações ou previsão de eventos antes que os mesmos ocorram. Nesses ambientes, uma colisão pode ser prevista em função da velocidade e direção do deslocamento de um objeto, sua distância para um obstáculo e seu coeficiente máximo de desvio ou atrito.

A plataforma XNA, lançada pela Microsoft em 2006, se tornou muito popular no meio acadêmico e comercial, focado para produção de jogos para Windows e X-box 360[1]. Dentro do público-alvo do XNA encontram-se estudantes e desenvolvedores independentes. Sua documentação é repleta de exemplos e conta com explicações sobre os fundamentos de processamento gráfico e técnicas de programação. Com o seu crescimento, surgiram diversos *websites* especializados na ferramenta,

[1] Porém o SDK profissional não está disponível para estudantes ou desenvolvedores independentes.

oferecendo desde tutoriais a classes e bibliotecas prontas. Existem hoje diversos trabalhos demonstrando diferentes técnicas para implementação de câmeras em XNA. Entre eles destacam-se Riemers [2008] e Fegelein [2008], por explicar todos os fundamentos matemáticos em se tratando de manipulação de câmeras em ambientes 3D. Também surgiram muito livros sobre a ferramenta, tais como a publicação de Nitschke [2007] e a de Carter [2007].

Este trabalho apresenta uma aplicação capaz de reconhecer as ações do ambiente e posicionar a câmera de tal forma que sejam considerados os eventos do cenário virtual e a previsão dos próximos acontecimentos, através de cálculos físicos e o aproveitamento de informações passadas pelo motor do jogo.

Capítulo 4

Sistema de Previsão da Simulação

Neste capítulo são apresentadas as principais técnicas que dão apoio à funcionalidade de posicionar câmeras dinâmicamente no ambiente virtual. São elas o reconhecimento de ações, a previsão da simulação física e a previsão da máquina de estados.

4.1 Reconhecimento de Ações

Um ambiente virtual geralmente é formado de objetos que são capazes de executar determinadas ações dentro do ambiente, sejam essas ações mais simples, sejam elas capazes de mudar drasticamente o próprio ambiente e afetar outros objetos. Para a cinematografia, mesmo objetos que ficam parados têm um significado dentro do ambiente e vão influenciar a cena de alguma forma. Por isso é necessário um mecanismo para reconhecer o que cada objeto está fazendo e que tipo de influência um dado conjunto de objetos representa para sua vizinhança ou para o ambiente virtual como um todo. Resumidamente falando, reconhecer as ações de um ambiente virtual significa identificar corretamente o que está acontecendo num dado momento dentro deste ambiente virtual.

Ações mais objetivas ou pontuais fazem parte do próprio motor do jogo ou fazem parte das propriedades ou máquina de estados de alguma classe. Neste caso, saber que uma ação ocorreu torna-se uma tarefa trivial, visto que pode haver um marcador ou uma variável indicando que a mesma ocorreu ou está ocorrendo. Abaixo estão alguns exemplos de ações objetivas em diversos tipos de jogos:

- Início de fase;
- Personagem começa a andar;
- Carro colide com uma parede;
- Inimigo morre;
- Passagem de uma unidade de tempo.

Entretanto, às vezes uma modelagem de todas as ações no próprio motor do jogo ou incluída na lógica do objeto pode ser uma tarefa muito difícil, devido à subjetividade de algumas ações e eventos. Por exemplo, num jogo de futebol, não é fácil reconhecer que um jogador prefere fazer suas jogadas pelo lado direito do campo. Num jogo de corrida, o jogador pode optar por nunca ultrapassar um certo carro. As ações que dependem do comportamento do jogador são as mais difíceis de reconhecer, pois é necessário uma interpretação dos acontecimentos ao longo do tempo.

Para facilitar o reconhecimento desse tipo de evento é possível a utilização de um agente que fique monitorando as ações dos objetos e o estado do ambiente ao longo do tempo, e que avise ao sistema caso encontre um padrão de comportamento que se assemelhe com os eventos esperados ou relevantes para a construção da cena cinematográfica, culminando com o posicionamento da câmera e seus próximos movimentos. O termo “agente” descreve uma idéia da Engenharia de Software referente a uma entidade capaz de agir com um certo grau de autonomia para completar tarefas [Barella et al. 2007], no caso, em intervalos de tempo regulares, observando o que acontece no ambiente.

4.1.1 Funcionamento do agente de reconhecimento de ações

O agente de reconhecimento de ações fica monitorando as ações dos objetos e o estado do ambiente ao longo do tempo, e avisa ao sistema quando encontra um padrão de comportamento que se assemelha com os eventos esperados ou relevantes para a construção da cena cinematográfica. A figura 3 mostra a estrutura de dados utilizada por este agente.

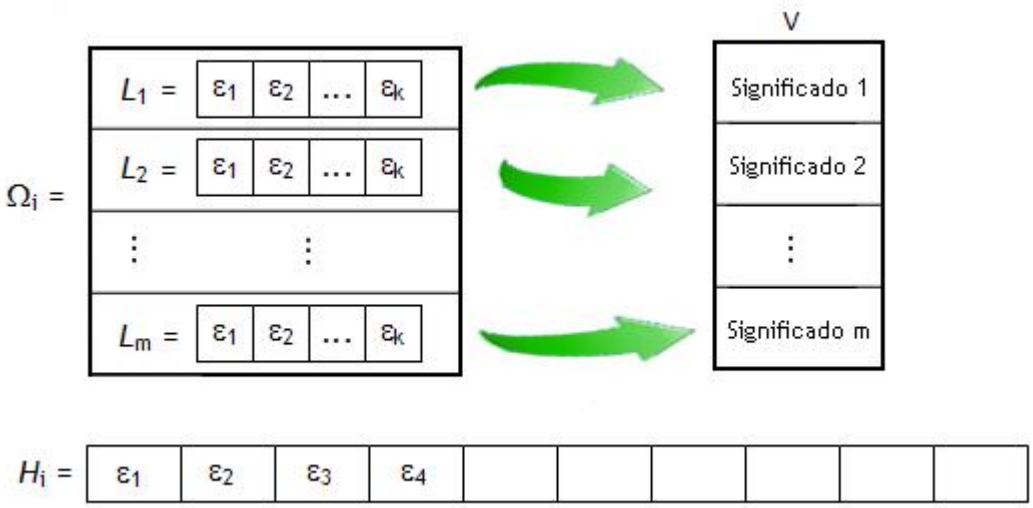

Figura 3: Estruturas de dados utilizada para o reconhecimento de ações de um objeto o_i.

Em primeiro lugar, em tempo de *design*, deve-se definir que conjunto de objetos $O = [o_1, o_2, \ldots, o_n]$ serão monitorados e quantos eventos passados serão considerados. Então, a cada *game-loop* o estado de cada objeto $o_i \in O$ será armazenado em um lista H_i contendo o histórico dos seus estados, uma lista H_i para cada objeto o_i. Alternativamente, o estado atual de um objeto $o_i \in O$ pode ser armazenado no histórico H_i à medida que novos eventos ocorrem, ou em intervalos regulares de *game-loops*. Além disso, conforme ilustrado na figura 3, ao invés de armazenar estados de objetos, a lista H_i pode ser prenchida com conjuntos de sentenças *booleanas* envolvendo as variáveis relevantes para a montagem da cena, também definidas em tempo de *design*.

A lista H_i é então comparada com um conjunto $\Omega_i = [L_1, L_2, \ldots, L_m]$ de comportamentos ou ações L, esperadas para aquele objeto o_i, definidos em tempo de

design. Cada entrada desse conjunto Ω_i é uma lista $L = [\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_k]$, contendo uma seqüência de estados para o objeto ou sentenças *booleanas* envolvendo todos os objetos. Uma lista L_x é semelhante a H_i, e representa uma ação ou condição com significado cinematográfico relevante. O histórico H_i vai sendo comparado com todas as m entradas L_x do conjunto Ω_i, até que se chegue à conclusão de que os acontecimentos passados se tratem de um evento relevante para filmagem, ou não. A comparação entre os elementos da lista L_x e do histórico H_i é feita avaliando a similaridade entre seus elementos. Se as duas listas são suficientemente semelhantes, então a ação L_x é dada como reconhecida, e uma referência a L_x será incluída na lista de ações reconhecidas R.

Se os eventos passados, representados por H_i, assemelham-se à uma ação esperada L_x, então representam uma ação relevante para se filmar, conseqüentemente possuindo um significado cinematográfico, como perigo, tranqüilidade, cautela, expectativa etc. Cada ação esperada L_x possui um correspondente x em um vetor V, que armazena os significados cinematográficos das ações contidas em Ω_i. Cada posição $x \in V$ guarda o significado cinematográfico da ação L_x. Isto influenciará as decisões do agente diretor quando este começa a operar posteriormente. A verificação é feita facilmente apenas relacionando os índices do conjunto Ω_i e do vetor V.

Alternativamente, o algoritmo para reconhecimento de ações pode ser executado em intervalos regulares de *game-loops* ou quando um novo evento acontece. A escolha de qual abordagem usar depende do tipo de mecânica de jogo e do quanto de trabalho o sistema pode suportar. Se, em uma fase de testes, o sistema se sobrecarrega de trabalho, uma solução pode ser a diminuição da freqüência de trabalho do agente.

4.2 Previsão da Simulação Física

A previsão física é uma técnica que ajuda a descobrir o que vai acontecer nos próximos intantes. Quase sempre se refere a movimentos e atualização de posições, visto que esses são o principal objeto das simulações físicas. Como no ambiente virtual um certo objeto pode estar sujeito a várias influências, a previsão ajuda a determinar a probabilidade com que um dado acontecimento possa vir a ocorrer ou como um dado acontecimento pode se concluir. A previsão retorna um valor em função da probabilidade de um evento ocorrer.

