UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Estudo e Experimentação de Animação Baseada em Comportamento

por

Marcelo Cohen

Dissertação submetida como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Flávio Rech Wagner
Orientador

Prof. Carla Maria Dal Sasso Freitas
Co-orientadora

Porto Alegre, Janeiro de 1996.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Cohen, Marcelo

Estudo e Experimentação de Animação Baseada em Comportamento / Marcelo Cohen. - Porto Alegre: CPGCC da UFRGS, 1995.
121 p.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1995. Orientador: Wagner, Flávio Rech; Co-orientador: Freitas, Carla Maria Dal Sasso

1. Computação gráfica. 2. Animação. 3. Comportamento. 4. Animação por comportamento. I. Wagner, Flávio Rech. II. Freitas, Carla, Maria Dal Sasso. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Hélgio Trindade
Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Claudio Scherer
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenador do CPGCC: Prof. José Antônio Palazzo de Oliveira
Bibliotecária - Chefe do Instituto de Informática: Zita Prates de Oliveira

AGRADECIMENTOS

Aos meus pais - Saul e Esther - e à minha irmã - Renata - que a vida inteira me apoiaram e que nos momentos difíceis aturaram pacientemente o meu mau humor;

À minha esposa, Elisa, pelo simples fato de ter cruzado o meu caminho. E também por ter ficado, sempre que era possível, ao meu lado;

À Carla, por toda a valiosa ajuda prestada e por não me deixar desanimar nunca;

A todos os meus amigos, que felizmente são tantos que é impossível agradecer a cada um;

Muito obrigado.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A descrição de animação modelada sempre foi motivo de pesquisa na área de Computação Gráfica. Visando simplificar a especificação do movimento, diversas técnicas surgiram: animação por quadros-chave, algorítmica (ou procedimental), baseada em distorções dos objetos e baseada em comportamento, entre outras.

Este trabalho apresenta uma nova abordagem para a obtenção de animação, mais especificamente, animação modelada pela definição de comportamento, na qual a idéia básica é que o usuário não deve ter o controle total e absoluto sobre todas as ações dos atores, mas sim deve ter uma noção de como a seqüência deve se desenvolver.

Esta abordagem desenvolveu-se a partir de um estudo realizado das principais técnicas existentes para a obtenção desse tipo de animação, procurando reconhecer as características positivas de cada uma, a fim de tentar reuní-las em uma abordagem única. Foram estudadas as seguintes abordagens: caminhos pré-definidos, *sensor-effector*, regras de comportamento, algoritmos genéticos e relações.

No sentido de facilitar ao máximo esta especificação, surgiu outro objetivo importante para o trabalho: a criação de um protótipo, o qual deveria possuir uma *interface* gráfica, buscando automatizar e simplificar as operações necessárias à definição das animações. O desenvolvimento deste protótipo serviu como meio para validar a abordagem, provando que esta pode ser efetivamente utilizada para a criação de seqüências animadas, com algumas limitações impostas por sua própria natureza.

O protótipo permite a definição de animações de uma forma totalmente interativa, facilitando a descrição de comportamentos e a manipulação dos atores em uma cena. Possui ainda uma função de pré-visualização, para visualizar as animações rapidamente e funções para gravação dos quadros individuais.

PALAVRAS-CHAVE: Computação Gráfica, Animação, Comportamento.

TITLE: "Study and Experimentation on Behavioral Animation"

ABSTRACT

The description of modeled animation had always been a research subject in Computer Graphics. To make the movement description easier, some methods then have appeared: keyframing, algorythmic animation, physically-based, based in object distortion, based in behavior description, among others.

This work presents a new approach to obtain computer animation, specifically, behavioral modelled animation, where the main idea is that the user shouldn't have total control about all actors' actions, but only a notion on how the sequence should look like.

This approach was developed from a study of the present research in behavioral animation, trying to recognize the good points of each method and join them in a single approach. The following methods were studied: predefined paths, *sensor-effector*, behavior rules, genetic algorithms and relations.

In the sense to ease at the most this specification, another goal showed up: the creation of a prototype, which should have a graphical interface, looking for the automatization and simplification of the needed operations for sequence definition. The development of this prototype served as a means to validate the new approach, proving that it can be used for the creation of animated sequences, with some limitations imposed by its own features.

The prototype allows the definition of animations in a fully interactive way, easing behavior description and actor manipulation in a scene. There is a preview function, to watch animations quickly and easily. Also are provided functions for saving individual frames.

KEYWORDS: Computer Graphics, Animation, Behavior.