A previsão dos acontecimentos do ambiente virtual pode ser usado para ajudar no reconhecimento de ações subjetivas, e é especialmente útil para posicionamento automático de câmeras. Por exemplo, em um ambiente virtual onde uma motocicleta se dirige para uma rampa, é possível, através de cálculos cinemáticos, descobrir onde e como ela vai cair, tomando assim as devidas providências para mostrar o fato da melhor forma possível, antes mesmo da motocicleta saltar. Logicamente, por meio da previsão, seria possível executar outros processos tais como fazer o piloto abandonar a motocicleta caso o salto se mostre desastroso. Previsão física também é útil para fins de melhoria da inteligência artificial. Por exemplo, na lógica de um lutador inteligente que calcula seu próximo ataque de acordo com o tempo restante para que seu inimigo aterrisse depois de um salto.

Por exemplo, em uma cena com um tanque de guerra A que atira um míssil contra outro tanque B, que está a 500 metros (ou qualquer unidade de medida) do tanque A, em um ambiente 3D com física que ignora a resistência do ar, conforme a figura 4, a previsão de sucesso do tanque A pode ser calculada. O míssil foi lançado com velocidade v_0 de 100 metros/*loop* e angulação Φ de 30 graus, e sofre ação da gravidade g de 10 metros/*loop*. Através da aplicação de fórmulas de cinemática é possível determinar onde o projétil irá cair (alcance A) e quanto tempo (Δt) ele demora para percorrer a trajetória [Taveira et al. 2009]:

$$A = \frac{v_0^2 \operatorname{sen}(2\phi)}{g} \qquad \Delta t = \frac{2\, v_0 \operatorname{sen}\phi}{g}$$

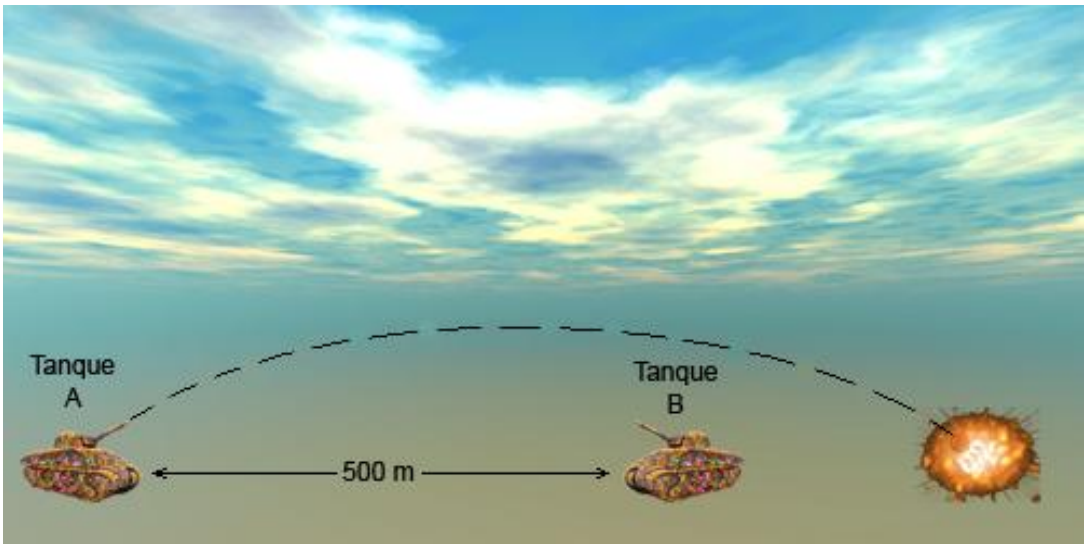

Figura 4: Previsão do resultado do tiro do tanque A.

Aplicando as fórmulas, obtém-se que o míssil vai cair a 809 metros do tanque A, nove *game-loops* depois do lançamento, concluindo-se que o tanque B não será atingido. Neste caso, o sistema já pode se preparar para filmar a ação de uma forma interessante para o telespectador.

Para se chegar a uma previsão confiável, é preciso tomar cuidado com alguns empecilhos. Sempre é preciso considerar o que o objeto tende a fazer, como forças de gravidade e outras forças aplicadas ao objeto. Os outros objetos que estão à sua volta também devem ser considerados, principalmente para saber a distância do objeto em questão e como essa distância vai variar nos próximos instantes. A disposição dos objetos em uma área é uma importante questão para a cinematografia. Por fim, deve-se considerar como o usuário ou outros componentes podem intervir na cena. Isso pode, em alguns casos, baixar muito a previsibilidade das ações, fazendo a câmera muitas das vezes seguir um comportamento mais genérico.

Devido à complexidade computacional de alguns cálculos físicos ou para se obter informações de muitos objetos, pode-se utilizar outras estruturas de dados usadas no

sistema, que já foram calculadas para outros fins ou que são calculadas em intervalos regulares. Por exemplo, é muito rápido obter as posições de objetos num espaço discretizado, quando esses objetos são representados por uma matriz. Neste caso, basta calcular a posição de um objeto no ambiente através da posição que está armazenado na matriz.

4.3 Previsão da Máquina de Estados

A previsão através de máquina de estados é feita para os casos em que algum evento com significado cinematográfico relevante possa ser alcançado pela seqüência de estados por que o objeto passa. Por exemplo, um jogador de futebol pode sempre ficar cansado quando corre intensamente por um longo tempo. Então, pode-se programar o agente diretor para que observe que, quando o jogador corre por um certo tempo ele ficará cansado, fazendo com que a câmera possa filmar o fato precisamente.

A implementação da previsão pela máquina de estados pode envolver o estudo de grafos e suas teorias, por isso não faz parte do escopo deste trabalho. É, porém, uma forma a mais de melhorar a experiência do telespectador quando acompanha uma partida de um *game*.

4.4 Integração com Motores de Física

Um motor de física é uma parte do programa que computa como objetos 3D devem se mover e interagir uns com os outros, no âmbito da física como é conhecida, usando variáveis como massa, velocidade, fricção e resistência do ar, tornando possível simular condições da vida real.

Aplicações 3D de tempo real podem ter sua própria implementação da física, ou usar a interface oferecida por alguns motores de física [Erleben 2002]. Muitas delas

possuem bons recursos, escalabilidade e estabilidade, o que popularizou sua utilização.

Objetos em um motor de física podem ser representados de diferentes maneiras. É comum que eles sejam representados através de primitivas como caixas, esferas ou cilindros. Objetos podem também ser representados como malhas de polígonos, que podem se combinar para constituir qualquer forma. Neste último caso porém, são necessários cálculos mais complexos para a obtenção dos resultados de simulação. Em aplicações 3D em tempo real com um modelo físico de câmera, os seus parâmetros físicos também podem ser tratados pelo motor de física. Isso garantirá que a câmera não atravesse objetos sólidos como paredes. Porém, a câmera não é afetada por forças gravitacionais e, quando acontece uma colisão, ela pode ou não balançar, dependendo da aplicação. Naturalmente o motor deve ter o conhecimento de todos esses atributos para tratá-la corretamente.

Uma simulação física pode ser dividida em duas fases mais importantes: detecção de colisão e simulação da dinâmica, conforme mostrado na figura 5. Toda a computação do motor começa quando a aplicação passa os parâmetros físicos dos objetos, e termina quando os atributos dos objetos (posição, velocidade etc.) são atualizados.

As fases de simulação da dinâmica e detecção de colisão são ainda divididas em outros módulos. O módulo controlador de tempo é a parte central do motor. Este módulo recebe todos os parâmetros da aplicação e controla quando e como os outros módulos devem ser ativados. Normalmente o controlador de tempo funciona em sincronia com o relógio da aplicação, e sabe o seu tempo máximo disponível para computação. Já o módulo de detecção de colisão, por questões de eficiência, faz primeiro uma seleção dos objetos candidatos a colisão (etapa denominada de fase de Aproximação), e depois faz testes mais detalhados com esses objetos selecionados (denominada de fase de refinamento). Como resultado da colisão é passado ao controlador de tempo uma lista dos contatos que estão acontecendo, e o controlador de tempo pode avisar à aplicação que as colisões aconteceram, e ativar o computador

de movimento, que fica encarregado de atualizar os parâmetros dos objetos de acordo com as forças de restrição retornadas pelo módulo computador de restrição [Seugling & Rölin 2006].

Figura 5: Funcionamento de um motor de física de corpos rígidos.

Capítulo 5

Solução para Movimentação de Câmeras

Neste capítulo são apresentadas as principais técnicas para posicionar e movimentar a câmera no ambiente. A partir das ações que foram reconhecidas e pelas inferências obtidas através da simulação física, como descrito no capítulo anterior, é possível posicionar a câmera e fazê-la se comportar segundo os princípios de cinematografia. Todos os dados obtidos nas etapas anteriores serão processados por um agente diretor, que possui seu próprio estilo de filmagem. Como resultado o agente diretor passa comandos de posicionamento e movimentação para a classe câmera, que posteriormente passa ao shader da aplicação a visualização a ser renderizada.

5.1 Parte Técnica e Parte Artística

O principal objetivo é posicionar e mover a câmera de acordo com os princípios de cinematografia, ou seja, gerando uma imagem da cena coerente com o tipo de ação e transmitindo a emoção esperada.

O agente diretor tem definido em tempo de *design* suas formas de filmar. Isso significa que ele possui seu estilo próprio de filmar perigo ou de filmar tranqüilidade, por exemplo, e tentará na maioria das vezes seguir seus princípios. Se o diretor

entende que existe perigo, ele vai tomar algumas decisões específicas, como filmar sempre de perto e diminuir o tempo das tomadas. Seu comportamento precisa ser definido no seu código, e é dependente do tipo de jogo. Num jogo de beisebol, por exemplo, existiriam diferentes configurações de câmera definidas para o diretor, para cada uma das emoções que se quisesse passar.