1 INTRODUÇÃO

1.1 Conceitos e Objetivos

Produzir animação é uma atividade (ou arte) bastante antiga. O primeiro registro disponível data de 1831, quando o francês Joseph Antoine Plateau inventou um dispositivo denominado *Phenakistoscope*, que consistia de um disco giratório sobre o qual era colocada uma série de desenhos, e de uma janela , através da qual o observador tinha a sensação de movimento da seqüência de desenhos [MAG 85]. Do invento de Plateau até nossos dias as técnicas de exibição evoluiram muito. Porém, a idéia básica do que seja animação ainda persiste, conforme pode ser notado nas diversas definições contemporâneas:

Animação - é a técnica que abrange qualquer forma cinematográfica na qual um objeto, muitas vezes imóvel, é visto movendo-se ou mudando de forma;

Animação - é o processo de geração dinâmica de uma série de quadros de um conjunto de objetos, no qual cada quadro é obtido a partir de uma alteração do quadro anterior [MAG 85].

Em animação tradicional, o seguinte conjunto de tarefas deve ser realizado:

1. Criação da História - Sinopse e *Storyboard*;
2. Produção do *Layout* - Projeto de atores e cenário;
3. Gravação da trilha sonora;
4. Animação - desenho dos quadros-chave;
5. Interpolação - desenho / geração dos quadros intermediários;

6. Verificação

7. Gravação.

Esta é uma área que desperta muito interesse em Computação Gráfica, principalmente graças aos resultados visuais obtidos e às diversas aplicações possíveis. Isto pode ser claramente observado pela quantidade de trabalhos já desenvolvidos.

Animação por computador é o termo que caracteriza a utilização de um sistema de computação para realizar algumas ou todas as tarefas citadas. Esta utilização pode ser classificada em cinco níveis, baseada no nível de especialização das tarefas que este sistema desempenha [MAG 85]:

- Nível 1: o computador é utilizado apenas para criação, pintura, armazenamento e modificação dos desenhos. São tipicamente *painting systems*, ou seja, sistemas voltados para a pintura, onde o usuário utiliza o computador simplesmente para desenhar e colorir as imagens;

- Nível 2: os sistemas deste nível são capazes de calcular e gerar automaticamente os quadros intermediários. São geralmente sistemas que realizam animação bidimensional;

- Nível 3: permitem que o animador realize operações sobre os objetos, como rotação, translação, escala, etc. Permitem também, geralmente, a manipulação de uma câmera virtual. Este é o tipo de sistema que surgiu no início das pesquisas em animação: não existe ainda o conceito de ator;

- Nível 4: são capazes de diferenciar os diversos componentes de uma cena (atores, cenário e câmeras), permitindo a definição de atores com movimento próprio. Aqui já existe o conceito de ator, diferenciando-o de um simples objeto;

- Nível 5: os sistemas deste nível são denominados "inteligentes", pois são capazes de "aprender" à medida que executam uma animação.

Os níveis 1 e 2 caracterizam processos de **animação auxiliada por computador** e os níveis 3, 4 e 5 caracterizam **animação modelada por computador**.

Neste trabalho há uma distinção entre um simples objeto e um ator. O primeiro é simplesmente a descrição de uma geometria, geralmente com atributos adicionais, como cor, textura, etc. Um ator engloba o conceito de objeto, mas apresenta alguns elementos adicionais:

- um ou mais sistemas perceptuais, responsáveis pela detecção de estímulos provenientes do ambiente onde se encontra;
- um sistema de coordenação central, o qual pode ser visto como o "cérebro" do ator, responsável pela escolha final da ação a ser realizada;
- um sistema de coordenação de comportamento, responsável pela presença de objetivos, direcionamentos (fome, cansaço, desejo sexual, etc) e memória;
- um sistema locomotor, o qual efetivamente realiza o movimento do ator, de acordo com as instruções recebidas do sistema de coordenação central;

A figura 1.1 ilustra claramente o relacionamento entre os quatro sistemas:

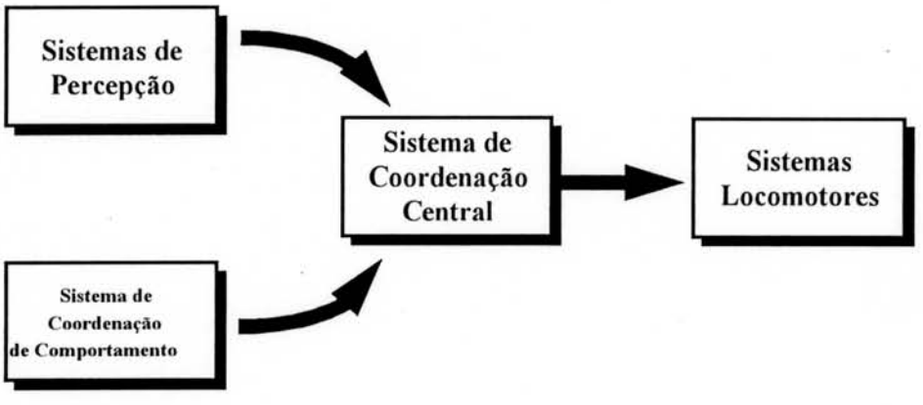


Figura 1.1 - Componentes de um ator

Outro conceito vital para o desenvolvimento do trabalho é o de **comportamento**. De acordo com a teoria clássica de estímulo e resposta [SKI 38], um comportamento é definido como uma seqüência alternada de eventos de estímulo e resposta, na qual um estímulo pode "aumentar a probabilidade da emissão" de uma resposta, a qual por sua vez gera outro estímulo. Já em termos de animação, comportamento pode ser descrito como:

- **movimento** - durante uma seqüência, os atores realizam e/ou deixam de realizar movimento, dependendo da situação. Essa ação (ou inação) pode ser chamada de comportamento;
- **mudança de aspecto** - transformações que afetam a aparência do ator (cor, forma, tamanho) também são consideradas como comportamento;

A última definição necessária é a de **ambiente**.

Ambiente - o ambiente se constitui do conjunto de objetos que funcionam como cenário de uma animação, ou seja, os objetos que não são os atores de interesse [SUN 93];

A partir do momento em que um ator é colocado em um ambiente, este sofre restrições adicionais, introduzidas pelo último. Estas restrições surgem de influências estáticas e dinâmicas do ambiente. Influências estáticas são aquelas que não variam no decorrer de uma animação, como, por exemplo, limites do ambiente, obstáculos e demais objetos estáticos. Já objetos em movimento e eventos variados são chamados de influências dinâmicas.

1.2 Organização do Texto

O tema deste trabalho é desenvolvido nos seis capítulos restantes da seguinte forma.

No capítulo 2 serão apresentadas as abordagens existentes para a obtenção de animação por comportamento. No capítulo 3 é realizada uma comparação entre as abordagens mencionadas, utilizando 4 (quatro) critérios: **generalidade, resultados, interatividade e simplicidade**. No capítulo 4 é descrita a abordagem proposta, bem como o funcionamento do método desenvolvido. O capítulo 5 trata da operação e de detalhes na utilização da interface gráfica. O capítulo 6 visa analisar o protótipo desenvolvido, com base nos mesmos critérios mencionados acima. Esta análise tem como objetivo justificar a validade ou não do método criado.

O capítulo 7 toma como base o resultado dos diversos testes realizados com o protótipo e conclui o trabalho, citando as possíveis melhorias e mudanças que poderiam ter sido feitas (ou que podem vir a ser realizadas) no sistema (e/ou no método), de forma a torná-lo mais eficiente e mais útil.

2 ABORDAGENS DE ANIMAÇÃO BASEADA EM COMPORTAMENTO

2.1 Caminho pré-definido

A técnica consiste em se considerar que o ator seguirá um caminho pré-estabelecido. Este caminho é geralmente representado através de pontos-chave, por onde o ator deverá obrigatoriamente passar. É comum também a utilização de técnicas de interpolação, a fim de suavizar o caminho final (figura 2.1).

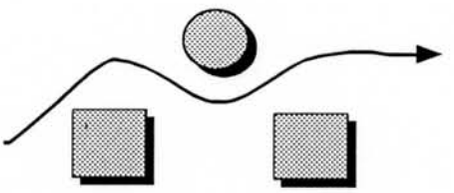


Figura 2.1 - Caminho pré-definido

O ambiente é totalmente conhecido antes de se iniciar qualquer movimento, bastando, portanto, analisar todas as possibilidades de caminhos e, de posse destas, escolher o melhor. Essa escolha segue algum critério, como por exemplo, obter o caminho mais curto.

Esta abordagem tem algumas vantagens: pode-se criar um movimento bastante complexo, uma vez que todo o caminho é calculado ANTES da geração de movimento. Porém, já que o ambiente é estático, a menor alteração neste exigirá que se recalcule parte ou todo o caminho (figura 2.2).

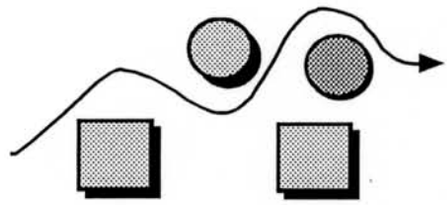


Figura 2.2 - Inclusão de obstáculo no ambiente

Green [SUN 93] aponta como representante desta abordagem o trabalho de Latombe [LAT 89], que utiliza um grafo de visibilidade pré-computado para os obstáculos, o que gera um caminho sem colisões e com custo mínimo.