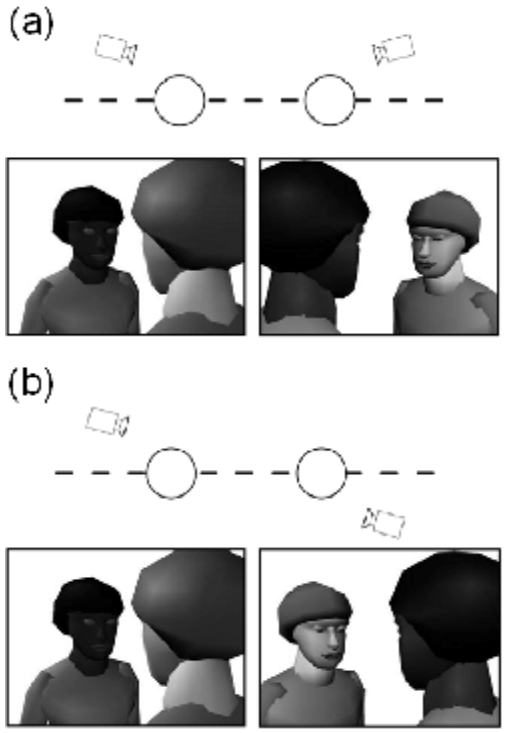

Figura 6: Cena com consistência espacial (a) e cena inconsistente (b).

Um conceito muito importante na construção de cenas é a linha de ação [Martin 1985], que pode ser melhor entendido como um plano que divide o espaço da cena. A figura 6 mostra uma aplicação da linha de ação, sendo esta representada por uma linha tracejada, e dois atores representados por círculos, vistos de cima. Para que a cena fique consistente, é desejável que todas as tomadas de câmera sejam feitas do mesmo lado do espaço definido pela linha de ação [Hawkins 2005]. A violação desta regra nos cortes (Figura 6.b) dá a impressão que os personagens trocam de lugar durante a conversa, e pode deixar o telespectador confuso. Este problema não acontece quando não existem cortes de câmera, mas filmagens sem cortes comprometem a linguagem cinematográfica utilizada. Em um ambiente virtual, a

consistência espacial pode ser preservada posicionando a câmera sempre de um mesmo lado de um plano imaginário. Este plano pode ser definido de diversas formas. Um exemplo, como o da figura 6, seria criar um plano que corta os dois atores, e posicionar a câmera sempre de um mesmo lado.

Usando conceitos da linguagem cinematográfica, segundo Martin [1985], também existem outras técnicas para dar mais dramacidade à ação, aplicadas ao contexto de um jogo:

- Se o principal objeto da ação tem alta probabilidade de interagir com outros objetos próximos, a câmera tende a ir para um ponto em que pode enquadrar todos objetos em questão.
- Se o personagem se utiliza de uma jogada especial, ou aciona um efeito incomum, a câmera pode se posicionar de forma a filmar os alvos que o personagem almeja.
- Se o jogo mostra uma situação segura ou mais calma, a câmera pode ficar mais longe para mostrar todo o cenário.
- Em um jogo de naves ou carros, quando existem muitos obstáculos à frente ou espaços apertados, existem grandes probabilidades de colisão. Neste caso a câmera tende a seguir o objeto, alternando com posições próximas a ele. Como muitas vezes nesses ambientes as ações são muito rápidas, o tempo de cada tomada é bastante curto.
- Quando um personagem fica rodeado de muitos inimigos, existe muito perigo. Neste caso a câmera pode se movimentar de modo a mostrar cada inimigo porém sem tirar o foco do personagem, alternando com movimentos rápidos entre cada golpe ou ação.
- Quando dois personagens perigosos se encontram, existe uma expectativa para saber como os dois irão interagir. Neste caso pode-se executar o movimento de sutilmente inclinar a câmera (movimento de *roll*) para enfatizar o perigo e a inquietação que o encontro representa[Hawkins 2005].

Na maioria dos jogos, são poucos os eventos que acontecem longe do objetivo principal da ação, que é o objeto sendo controlado pelo jogador, o personagem. Portanto, em quase todas as tomadas, este personagem estará visível para a câmera; mas isto não significa que o alvo será sempre fixo nele. Principalmente nas situações de perigo, a câmera pode ter como alvo um inimigo ou uma porção do ambiente onde este personagem irá chegar em poucos segundos. O alvo da câmera pode ser, em alguns casos, um objeto que sofrerá interação do personagem, muitas vezes dando uma dica ao telespectador sobre o próximo passo do personagem no jogo.

Outro fator que deve ser considerado é o tempo das tomadas. Hoje em dia, o tempo em que uma tomada dura, para cenas de ação e/ou reflexo é de 2 segundos [Hawkins 2005]. Isso ajuda a criar no telespectador a mesma sensação de tensão vivida pelo protagonista ou jogador. Quanto mais confortável e tranqüila for a situação do personagem e quanto mais afastada estiver a câmera do solo, mais longa será a tomada, indicando serenidade.

5.2 A Câmera e seus Atributos

Na arquitetura proposta neste trabalho, a câmera é controlada através de uma classe estática, ou seja, só pode haver uma instância desta classe na aplicação. Instanciar mais de um objeto da classe câmera causaria um erro de execução. Ela é definida como descrito abaixo, em linguagem C# , nos moldes do *framework* do XNA.

As principais propriedades da classe são:

```
public vector3 position;
public vector3 target;
public vector3 up;
private vector3 velocity
private vector3 accel;
private float angularVelocity;
private CameraState state;
```

- `position`. Representa o ponto onde a câmera será posicionada no ambiente virtual, em termos absolutos. A propriedade também precisa ser definida como pública, para que os métodos de renderização do XNA possam ser usados sem grandes adaptações. O tipo `vector3` é uma estrutura que define um vetor com 3 componentes: `X`, `Y` e `Z`. Apesar de representar um vetor, esta estrutura pode também representar pontos no espaço 3D.
- `target`. Representa o alvo da câmera; define o que a câmera estará focando, o ponto para onde ela aponta. Também precisa ser definida como pública, para que os métodos de renderização do XNA possam ser utilizados.
- `up`. Representa a orientação vertical da câmera, através de um vetor unitário. Para a orientação normal da câmera, o valor a ser utilizado é `(0, 1, 0)`, que é um vetor unitário apontando para cima. Para orientar a câmera como se ela estivesse virada em 90 graus no sentido horário, por exemplo, o vetor unitário a ser usado seria `(1, 0, 0)`. Esta propriedade também é utilizada pelo XNA para a renderização da cena.
- `velocity`. Representa o vetor velocidade da câmera quando está em movimento. Quando a câmera está se movimentando, a cada *frame*, a unidade de tempo utilizada, sua propriedade `position` será adicionada de `velocity`, atualizando a sua posição.
- `accel`. Representa o vetor aceleração para a movimentação da câmera. Quando a câmera está se movimentando, a cada *frame* sua propriedade `velocity` será adicionada de `accel`, atualizando a sua velocidade, que também alterará a propriedade `position` da câmera.
- `angularVelocity`. Representa a velocidade angular de deslocamento de `target` para quando a câmera gira em torno de seu próprio eixo. Esta propriedade é utilizada quando a câmera está fazendo os movimentos de *pan* e *pitch*.
- `state`. Esta propriedade serve para ter um melhor controle da câmera em tempo de execução. Vem de uma enumeração chamada `CameraState`. Esta enumeração contém os possíveis valores:
    - `Idle`. Quando a câmera está parada.
    - `Yaw`. Quando a câmera está fazendo o movimento de *yaw*.
    - `Pitch`. Quando a câmera está fazendo o movimento de *pitch*.

- `Roll`. Quando a câmera está fazendo o movimento de *roll*.
- `Dolly`. Quando a câmera está se movimentando em linha reta até uma posição final.
- `Path`. Quando a câmera está percorrendo um caminho predefinido.

Abaixo estão as declarações dos principais métodos da classe câmera. A maioria deles são públicos porque podem ser chamados diretamente pelo sistema, através do agente diretor. Pelo fato de que a linguagem C# não exige a declaração prévia dos métodos, como na linguagem C++, elas só estão relacionadas por questões didáticas.

```
public void Yaw(float Angle, float AngularVel);
public void Pitch(float angle, float AngularVel);
public void Roll(float angle, float AngularVel);
public void Dolly(vector3 FinalPoint, vector3 Velocity);
```

- `Yaw`: Este método coloca a câmera para o estado `Yaw`, que indica que a câmera irá fazer um movimento giratório em torno de si mesma, em seu eixo Y, alterando o valor da propriedade `target` da câmera. Toma como parâmetros um ângulo que a câmera deve girar e a velocidade angular da movimentação. Os parâmetros alteram as propriedades correspondentes da classe e outras variáveis de controle.
- `Pitch`: Este método altera o estado da câmera para `Pitch`, que indica que a câmera irá fazer um movimento giratório em torno de si mesma, em seu eixo X, alterando o valor das propriedades `target` e `up` da câmera. Toma como parâmetros um ângulo que a câmera deve girar e a velocidade angular da movimentação. Os parâmetros alteram as propriedades correspondentes da classe e outras variáveis de controle.
- `Roll`: Este método altera o estado da câmera para `Roll`, que indica que a câmera irá fazer um movimento giratório em torno de si mesma, em seu eixo Z, alterando o valor da propriedade `up` da câmera. Toma como parâmetros um ângulo que a câmera deve girar e a velocidade angular da movimentação. Os parâmetros alteram as propriedades correspondentes da classe e outras variáveis de controle.

- `Dolly`: Este método altera o estado da câmera para `Dolly`, que indica que a câmera irá fazer um movimento retilíneo até o ponto `FinalPoint`, especificado como parâmetro. A velocidade de movimentação também é especificada, o que altera a propriedade `velocity` da câmera. Este método altera somente a posição da câmera e mantém seu alvo inalterado.

5.3 O Agente Diretor

O posicionamento de câmera é resultado de todas as etapas anteriores. O motor de jogo informa os eventos relativos à lógica do jogo ou as propriedades e máquinas de estados dos objetos. Com base nessas informações, é possível que eventos subjetivos sejam reconhecidos e que os próximos eventos sejam previstos. Então, o agente diretor pode posicionar a câmera de acordo com a linguagem cinematográfica utilizada, pois ele já tem acesso a todas as informações que necessita.