2.2 *Sensor-Effector*

O comportamento nesta abordagem é descrito por sensores, atuadores (*effectors*) e uma rede neural os conectando.

A maioria das pesquisas utilizando esta abordagem têm sua base no trabalho de Braitenberg [BRA 84], onde são mostrados exemplos de comportamento produzidos em um ambiente simples, gerados através de um modelo conectivo, simulado por fios e partes mecânicas.

Baseado nas idéias de Braitenberg [BRA 84], *BrainWorks* é uma interface gráfica interativa para a construção de um sistema nervoso de um animal simplificado. Uma vez criado, pode-se verificar diversos tipos de comportamento, tais como atração e repulsão.

Porém, do ponto de vista de animação, o trabalho mais significativo é o de Jane Wilhelms [WIL 90], também inspirado nas idéias de Braitenberg. A idéia aqui é a construção de uma rede neural, interligando sensores (*sensors*) e atuadores (*effectors*). Os sensores são responsáveis por coletar dados sobre o ambiente e podem ser de dois tipos:

- distância: detectam a distância entre objetos. São geralmente utilizados para evitar colisões. Existem também sensores de proximidade, que tem o papel inverso;

- qualidade: detectam propriedades de objetos, como por exemplo, cor. Um objeto pode apresentar zero ou mais qualidades, definidas por um valor escalar.

Os sensores são conectados aos atuadores, os quais funcionam como motores a jato presos a determinadas partes do objeto. Seu objetivo é realizar o movimento, simulando a atuação de uma força. O modelo dinâmico utilizado é uma simplificação, no qual uma força é proporcional à velocidade e não à aceleração.

Entretanto, esta conexão não é feita diretamente, mas sim através de um ou mais nodos de conexão. Estes nodos recebem a entrada de um ou mais sensores, aplicam uma determinada função e enviam a saída para um ou mais atuadores. A figura 2.3 apresenta um exemplo deste processo: a partir de quatro sensores, um par diferenciando uma qualidade (cor) e o outro detectando distância, o comportamento do ator varia entre a atração e a repulsão. No exemplo mostrado, o ator é atraído por objetos (ou atores) de cor azul e repelido pela cor vermelha.

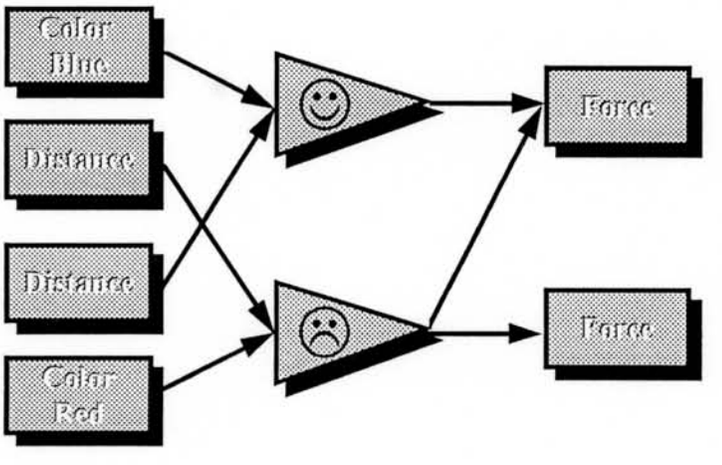


Figura 2.3 - Rede de conexão

2.3 Regras de Comportamento

Semelhante à abordagem anterior, esta também usa dados sensoreados como entrada e controle de motores como saída. A distinção entre as duas consiste na utilizacão de regras para a determinação do comportamento, em vez de uma rede neural.

Um exemplo de regras de comportamento, obtido de [COD 88] é:

1. Se ambos FOME e MEDO são fortes, faça uma composição entre COMBATER e PROCURAR ALIMENTO;
2. Se MEDO é forte, COMBATER;
3. Se FOME é forte, PROCURAR ALIMENTO;
4. Se PROCURA informa que há alimento disponível, PROCURAR ALIMENTO;
5. Caso contrário, EXPLORAR.

Neste exemplo, o ator em questão pode executar três comportamentos distintos: COMBATER, PROCURAR ALIMENTO e EXPLORAR. O que vai determinar a mudança de comportamento são indicadores, como FOME, MEDO e a presença ou não de alimento. É então construída uma árvore de decisão, permitindo a escolha da regra a ser usada, de acordo com algum critério. Este pode ser, por exemplo, a atribuição de pesos ou limiares às regras, de forma que as regras de maior prioridade sejam avaliadas antes.

Um trabalho utilizando esta abordagem é *Petworld* [COD 88], que modela um universo bidimensional de animais, pedras e árvores. Os animais possuem uma orientação e podem movimentar-se nessa direção, além de poderem carregar uma pedra, comer árvores para se alimentar e utilizar pedras para construir ninhos. Ainda podem morrer de fome ou de ferimentos sofridos em combate com outros animais. É importante observar que o objetivo aqui não é modelar uma animação, mas sim simular um ecossistema simples, sem se preocupar com o movimento resultante.

Utilizando a mesma abordagem com objetivos completamente diferentes, Reynolds [REY 87] modela o comportamento de coletivos, mais especificamente pássaros. Esse comportamento baseia-se também em regras simples, em ordem decrescente de importância:

1. Evitar colisões com os vizinhos próximos;

2. Procurar igualar a velocidade com os vizinhos próximos;

3. Procurar ficar o mais próximo possível dos vizinhos;

É importante observar que o modelo é local, uma vez que não considera todo o bando na sua avaliação, mas apenas os pássaros mais próximos daquele que está se analisando. O objetivo é explicitamente gerar uma animação, considerando o comportamento dos pássaros em um ambiente restrito.

A figura 2.4 apresenta dois momentos em uma animação gerada por esta técnica: no primeiro, o bando aproxima-se de um obstáculo, mas não sofre qualquer influência deste; no segundo, o bando têm que desviar, para evitar a colisão.

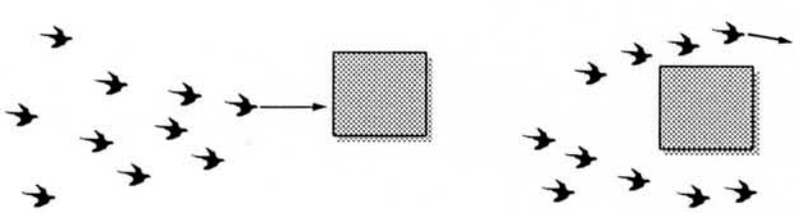


Figura 2.4 - Desenvolvimento do comportamento de coletivos

2.4 Algoritmos Genéticos

A idéia de algoritmos genéticos foi originalmente desenvolvida por Holland [HOL 75]. Porém, foi John Koza [KOZ 92] quem adaptou a idéia original de Holland para aplicar em programas de computador, criando o conceito de Programação Genética (PG): técnica para a geração automática de programas, que satisfaçam algum critério específico de aptidão (*fitness*). Essa aptidão diz respeito à eficiência que um programa gerado tem para resolver um problema específico e é computada a cada vez que o programa é executado. É criada uma população inicial de programas, normalmente de forma estocástica. Uma vez avaliada a aptidão de cada um, escolhe-se geralmente os dois melhores e gera-se um "filho", isto é, um novo programa combinando porções dos dois originais. O processo é repetido até se obter uma solução satisfatória, o que em geral significa haver um programa na população que tenha atingido uma aptidão suficientemente alta.

Aplicando esses conceitos, Reynolds [REY 92, REY 93, REY 93a] utilizou esta abordagem para gerar programas em LISP que resolvessem o problema de movimentar um objeto em um ambiente com obstáculos. Primeiro, considerou somente a questão da movimentação de um único ser. Aqui o objetivo era simplesmente conseguir um programa que realizasse a movimentação do ser, evitando colidir com os obstáculos (paredes). Em um segundo passo, considerou a interação de vários seres "amigáveis" e um predador. Agora, além de evitar a colisão com as paredes, era necessário não colidir com os parceiros e fugir do predador. Finalmente, apontou a necessidade de haver "ruído" no sistema, pois os sensores que detectam os obstáculos devem retornar dados imprecisos, uma vez que o objetivo é simular um ambiente mais próximo da realidade, ou seja, com variações. Esse ruído é simplesmente uma fração randômica acrescentada ao resultado.

A figura 2.5 apresenta dois momentos de um processo evolucionário: com uma aptidão de 42% (não satisfazendo os objetivos, pois o ator sofre uma colisão) e com aptidão de 100% (satisfazendo os objetivos, pois o ator percorre todo o ambiente sem colidir com as paredes).