O agente diretor recebe como entrada o estado do ambiente e uma referência aos principais objetos envolvidos, isto é, uma lista de ponteiros $O = [o_1, o_2, \ldots, o_n]$ para acessar as propriedades do personagem principal e as propriedades de seus inimigos ou outros objetos o_i que possuam alguma influência nos acontecimentos. Na prática, é mais fácil fazer com que o agente diretor tenha acesso direto ao estado do ambiente virtual, às classes de personagens e listas de inimigos ou outros objetos. Além disso, o agente diretor tem acesso à lista de ações reconhecidas *R* e os resultados da previsão física.

A partir daí o agente diretor começa a montar uma forma de filmar a cena. Uma árvore de decisão *A*, chamada Árvore de Eventos, é construída dinamicamente e por meio desta o resultado final *C* será alcançado. A raiz de *A* serve apenas como ponto de partida, e terá tantos filhos quanto o número de objetos na lista $O = [o_1, o_2, \ldots, o_n]$, ou seja, n_2 nós. A notação n_2 indica o número de nós no segundo nível da árvore. Feito isto, o diretor escolhe uma das folhas aleatoriamente, e começa a percorrer um caminho *C* dentro da árvore; as outras folhas são então descartadas para minimizar o

consumo de memória. Este movimento significa que aquele objeto *o* foi escolhido para ser o assunto principal da cena. Deste nó, são geradas novas folhas; o número de folhas será uma *n*-tupla igual a quantidade de ações reconhecidas *r*, mais o número de ações previstas *p*, mais o número de eventos ocorridos recentemente *u*. Ou seja, número de folhas no terceiro nível da árvore será:

$$n_3 = r + p + u$$

Mais uma vez o diretor escolhe uma folha aleatoriamente; desta vez a folha escolhida indicará a ação a ser filmada. A escolha pode ser um evento que acabou de ocorrer, uma ação reconhecida ou uma ação que poderá se concretizar em poucos instantes. Cada uma possui seu valor cinematográfico correspondente, que foi definido em tempo de *design*.

Por fim, o agente diretor cria seu último conjunto de folhas na árvore de cena. Desta vez o número de folhas será igual ao número de objetos envolvidos na ação escolhida no passo anterior. A escolha aleatória do diretor definirá quais objetos serão enquadrados e como a câmera será posicionada de fato, seguindo os conceitos da linguagem cinematográfica definida em tempo de *design*.

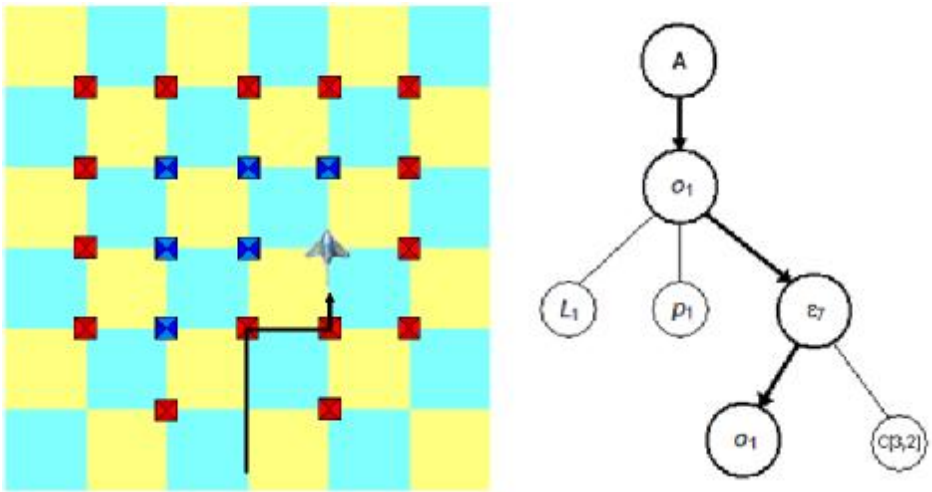

Figura 7: Exemplo de funcionamento da Árvore de Eventos.Os nós marcados representam o caminho escolhido pelo diretor.

Como ilustrado acima, na figura 7, no jogo implementado, a nave está tentando pegar os cristais azuis, necessários, de uma área totalmente envolvida por cristais

vermelhos, letais. A cena será construída segundo a Árvore de Eventos. No segundo nível da árvore (o primeiro é a raiz) existirá apenas um nó, pois a nave (objeto o_1) é o único objeto sendo monitorado. No terceiro nível, se o último evento ocorrido foi a coleta de um cristal azul (evento objetivo ε_7), existe apenas a ação reconhecida de que o jogador está tentando coletar os cristais em uma certa ordem para ganhar mais pontos (evento subjetivo L_1), e foi previsto apenas que há uma alta chance de colisão à frente com um cristal vermelho (evento previsto p_1), a Árvore de Eventos terá 3 nós neste nível, cada um representando uma situação. O diretor escolherá uma dessas folhas aleatoriamente para continuar. Escolhendo o caminho ε_7, são criadas novas folhas para este nó, cada uma representando um objeto envolvido no evento: a nave e o cristal recém-coletado (representado como C[3,2], pois cristais estão dispostos em uma matriz C). Cada folha deste nível possui parâmetros como posição e velocidade dos objetos e vai interferir na forma de posicionar a câmera, atuando como um modificador local.

O caminho percorrido na árvore para uma decisão é armazenado, pois o evento escolhido anteriormente precisará ser consultado de novo para que seu significado cinematográfico seja lido. Daí a importância do conhecimento de cinematografia no desenvolvimento desse tipo de aplicação. Com base nessa informação, o tempo mínimo da tomada pode ser determinado, e podem ser aplicados determinados movimentos de câmera, bem como definir a posição da câmera em relação ao alvo determinado.

Após isso, a cena é renderizada, e o tempo da tomada começa a ser contado. Esgotado este tempo, o agente diretor volta a esperar novos eventos para repetir o processo. Dependendo do tipo de ambiente e do tipo de controle que o jogador tenha sobre ele, isto pode acontecer mais rapidamente ou lentamente. Como esta abordagem é suscetível a problemas, principalmente devido à imprevisibildade dos comandos do usuário, é interessante fazer com que o motor dispare um evento de tempos em tempos, como um marcador de tempo da fase ou de jogo, para forçar o diretor a redefinir sua forma de filmar a cena.

O fator de aleatoriedade na forma de decisão do diretor se deve a dois motivos: O primeiro é fazer com que o processo não se torne totalmente determinístico. O segundo é devido à limitação de se trabalhar em tempo real. Fazer o diretor avaliar cada evento e cada objeto poderia criar um esforço computacional muito maior do que o jogo em si, e também abordar assuntos teóricos que ficam fora do escopo deste trabalho. A escolha aleatória tem a vantagem de possibilitar que um mesmo *replay*, a cada vez que fosse visto, tenha uma configuração de câmera diferente. Para a transmissão em tempo real, cada telespectador poderia receber uma imagem filmada de um ângulo diferente.

As informações mais importantes para o agente diretor são a posição e o tamanho dos objetos. Com isso ele pode posicionar a câmera, definir o alvo e enquadrar os objetos, como será explicado na próxima seção.

5.4 Posicionamento

Ao posicionar a câmera para filmar uma cena, o agente diretor precisa lidar com as particularidades do ambiente virtual, tais como o posicionamento e tamanho dos objetos envolvidos na filmagem. Isso acontece logo após o diretor escolher uma forma de filmar um certo evento, consultando o significado que aquele evento tem para a narrativa em sua totalidade. Algumas questões envolvento o posicionamento da câmera é explicado a seguir, segundo Hawkins [2005].

5.4.1 Enquadramento

A forma mais tradicional de enquadrar objetos na tela é através da projeção de seus pontos na tela, acessando sua matriz de projeção. Para saber se um objeto está totalmente contido na tela basta projetar alguns pontos extremos do *bounding volume* do objeto. Esta abordagem não é perfeita, mas útil na maioria dos casos. Quando se quer enquadrar um conjunto de objetos, basta fazer o teste da projeção

com todos os objetos envolventes individualmente, ou aglomerar todos os volumes dos objetos para formar um único grande volume.

Com o *framework* utilizado, é possível definir um ponto de um objeto como o alvo da câmera, para fazê-la apontar para o objeto automaticamente, independente de sua posição, apenas definindo a propriedade `target` da câmera. Portanto, é possível definir uma posição e alvo da câmera, e testar se os objetos importantes para a cena estão bem enquadrados. No entanto, enquadrar o objeto de forma que ele fique grande ou pequeno na cena tem seu impacto para o telespectador, e está definido no evento ocorrido para filmagem. Métodos tradicionais consistem em calcular a razão entre a área da tela e a área da projeção do *bounding volume* do objeto focado [Hawkins 2005]. A relação entre a área da projeção do bounding volume do objeto A_{bvol} e a área da tela A_t é definido da forma:

$$\frac{A_{bvol}}{A_t}$$

O agente diretor tem predefinido a proporção que cada elemento deve ter na tela, dada uma situação para filmar, definido no significado do evento escolhido. Com isso ele posiciona a câmera num ponto relativo ao alvo e começa a testar pelo enquadramento dos objetos. Se o objeto está grande ou pequeno, pode-se afastar ou aproximar a câmera, respectivamente. Se o objeto deve aparecer grande, a câmera se aproximará do alvo em linha reta até que esta restrição seja satisfeita. De maneira simétrica o diretor trabalha para objetos que devem aparecer pequenos. Da mesma forma, para algum objeto que está mal-enquadrado na tela, a câmera pode se mover para a frente ou para trás, ou mudar o alvo para a esquerda ou direita, dependendo do significado cinematográfico que o evento possui.

5.4.2 Oclusão

Em ambientes virtuais, pode ser muito difícil detectar quando um objeto está bloqueando a visão. Métodos tradicionais consistem em traçar raios até o alvo da

câmera e avaliar se estes raios interceptam algum outro objeto que está entre a câmera e o alvo [Hawkins 2005]. Caso a oclusão seja detectada, deve-se mudar a posição da câmera para um outro ponto onde não haja mais oclusão.