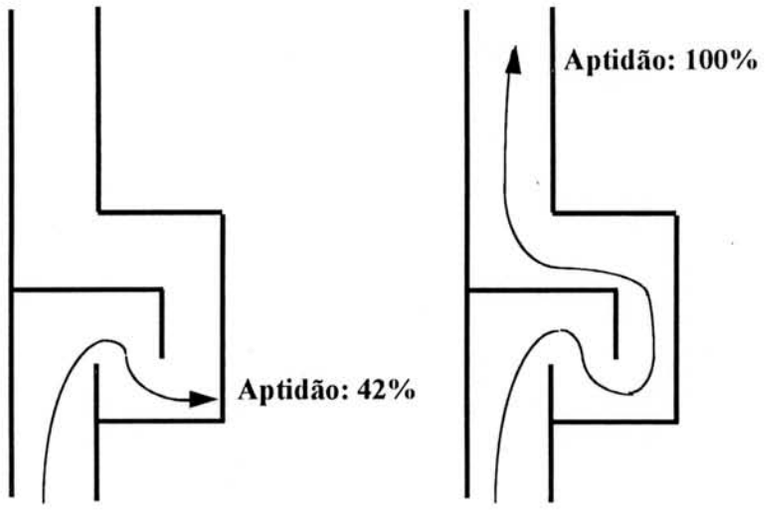


Figura 2.5 - Evolução de um Algoritmo Genético

Polyworld [YAE 93], trabalho de Larry Yaeger, é um simulador ecológico complexo, unindo técnicas de Computação Gráfica, redes neurais e um algoritmo genético *steady-state*, isto é, onde o tamanho da população de gens é sempre mantido o mesmo (pois a cada ser criado, um é excluído).

O universo é representado por um plano, dividido por algumas barreiras intransponíveis. A comida é representada por objetos de cor verde, dispostos aleatoriamente. Os seres que habitam esse universo utilizam a visão como entrada para o cérebro, modelado por uma rede neural. Estes podem combater entre si, buscar comida e se reproduzir. A reprodução realiza as operações de mutação e *crossover*, definidas pelo algoritmo genético, gerando um novo ser a partir do código genético de outros dois.

2.5 Relações

O ambiente pode influenciar o movimento dos atores de muitas formas. Por exemplo, a presença de obstáculos geralmente inibe ou limita certos movimentos, pois os atores normalmente devem evitá-los. Porém, enquanto evita um obstáculo, o ator pode ser obrigado a se desviar de outro ator próximo, outro obstáculo ou até algum evento inesperado que venha a ocorrer. Este exemplo serve para demonstrar

que o comportamento é determinado pelo conjunto de influências atuando sobre o ator.

A abordagem de Green [SUN 93] visa formalizar a definição dessas influências, através do conceito de **relação**. Uma relação serve para especificar como o ambiente afeta o movimento de um ator. Possui três componentes principais:

- Fonte: objeto(s) que causa(m) o movimento;
- Destino: objeto(s) que realiza(m) o movimento;
- Resposta: especifica de que forma o destino agirá quando influenciado pelo fonte, ou seja, especifica o movimento resultante.

A resposta só é executada quando um conjunto de condições para o objeto fonte sejam satisfeitas, como por exemplo: distância, hostilidade, tamanho, cor, etc. Ainda há parâmetros que permitem alterar a resposta ao longo da animação.

2.5.1 Estados de Controle

Uma relação pode estar em quatro estados de controle:

- potencial: pode participar da ação, mas está inativa no momento;
- ativa: quando a relação está contribuindo para o movimento;
- suspensa: quando a relação está bloqueada por algum motivo;
- encerrada: quando a relação não pode mais ser utilizada, o que ocorre quando ou o fonte ou o destino desaparecem do ambiente.

2.5.2 Mecanismos de controle

A fim de facilitar a concorrência de relações quando ocorrem conflitos, foram criados alguns mecanismos de controle:

- controle de ambiente: altera o estado de controle de uma relação quando o ambiente é modificado de alguma forma;
- controle de interação: permite com que uma relação altere o estado de controle de outra;
- controle de padrão: permite ao animador controlar as mudanças de estado através de dois mecanismos, via temporização e via relações. O controle via temporização altera o estado de um conjunto de relações para potencial em um determinado instante de tempo. O controle via relações faz com que o estado de certas relações torne-se potencial, quando uma relação específica for ativada.
- controle de seqüência: baseia-se na ordenação temporal de padrões de comportamento.

Os demais conflitos são resolvidos através da atribuição de uma prioridade a cada relação pertencente ao mesmo aspecto do movimento.

3 COMPARAÇÃO DAS ABORDAGENS

Os seguintes critérios foram escolhidos e utilizados para a comparação das abordagens:

- Generalidade: informa se o método é utilizável em outras aplicações além da apresentada no artigo correspondente;
- Resultado: diz respeito à eficiência do método, ou seja, se o movimento gerado é satisfatório em termos visuais;
- Interatividade: mostra se a interação com o usuário (se houver) é simplificada, isto é, se a *interface* é intuitiva;
- Simplicidade: indica a dificuldade de se obter um determinado comportamento, utilizando os recursos do método.