Uma melhor saída para evitar este problema é se aproveitar do conhecimento prévio do ambiente virtual e do funcionamento do jogo para não posicionar a câmera em pontos onde objetos podem entrar e impedir a visão. Por exemplo, se o ambiente não tem teto e é possível determinar a altura máxima que um objeto pode chegar, sabendo-se que nenhum objeto pode ficar em cima do outro, pode-se posicionar a câmera acima daquela altura sem se preocupar com oclusão. Isto pode ser codificado diretamente no agente diretor ou usar uma estrutura de dados para deixar disponível para consulta os locais onde a câmera pode ser melhor posicionada ou onde a câmera não deve ser posicionada.

5.4.3 Consistência espacial

O problema da boa consistência espacial existe principalmente em função da oclusão de objetos. Se a melhor forma de filmar uma cena consiste em posicionar a câmera atrás de uma parede, pode-se tornar aquela parede temporariamente invisível, enquanto a tomada está sendo feita [Hawkins 2005]. A figura 8 mostra esta situação.

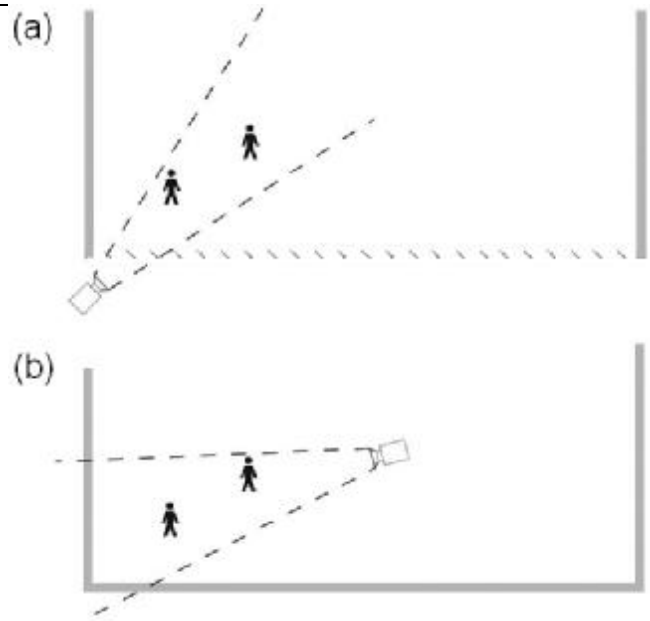

Figura 8: Filmagem através da parede sem perder a consistência espacial.

Alternativamente, um efeito bem aceito nos *games,* quando um objeto grande, como uma árvore, fica na frente do ator principal, é tornar o objeto semitransparente. Neste caso, tornar o objeto totalmente invisível causará confusão no telespectador, e pode causar o efeito indesejável de fazer o objeto aparecer de repente quando este sai da frente do ator. Deixando o objeto semitransparente informa-se ao espectador que existe um objeto na frente, mas ainda assim ele não perde detalhes da tomada.

5.5 Movimentação

O agente diretor também é o responsável por cuidar da movimentação da câmera. Ao ler o significado cinematográfico do evento a ser filmado, ele altera o estado da classe câmera para indicar que há algum tipo de movimento ou não, através dos métodos públicos da câmera. A maioria dos comandos de movimentação de câmera tomam como parâmetro um ângulo e velocidade de deslocamento. Esses comandos alteram o estado da câmera, que fará a câmera se comportar corretamente em seu método `Update`. O tempo que a câmera se manterá neste estado será sincronizado com o tempo de tomada definido pelo agente diretor, por isso a câmera concluirá seu

movimento no exato momento que o tempo de tomada se esgotará. Isto é feito através de cálculos simples, envolvendo o tempo de tomada, o ângulo e a velocidade de deslocamento.

Capítulo 6

Implementação e Resultados

Neste capítulo são apresentados o experimento usado para validar a técnica, os conceitos particulares da ferramenta de desenvolvimento e os resultados obtidos.

6.1 Renderização no XNA

Utilizando-se a ferramenta XNA, para renderizar uma cena é preciso uma coleção de objetos 3D que compõem a cena e um *shader* que é o responsável por desenhá-la. Este *shader* toma uma configuração de câmera como parâmetro, e através de um *loop* renderiza as malhas uma a uma. Normalmente, para desenhar a cena corretamente, todo objeto 3D deve ser passado ao *shader* usando a mesma configuração de câmera. Uma câmera é composta por 2 elementos principais: as matrizes 4x4 conhecidas como *view* e *projection*. Ambas são processadas pelo *shader* na GPU, o que torna o processo extremamente rápido. A matriz *view* estabelece a orientação da câmera; ela contém informações como sua posição, definida por um ponto no espaço, o alvo, que é o ponto para onde a câmera aponta, e o vetor que representa a orientação vertical da câmera. A matriz *projection* estabelece como a câmera projeta sua visão para a tela; e contém informações como a abertura focal, a relação entre altura e largura da

janela de exibição, e os limites (a distância) do que a câmera pode enxergar, conhecidos como *nearplane* e *farplane*. Será efetivamente exibido na tela tudo o que estiver dentro do volume definido pela matriz *projection*, conforme a Figura 9.

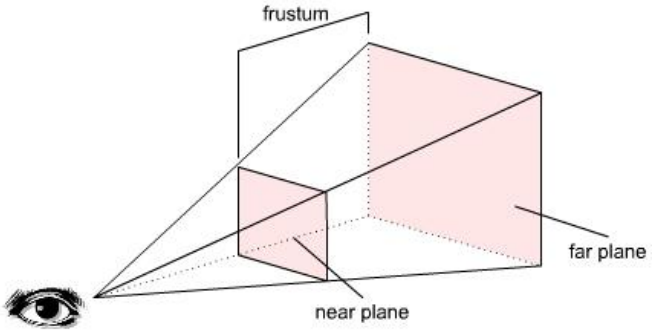

Figura 9: Frustum de visão.

É possível adicionar outros atributos à câmera transformando-a em uma classe. Para que a câmera tenha propriedades físicas, basta adicionar membros como velocidade, massa e aceleração como atributos da classe, exatamente como explicado no capítulo 5.2. Também podem-se adicionar propriedades e métodos que facilitem seu uso, assim a manipulação de camera será mais eficiente e versátil [Fegelein 2008].

O XNA fornece um método muito útil para manipulação de vetores que pode ser utilizado para a manipulação de câmeras, `Vector3.Transform()`. Ele permite que um vetor seja facilmente rotacionado no espaço de acordo com o quatérnio ou matriz (de rotação) passada como parâmetro. Com este método a implementação de câmeras em primeira pessoa ou de câmeras que tentam manter uma distância fixa dos seus alvos é facilitada. O uso de quatérnios é recomendado quando é preciso armazenar rotações relativas ao objeto; são muito mais eficientes que armazenar rotações alinhadas com os eixos cartesianos [Riemers 2008]. Também são muito úteis para implementar movimentos relativos da câmera, tais como *yaw*, *pitch* e *roll*. Esses movimentos são originários da aviação mas que são muito usados em cinematografia, na movimentação de câmeras. A Figura 10 ilustra o significado de cada um desses movimentos.

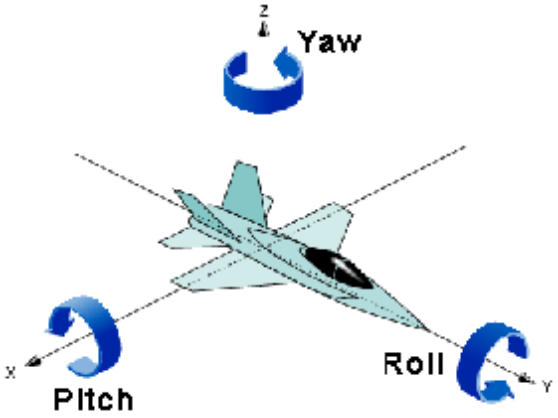

Figura 10: Yaw, pitch e roll.

Para usar a câmera durante a programação do jogo, depois de todos os cálculos da lógica da classe câmera, é preciso informar ao shader as matrizes *view* e *projection*. O XNA fornece formas para agilizar o cálculo das matrizes *view* e *projection*, com os métodos `Matrix.CreateLookAt()` e `Matrix.CreatePerspectiveFieldOfView()`, respectivamente.

6.2 Projeto Crystal Arena

Para ilustrar o uso da técnica de posicionamento de câmeras aproveitando resultados da simulação física do ambiente, foi produzido um jogo em XNA, chamado Crystal Arena. Este jogo consiste em controlar uma espaçonave em um ambiente 3D compreendido por um terreno fechado e vários itens e obstáculos, compondo um labirinto. O objetivo é coletar todos os cristais azuis para abrir a saída e passar para a próxima fase antes do tempo se esgotar. Ao mesmo tempo o jogador precisa lidar com cristais vermelhos que devem ser evitados e tesouros que garantem mais pontos. A figura 11 mostra uma tela de jogo.

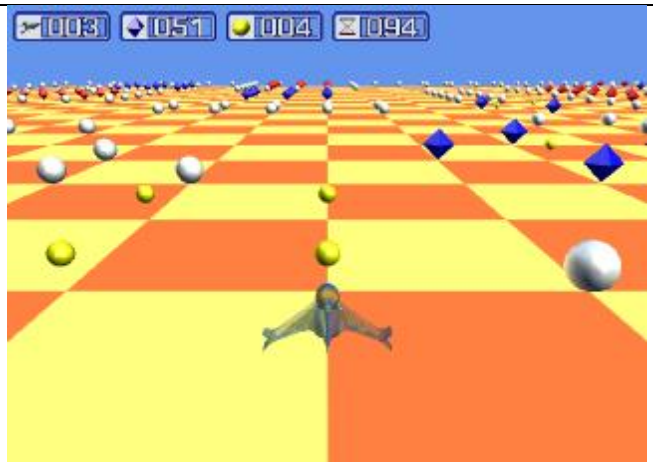

Figura 11: O jogo Crystal Arena.

Em um fase de jogo, o terreno é quadriculado, e a nave só pode se mover pelas divisões de cada quadrado, ou seja, as linhas que eles formam. Além disso os itens são posicionados nos pontos onde as linhas se cruzam. Isso permite facilmente modelar a fase através de uma matriz 32x32 contendo todos os elementos (cristais azuis, cristais vermelhos, tesouros e bolas brancas). Os valores possíveis para cada posição da matriz são:

- 0: espaço vazio. Representa um local onde não há nada; por estes locais a nave pode se locomover sem que nenhum evento aconteça.
- 1: bola branca. Representam os limites e as paredes do terreno. Exceto pelas bolas que representam a entrada e saída da fase, o valor permanece inalterado com o tempo.
- 2: cristal vermelho. São os cristais que devem ser evitados. Geralmente permanecem inalteradas com o tempo, mas quando na condição de manobra especial, podem se transformar em tesouro.
- 3: cristal azul. São os cristais que devem ser coletados; quando isso acontece se transformam em cristais vermelhos.
- 4: tesouro. São as pequenas bolas amarelas que dão mais pontos. Quando coletadas deixam o espaço vago, se transformam no valor zero.

O conteúdo da matriz vai sendo atualizado à medida que o jogo avança, ela representa o que existe no momento. Os objetos são criados e renderizados de acordo

com o conteúdo desta matriz. Quando um objeto se transforma ou desaparece, o conteúdo da matriz é automaticamente atualizado para manter a consistência entre os dados gráficos e os dados estruturais. Seu conteúdo só pode mudar quando a nave sofre alguma colisão, o que faz o método de gerenciamento de colisões o processo mais complexo da mecânica do jogo.

A nave não tem representação na matriz, mas sua posição é facilmente calculada. Por outro lado, para aumentar o desempenho, cada item do cenário tem a sua posição na matriz como um atributo. Quando a nave colide com um objeto, basta consultar o atributo do objeto para saber a posição da matriz que será atualizada, quando aplicável. Saber a posição da nave não é tão importante na hora do jogo, mas é fundamental para a geração de boas imagens no *replay*.

A câmera acompanha o movimento da nave, mantendo uma distância fixa, ficando sempre atrás e acima da nave. Com isso o jogador tem uma vista do terreno relativamente ampla, permitindo visualizar os objetos que estão na frente da nave, e também nas posições adjacentes dos lados. Mesmo quando a nave está fazendo a curva, a câmera mantém a mesma distância, percorrendo uma trajetória esférica com o centro da nave, se movendo 90 graus no mesmo sentido da nave.

A espaçonave se locomove como se estivesse deslizando pelo solo, tende a se mover continuamente para a frente, e, conforme a intervenção do jogador, pode ser rotacionada em 90 graus para a esquerda ou 90 graus para a direita, seguindo as linhas contidas no solo quadriculado. Sua velocidade é constante enquanto ela se move, mas enquanto ela está virando para os lados, sua velocidade de deslocamento é nula. Isto acontece para que a nave não saia da trajetória permitida enquanto está virando e para que o jogador não se confunda com o movimento da nave e o movimento simultâneo da câmera. Para as colisões com a nave, o sistema se comporta da seguinte maneira:

- Bola branca: A nave passa a se movimentar na direção contrária, sem se virar. Se a colisão é frontal, a nave passa a se deslocar para trás, na mesma velocidade.
- Cristal vermelho: A nave é destruída e o jogador perde uma vida.
- Cristal azul: A colisão com a nave os transforma em cristais vermelhos. Ao mesmo tempo o sistema atualiza o contador de cristais azuis. Quando este chega a zero, a saída da fase se abre. Quando um cristal azul é coletado, também são feitos os testes para a jogada especial.
- Tesouro: Quando coletados desaparecem e o contador de tesouros é atualizado. No fim da fase serão computados para dar pontos ao jogador.

O jogador também pode fazer uso de uma jogada especial, que consiste em, enquanto coletando cristais azuis que vão se tornando vermelhos, circundar um grupo de cristais azuis com cristais vermelhos, conforme a figura 12. Ao concluir esta manobra, todos os cristais neste conjunto se transformam em tesouros. Neste momento a matriz que representa a fase também é atualizada, localizando as posições da matriz que representam aquele grupo de cristais, e mudando seus valores para que representem agora tesouros. Os grupos de cristais que aceitam este tipo de manobra são os de tamanho 3x3, os de 4x4 e os de 5x5.

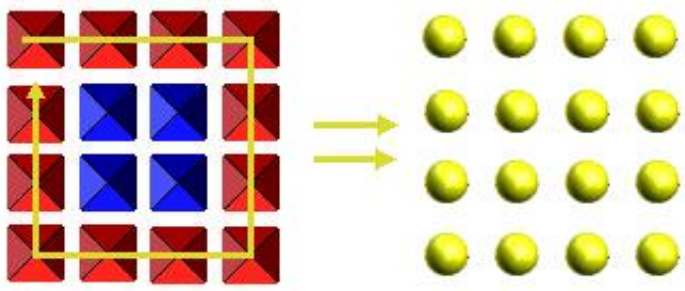

Figura 12: Transformação de cristais em tesouros.

Enquanto o jogador está controlando a nave, seus comandos vão sendo armazenados, juntamente com o tempo que o comando acontece. Quando o jogador perde uma vida ou passa de fase, existe a opção de visualizar o *replay* de sua jogada. Neste ponto a câmera pára de seguir a nave e entra no modo cinematográfico, onde os resultados da previsão física no ambiente são aproveitados para posicionar a

câmera. A nave volta à posição inicial, o cenário e todos os itens voltam ao estado inicial e a simulação recomeça, mas desta vez sem a intervenção do usuário para movimentar a nave. Agora a nave é controlada através da lista de comandos e toda a lógica de jogo é aplicada novamente, além da técnica para posicionar a câmera.

Devido à simplificação nos movimentos, a nave só pode colidir com objetos de frente ou de costas, sempre perpendicularmente, mas isso não inviabiliza os cálculos físicos e a técnica utilizada. Um jogador telespectador receberia as mesmas imagens que aquelas produzidas no *replay*. Porém, em um *replay*, os próximos estados do teclado, dos objetos ou do ambiente são conhecidos. Este fato não deve ser considerado nos algoritmos, para não prejudicar o caráter de previsão. Também, por este mesmo motivo, não deve ser feito nenhum pré-processamento para gerar ou otimizar o *replay*.

6.3 Aplicação da Técnica

Depois que o jogador passou de fase ou perdeu uma vida, ele tem a opção de visualizar o *replay* de sua jogada. É neste momento que a técnica passa a ser aplicada. Entretanto, para o domínio do jogo produzido, existem algumas adaptações do método explicado anteriormente.

6.3.1 Reconhecimento de ações

Segundo a explicação apresentada no capítulo 5, os eventos no jogo produzido também são classificados em objetivos e subjetivos. Os eventos objetivos são aqueles que fazem parte da própria lógica do jogo, através da chamada de algum método ou da alteração de algum estado ou propriedade de um objeto.

No jogo desenvolvido, os 13 eventos que fazem parte da sua lógica são:

- Início da fase, ε_1;
- Nave se move para a frente ou para trás, ε_2 e ε_3;
- Curva à direita ou à esquerda, ε_4 e ε_5;
- Impacto com bola branca, ε_6;
- Impacto com cristal azul, que se torna vermelho, ε_7;
- Impacto com cristal vermelho, que destrói a nave, ε_8;
- Coleta de tesouro, ε_9;
- Conclusão de jogada especial, ε_{10};
- Abrir a saída quando acabam os cristais azuis, ε_{11};
- Alcançar a saída da fase, ε_{12};
- Tempo esgotado, ε_{13}.

Para as ações subjetivas, existe uma lista Ω contendo cinco ações objetivas predefinidas L_x, compondo os seguintes eventos subjetivos:

- Nave começa a cercar cristais azuis para jogada especial, L_1;
- Fazer manobras complicadas em espaços apertados, L_2;
- Se aproximar da saída, L_3;
- Chegar perto das extremidades da fase, L_4;
- Nave se move em linha reta, L_5.

Para saber se a sucessão de eventos coincide com a seqüência desejada, os 20 últimos eventos são armazenados em outra lista *H*, que é atualizada toda vez que um evento acontece, e será comparada com as listas $L_x \subset \Omega$. A comparação entre as listas pode seguir qualquer heurística, mas para este experimento foi utilizada uma forma de comparação bem simples, apenas testando se uma seqüência está contida na outra. Cada evento objetivo ε_j é representado por um número, isto é, quando um evento acontece seu número *j* é adicionado à lista *H*. Outras formas de comparação e representação dos eventos podem ser empregadas, desde que isso não seja muito custoso para o sistema.

Toda vez que um evento ocorre, a lista *H* é comparada com todas as cinco entradas L_x da lista de ações subjetivas previstas, Ω. Mesmo que seja constatado semelhança com uma entrada L_x particular, todas precisam ser comparadas, o que tornará a Árvore de Eventos para o agente diretor mais rica. Além disso, para cada evento L_x dado como reconhecido existe uma variável $x \in V$ responsável por informar o valor cinematográfico que a ação representa, onde *V* é o vetor que armazena todos os significados cinematográficos para todos os tipos de eventos. Por fim, a preferência em atualizar a lista *H* na hora que acontece um novo evento é devida à rapidez com que os eventos ocorrem no ambiente. Isso diminui um pouco o trabalho do sistema em comparação com a atualização em todo *frame*.

Como exemplo para o processo, tem-se que quando a nave fizer uma curva depois de tocar 2 cristais azuis nas extremidades de uma área prenchida por outros cristais azuis, então muito provavelmente trata-se de uma jogada especial. Se o jogador falha ao coletar os cristais necessários para a jogada especial, apenas o evento subjetivo L_1, "tentativa de jogada especial", será registrado; não existirá, porém, a ocorrência do evento objetivo ε_{10}, "conclusão de jogada especial".

6.3.2 Previsão da simulação física

No jogo produzido, a previsão da simulação física é feita focada totalmente no movimento da nave, visto que ela é o objeto principal da ação e devido ao fato de nenhum outro objeto se movimentar. Da mesma forma, a maior parte das previsões é feita focada na colisão da nave com outros objetos. Mesmo a conclusão da jogada especial ou a coleta de tesouros é uma sucessão de colisões, por isso as colisões são um elemento chave da previsão.