Esta comparação é apresentada na tabela 3.1.

Tabela 3.1 - Tabela comparativa entre as abordagens

Abordagem	Generalidade	Resultado	Interatividade	Simplicidade
Caminho pré-definido	Ruim	Bom	Nenhuma	Simples
Sensor-Effector	Razoável	Muito Bom	Alta	Razoável
Regras de Comportamento	Boa	Muito Bom	Baixa	Razoável
Relações	Muito Boa	Muito Bom	Alta	Complexa
Algoritmos Genéticos	Ruim	Bom	Nenhuma	Simples

3.1 Generalidade

Em termos de generalidade, os piores métodos são os que utilizam algoritmos genéticos e caminhos pré-definidos. O primeiro é um método aplicável somente a situações bem específicas, pois cada tipo de comportamento exige um critério de aptidão diferente. O segundo tem a restrição óbvia de exigir a presença de um caminho completo antes do processamento da animação, ou seja, não há nenhum tipo de decisão no decorrer da seqüência.

O método *sensor-effector* é um pouco mais genérico, na medida que é o usuário quem cria a rede e, conseqüentemente, dita as regras a serem seguidas por cada ator. Porém, a forma com que as regras são descritas limita a aplicação do método, uma vez que a rede é imutável.

A abordagem de regras de comportamento permite uma generalidade um pouco maior, já que estas podem ser criadas como o usuário assim o desejar, incluindo condições ao longo do tempo. O grande problema é a questão de como são definidas as regras utilizáveis.

A análise das cinco abordagens determina como mais genérica a de relações, pois além de permitir a definição das regras através de um formalismo, apresenta uma grande capacidade de controlar a cena a partir de condições que surjam, instantes de tempo, etc.

3.2 Resultados

Considerando o critério de resultados, nenhuma abordagem se destaca em relação às outras. As que apresentam o menor desempenho (que mesmo assim ainda pode ser considerado bom) são as mesmas que têm a menor generalidade: caminho pré-definido e algoritmos genéticos. A primeira gera um caminho satisfatório, livre de colisões, porém fixo. A segunda também, porém refina o caminho obtido a cada geração.

As outras três abordagens (*sensor-effector*, regras de comportamento e relações) permitem um maior controle do caminho gerado, com a vantagem que este último pode sofrer alterações no decorrer da animação.

3.3 Interatividade

Não há nenhuma interação com usuário nos métodos de caminho pré-definido e algoritmos genéticos. Isso ocorre porque os métodos funcionam sozinhos, não é necessária a intervenção do operador.

A abordagem de regras de comportamento permite um pouco de interação, pois o usuário deve, em algum momento, definir as regras que serão utilizadas.

Porém, os métodos que interagem graficamente, geralmente através da criação gráfica de uma estrutura de controle, são os que exibem o maior grau de interatividade: *sensor-effector* e relações.

3.4 Simplicidade

Do ponto de vista da simplicidade, os melhores métodos são os de caminho pré-definido e de algoritmos genéticos. O primeiro porque basta estabelecer um caminho e o segundo porque basta estabelecer um critério de aptidão. É importante ressaltar que essa simplicidade advém justamente da falta de generalidade dos métodos.

Apresentando uma maior dificuldade para obtenção de resultados, as abordagens *sensor-effector* e regras de comportamento exigem a criação de alguma estrutura adicional (uma rede ou um conjunto de regras). Porém, esta exigência é compensada pela generalidade e desempenho obtidos.

O método mais complexo é o de relações. Isto é devido principalmente à quantidade de recursos disponíveis, desde a utilização de código em C às várias estruturas de controle existentes.

4. ABORDAGEM PROPOSTA

Uma vez realizada a comparação das abordagens, são conhecidas as características e/ou propriedades mais relevantes de cada uma, do ponto de vista da descrição do comportamento. A abordagem descrita a seguir busca reunir algumas destas características. Na verdade, características das abordagens *Sensor -Effector* e *Relações*, pois as demais (caminhos pré-definidos e algoritmos genéticos) não apresentam a generalidade e interatividade necessárias.

A abordagem baseia-se na definição de três elementos básicos (classes de atores, comportamentos e cenas), os quais são descritos a seguir.

4.1 Classes de Atores

A idéia de classe surge da premissa que é desejável a capacidade de manipulação de diferentes tipos de atores, com características distintas. A classe é, então, a estrutura para a definição de cada tipo de ator, permitindo a descrição de sua representação visual **(geometria)** e suas características **(atributos)**. As classes são modeladas hierarquicamente, cada uma sendo derivada de alguma outra já existente, o que visa facilitar o processo de criação de novas classes.