A nave tende sempre a andar em linha reta, e como o terreno é fechado, se o jogador não intervir, pode haver uma colisão com as extremidades do cenário, se não houver outro objeto na frente. Em outras palavras, sempre há chance de colisão à frente. Outro fato importante é que a nave só pode ser manobrada quando está sobre

os cruzamentos das linhas formadas pelo solo quadriculado. De um cruzamento a outro, o usuário não tem o que fazer senão esperar que ela chegue ao próximo cruzamento. É nos cruzamentos que o jogador tem poder para virar a nave e onde a colisão com objetos pode ocorrer. Portanto, pode se dizer que, conforme a nave se aproxima de um obstáculo à frente, maior a probabilidade de colisão com este objeto. Se a nave passou pelo último cruzamento livre antes de um objeto e o jogador não mudou a direção da nave, então é certo que a colisão irá ocorrer. A probabilidade da colisão ocorrer, entretanto, diminui conforme aumenta a distância ou o número de cruzamentos livres entre a nave e o obstáculo. Com isso em vista, foi arbitrado uma constante que ajuda no cálculo da probabilidade de colisão, calibrando assim a sensibilidade do diretor. Um valor mais alto faz o diretor pensar que não haverá colisão, enquanto um valor mais baixo o faz concluir que haverá muitas chances de colisão. O valor 20 para essa constante mostrou-se bastante coerente com o que se quis mostrar.

$$\text{probabilidade de colisão} = (100 - \text{cruzamentos} \times 20)\ \%$$

Para tornar todo o processo computacionalmente mais simples, a distância em cruzamentos e a própria posição da nave é extraída da matriz que representa a fase. Devido à discretização do espaço, a posição da nave no ambiente virtual é facilmente transformada para as coordenadas da matriz da fase, e assim fazer todos os cálculos probabilísticos.

Uma outra previsão importante que é necessária para o *replay* é a conclusão de jogadas especiais. Mais precisamente a falha dessas jogadas, pois o agente diretor tenderá a filmar a jogada bem sucedida, e é preciso informá-lo que a jogada falhou para que ele mude seu comportamento. Toda vez que a nave se aproxima de uma área onde é possível executar a jogada, como mostrado na figura 13, a probabilidade de o jogador iniciar a jogada especial existe e o agente diretor se prepara para isso. Se o jogador começou a coletar os cristais azuis para a jogada especial, então a probabilidade de sucesso da jogada aumenta a cada cristal azul das extremidades do grupo que é coletado. Lendo o conteúdo da matriz da fase é possível identificar os

locais passíveis da jogada especial, mas não é tão rápido identificar se o jogador está indo bem na jogada. Por isso, é feita a contagem dos cristais necessários para concluir a jogada especial, da forma:

$$x = (nlinhas)^2 - (nlinhas - 2)^2$$

O número de cristais enfileirados numa mesma linha ou coluna é *nlinhas*. Se o jogador coletou mais de *x* cristais azuis em seqüência e a jogada especial não foi dada pelo sistema como concluída, então o jogador falhou. Assim o diretor será alertado e mudará seu comportamento.

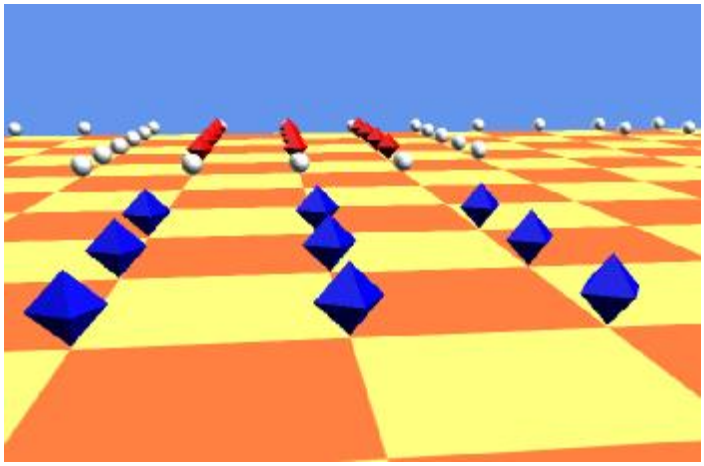

Figura 13: Conjunto de cristais passíveis de jogada especial.

6.3.3 Posicionando a câmera

A partir das ações que foram reconhecidas e pelas inferências obtidas pela manipulação física, é possível posicionar a câmera e fazê-la se comportar segundo os ensinamentos de cinematografia. O agente diretor é o responsável por manipular a câmera, com auxílio da Árvore de Eventos. Ele tem permissão para leitura e escrita das propriedades da câmera e acesso para leitura de outras variáveis de todo o sistema, como os atributos da nave e da matriz da fase.

O diretor tende a seguir algumas diretrizes de trabalho, ou seja, ele segue algumas regras de acordo com a situação atual da câmera, da nave e do ambiente. Essas regras derivam dos conceitos da linguagem cinematográfica, segundo Martin [1985]. As restrições para dar mais dramacidade à ação, aplicadas ao contexto do jogo, estão descritas a seguir:

- Se a nave está muito próxima de uma região onde é possível executar a jogada especial, a câmera tende a ir para um ponto em que pode enquadrar todos os cristais do grupo e mostrar sua transformação numa jogada bem sucedida.
- Se o usuário erra a jogada especial, a situação pode se tornar perigosa, e a câmera deve se aproximar.
- Se a nave está numa situação segura (evento reconhecido quando existem poucos cristais ou bolas brancas próximas da nave), a câmera fica mais longe e mostra todo o cenário.
- Quando existem muitos obstáculos em espaços apertados, existe muito perigo de colisão. Neste caso a câmera fica sempre perto da nave, e o tempo da tomada é bastante curto.
- Quando a nave está rodeada de muitos cristais vermelhos, existe muito perigo. Neste caso a câmera fica sempre perto da nave e o movimento de *roll* pode ser executado para enfatizar o perigo [Hawkins 2005]. O tempo das tomadas também é bastante curto.

Exceto o fato de a saída se abrir, nenhum outro evento acontece muito longe do objetivo principal da ação, que é a nave em movimento. Portanto, em quase todas as tomadas a nave passará na frente da câmera em algum momento, mas isso não significa que o alvo será sempre fixo na nave. Principalmente nas situações de perigo, a câmera pode ter como alvo uma bola branca em que existe alta probabilidade de colisão. Neste momento, provavelmente a nave irá passar, colidindo ou desviando a tempo. Outro fator que deve ser considerado é o tempo das tomadas, que dura em média 2 segundos, conforme Hawkins [2005]. Isso ajuda a criar no telespectador a mesma sensação de tensão vivida pelo protagonista ou jogador. Quanto mais

confortável for a situação da nave e quanto mais afastada estiver a câmera do solo, mais longa será a tomada, indicando tranqüilidade.

Devido ao próprio mecanismo de renderização no XNA e da arquitetura do jogo, os problemas relacionados a enquadramento e oclusão são mínimos. A renderização do XNA exige que a câmera tenha um alvo para ser filmado, indicado por um ponto no espaço. Com isso pode-se passar facilmente o ponto zero da nave ou de algum outro objeto para que este seja automaticamente enquadrado e centralizado pela câmera. Para descentralizar o objeto na tela, basta passar como alvo da câmera um ponto relativo ao ponto zero do objeto. Por exemplo, para afastar a nave para o lado esquerdo da tela, basta fazer a câmera apontar para um ponto com a coordenada X maior que a da própria nave. Isso se aplica quando a câmera está posicionada em um ponto com valor Y e Z maiores que a nave.

Para evitar o problema da oclusão, o método mais fácil é posicionar a câmera sempre acima de uma certa altura, que é a altura dos objetos no ambiente. Com a câmera num plano mais alto, ela sempre focará seu alvo sem nenhum obstáculo. Nada é feito quando a câmera precisa ficar mais baixa, porque, quando há oclusão de fato, a câmera permanecerá pouco tempo nesta posição. Uma alternativa para evitar o problema totalmente seria descobrir as posições da câmera e do alvo em coordenadas da matriz da fase, e então traçar uma reta de um ponto a outro, verificando se as coordenadas no caminho são preenchidas por algum objeto que possa obstruir a visão.

A consistência espacial é mantida posicionando a câmera de tal forma que sua componente Z no espaço seja sempre menor que a componente Z da nave. Com isso o telespectador tem uma boa noção da direção da nave e de que parte do terreno ela está sobrevoando. Também, para não gerar confusão ao telespectador, não existem cortes ou movimentos de câmera enquanto a nave está fazendo uma curva, colidindo com uma bola branca, ou está muito próximo da saída da fase. Se a câmera está se movendo nesse momento, ela pára imediatamente.

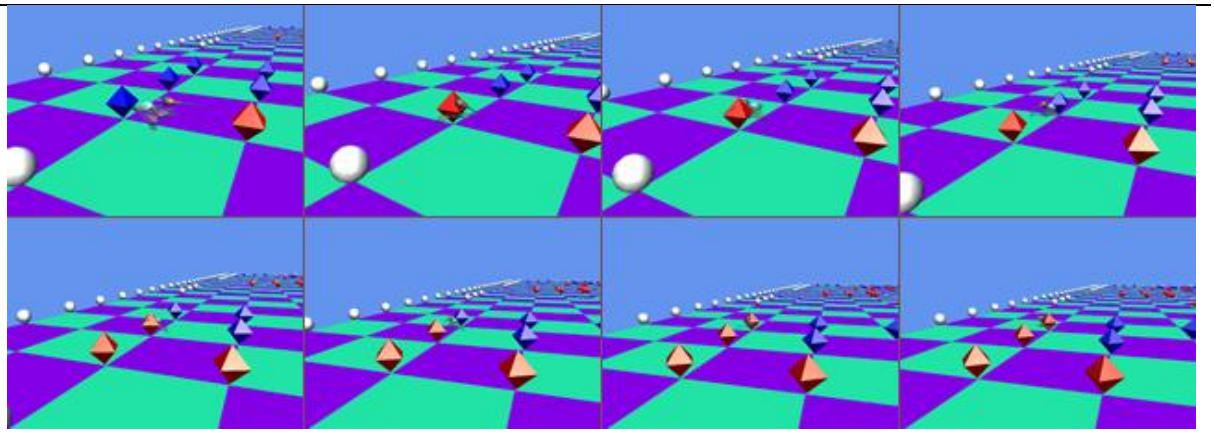

Figura 14: Posicionando a câmera mantendo consistência espacial.

6.4 Resultados

No jogo produzido, o *replay* é construído por meio de uma estrutura de dados que armazena todos os comandos do jogador ao longo do tempo, chamada lista de comandos. Uma entrada do jogador é armazenada juntamente com o número de *game-loops* passados desde o início da fase, por meio de uma variável responsável por contar o tempo de jogo, atualizada a cada frame. Esta é a única diferença para um jogo que, ao invés do *replay*, estivesse sendo transmitido em tempo real.

A abordagem da passagem de tempo por *frames* passados é mais robusta do que armazenar o tempo passado em milissegundos, pois o *replay* pode se tornar defasado em função da execução de outros processos no sistema, como a própria técnica para geração do *replay*. A execução de outros processos no sistema operacional também pode gerar alguma perturbação no *replay* se ele fosse contado em milissegundos.

Para não comprometer a intenção da previsão, apenas a posição atual da lista de comandos deve ser observada, simulando uma situação de jogo em tempo real, transmitida ao vivo. Elementos que representam as próximas entradas do jogador não serão considerados. Para a aplicação de toda a técnica, não se considera os comandos anteriores, mas sim os eventos anteriores, tais como virar, colidir etc. Para isso, durante o *replay* são utilizadas outras variáveis e estruturas de dados. Com a adição

dessas variáveis, o consumo de memória aumenta muito pouco, em uma porcentagem muito pequena do consumo do jogo inteiro.

A diferença de desempenho entre o jogo e o *replay* está relacionada com os processos que são executados a mais para posicionar a câmera corretamente. São poucos os processos que não são executados no momento do replay; um deles é a exibição dos painéis informativos (HUD – *Head Up Display*), relativamente leve. A entrada de dados é substituída pela leitura da lista de comandos, que é levemente mais custosa para o sistema. A figura 15 mostra um ciclo completo do *game-loop*, que é executado no momento do *replay*. Durante a interação com o jogador, as etapas de reconhecimento de ação, previsão física e posicionamento da câmera não estão presentes. Todas essas etapas podem representar um acréscimo de até 50% no processamento da aplicação como um todo. A diferença foi medida através do *frame rate* do jogo em execução.

Figura 15: Fluxo de game-loop.

Outro fato observado foi que, durante o estado em que o jogador pode controlar a nave, existe um processamento muito maior quando a nave colide com um cristal azul, pois é preciso testar se a jogada especial foi concluída. Em caso positivo, ainda existe o trabalho de transformar os ítens da matriz da fase. Porém, durante o *replay*, a colisão com um cristal azul não aumenta tanto o trabalho de processamento, pois já

existem muitos outros processos acontecendo nesse momento. O gargalo de processamento se transforma da renderização durante o jogo, para a lógica de posicionar a câmera dinamicamente durante o *replay*.

Capítulo 7

Conclusões

Este trabalho mostrou uma solução para posicionamento de câmeras, a partir de informações passadas pelo motor de jogo, usando as técnicas de reconhecimento de ações e previsão física dentro do ambiente virtual, além de conceitos de cinematografia. Foi mostrado como funciona a plataforma XNA, com a qual o jogo para exemplo foi construído, e algumas técnicas cinematográficas que foram seguidas.

Foram apresentados todas as estruturas de dados utilizadas, destacando-se como os objetos, comportamentos e eventos se relacionam dentro do sistema, bem como a Árvore de Eventos, utilizada pelo agente diretor para montar uma cena. A Árvore de Eventos é capaz de mesclar o resultado das etapas de reconhecimento de ações e previsão da simulação física, deixando tudo à disposição do diretor, auxiliando-o na montagem da cena com significado cinematográfico relevante. O uso dessas estruturas se mostrou uma boa opção na tentativa de minimizar o consumo de memória e melhorar o desempenho do sistema. Outro fator fundamental para o sucesso da técnica foi a adição de um fator de aleatoriedade, que evita o determinismo e deixa o resultado final mais interessante para o telespectador, além de ser computacionalmente mais rápido que qualquer outra técnica de decisão.

Todas as adaptações no ambiente virtual e na dinâmica do jogo foram mostradas de forma que a técnica pudesse ser aplicada corretamente e de forma otimizada. Todas as vantagens e particularidades do *framework* do XNA foram aproveitados para aumentar o desempenho da aplicação.

De acordo com as experiências tomadas e resultados alcançados, conclui-se que a utilização da técnica é viável, tanto para a geração de *replays*, quanto para a transmissão em tempo real para um grupo de telespectadores, como explicado em [Drucker 2005]. Porém, em algumas plataformas a aplicação pode ser executada exigindo o máximo dos recursos disponíveis, por isso é preciso tomar certos cuidados na substituição de técnicas utilizadas. Por exemplo, na determinação da similaridade entre duas listas, uma técnica mais sofisticada pode terminar por sobrecarregar o sistema e comprometer a boa execução da aplicação. É importante passar por uma fase de testes antes de usar uma técnica diferente para um dado fim.

Como extensões da técnica e trabalhos futuros, sugere-se a implementação do ambiente de rede para o jogo, fazendo com que outros usuários conectados ao jogo recebessem as imagens de forma cinematográfica, sendo assim puramente espectadores. Com isso poderiam ser avaliados também o volume de dados transmitidos pela rede e até mesmo rever a discussão sobre a transmissão de imagens da cena de diferentes formas aos telespectadores, validando ou não a aleatoriedade nas escolhas do agente diretor.

Referências Bibliográficas

AMERSON, D. E KIME, S., 2001. Real-time Cinematic Camera Control for Interactive Narratives. Anais da AAAI SSS 2001, pág. 1-4.

BARELLA, A., CARRASCOSA, C., BOTTI, V., 2007. Agent Architectures for Intelligent Virtual Environments. Anais da IEEE/WIC/ACM IAT 2007.

BLIZZARD, 2009. Warcraft III [online]. Disponível em http://www.blizzard.com/war3/ [Acessado em 17 de março de 2009].

CAPCOM, 2009. Resident Evil [online]. Disponível em http://www.residentevil.com [Acessado em 17 de março de 2009].

CARTER, C., 2007. Microsoft XNA Unleashed. Editora Sams. 2007.

DRUCKER, S., 1994. Intelligent Camera Control for Graphical Environments. Tese de Doutorado, Massachusetts Institute of Technology.

ERLEBEN, K., 2002. Module Based Design for Rigid Body Simulators. Relatório técnico, University of Copenhagen.

ESPOSITO, N., 2005. A Short and Simple Definition of What a Video Game Is. Anais da DiGRA 2005.

FEGELEIN, 2008. Microsoft XNA Framework; Creating a Freelook Camera [online]. Disponível em http://www.fegelein.com/?p=18 [Acessado em 8 de agosto de 2008].

FILHO, M., 2007. Direção de Fotografia Aplicada aos Games [online]. Disponível em http://www.benzaiten.com.br/artigo/direcao_de_fotografia.htm [Acessado em 27 de novembro de 2008].

HAWKINS, B., 2005. Real-Time Cinematography for Games. Editora Charles River Media, 2005.

HE, L., COHEN, M., SALESIN, D., 1996. The Virtual Cinematographer: A Paradigm for Automatic Real-Time Camera Control and Directing. Anais da ACM SIGGRAPH 96, pág. 217-224

HERMANN, R. E CELES, W., 2005. Posicionamento Automático de Câmeras em Ambientes Virtuais Dinâmicos. Anais do SBGames 2005, pág. 174-185

MARCHAND, É. E COURTY, N., 2002. Controlling a Camera in a Virtual Environment. The Visual Computer Journal, fevereiro de 2002, pág. 1-19.

MARTIN, M., 1985. A Linguagem Cinematográfica. Editora Brasiliense, 1985.

MNEMOCINE, 2008. Linguagem e Técnica Cinematográfica [online]. Disponível em http://www.mnemocine.com.br [Acessado em 8 de agosto de 2008].

MSDN, 2009. XNA Developer Center [online]. Disponível em http://msdn.microsoft.com/en-us/xna/default.aspx [Acessado em 11 de fevereiro de 2009].

NINTENDO, 2009. Super Mario 64 [online]. Disponível em www.nintendods.com/sm64ds/ [Acessado em 17 de março de 2009].

NITSCHKE, B., 2007. Professional XNA Game Programming for Xbox 360 and Windows. Editora Wrox, 2007.

PINHANEZ, C., 1999. Representation and Recognition of Action in Interactive Spaces. Tese de Doutorado, Massachusetts Institute of Technology.

PIRES, D., PASSOS, E., CLUA, E., 2008. Posicionamento de Câmeras através de Previsão das Simulações Físicas. Anais da SBGames 2008.

RIEMERS, 2008. Quaternion Camera [online]. Disponível em http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series2/Quaternions.php [Acessado em 8 de agosto de 2008].

ROUSE, R., 2001. Game Design Theory & Practice. Editora Wordware Publishing. 2001.

SEUGLING, A. E RÖLIN, M., 2006. Evaluation of Physics Engines and Implementation of a Physics Module in a 3D Authoring Tool. Tese de Mestrado, Umea University.

SONY, 2009.a. God of War [online]. Disponível em http://www.godofwar.com [Acessado em 17 de março de 2009].

SONY, 2009.b. Gran Turismo 3 [online]. Disponível em http://www.granturismo-3.com [Acessado em 17 de março de 2009].

TAVEIRA, A., BARREIRO, A., BAGNATO, V., 2009. Simples Demonstração do Movimento de Projéteis em Sala de Aula [online]. Disponível em http://www.periodicos.ufsc.br/index.php/fisica/article/view/7504/6885 [Acessado em 15 de abril de 2009].

VALVE, 2009. Counter Strike: Source on Steam [online]. Disponível em http://www.counter-strike.net [Acessado em 17 de março de 2009].