4.1.1 Geometria

A geometria descreve a representação visual do objeto. Apesar de não ser uma condição obrigatória, optou-se pela utilização de objetos tridimensionais, modelados por polígonos (faces). Esta representação é comumente denominada *B-Rep* (*boundary representation*) [WAT 92], uma vez que não há informação de volume, mas apenas de superfícies limitantes (figura 4.1). O objeto é visto como uma "casca", é oco. Este tipo de representação geralmente modela o objeto de forma a centralizá-lo

na origem do universo, o que simplifica a sua exibição. Há três razões importantes para a escolha deste tipo de representação:

- é um formato suportado por praticamente todos os sistemas de modelagem tridimensional, o que facilita a importação de objetos para utilização no sistema;
- é um tipo de representação facilmente armazenável e recuperável, o que é desejável para a aceleração do processo de desenho;
- é uma representação que apresenta facilidades para a realização de detecção de colisões, já que é possível ter uma idéia exata do volume ocupado pelos objetos, porque se possui a informação sobre todas as suas faces.

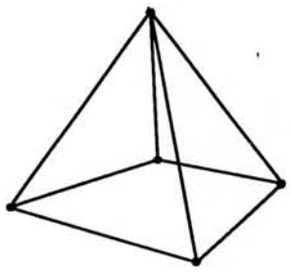


Figura 4.1 - Geometria descrita por superfícies limitantes (*B-Rep*).

Para cada objeto lido pelo sistema, são calculados dois pares de vértices, os quais definem os cantos de um cubo que envolve totalmente este objeto. Este "cubo envolvente" é comumente denominado *bounding box* (figura 4.2) e, posteriormente, será utilizado no algoritmo de detecção de colisões.

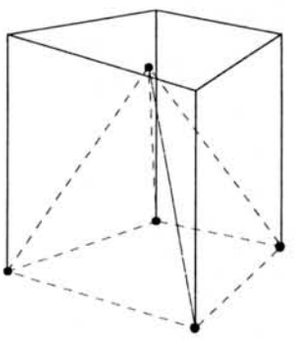


Figura 4.2 - Objeto tridimensional e sua *bounding box.*

4.1.2 Atributos

Os atributos são utilizados para descrever as características de uma classe de atores. Funcionam como variáveis de um programa, podendo ser alterados no decorrer da animação, utilizados para cálculos, ou como estruturas de controle. São utilizados dois tipos de atributos: simples e compostos.

Atributos simples modelam grandezas escalares (como velocidade, peso, etc), isto é, números. Para efeitos de simplificação, todos os atributos simples são armazenados como números em ponto flutuante (reais).

Atributos compostos são utilizados para o armazenamento de grandezas vetoriais (direção, força, etc) ou grandezas que tenham que ser representadas por três componentes (posição, orientação e escala de um objeto no espaço, etc). Os atributos compostos são representados por **[X, Y, Z]**, onde *X, Y* e *Z* definem os seus componentes.

4.1.3 Classe Padrão

Como mencionado na seção 4.1, cada classe é derivada de outra classe já existente. Para permitir esse tipo de modelagem, é preciso que haja uma superclasse, de forma que se possa derivar outras desta. A superclasse é denominada **Classe Padrão** e contém um certo número de atributos básicos. Estes são assim denominados porque são fundamentais para a posterior exibição de um ator (instância) da classe:

- **POSIÇÃO (Composto)** - Representa a posição do ator no espaço tridimensional. Na verdade, a posição que será utilizada como referência (origem) para o desenho da geometria e para a detecção de proximidade e colisão. Aqui surge a explicação do porquê de a geometria ter que ter sido especificada em torno da origem do universo: operações de escala e rotação são definidas em torno desta origem local;

- **ORIENTAÇÃO (Composto)** - Representa a orientação do ator no espaço tridimensional, em relação aos eixos X, Y e Z. É importante observar que apesar da orientação ser armazenada como uma grandeza composta, interpreta-se cada componente como um ângulo de rotação do ator (expresso em graus), em relação a cada um dos eixos coordenados. Esta rotação é efetuada na ordem X,Y e Z;

- **ESCALA (Composto)** - É o fator de escala utilizado na exibição da geometria do ator. Também armazenado como grandeza composta, mas interpretado como três fatores de escala: para as coordenadas nos eixos X, Y e Z;

- **DIREÇÃO (Composto)** - Indica a direção para onde o ator deve se deslocar no movimento. A "direção" especifica, em termos matemáticos, a **direção** e o **sentido** do vetor de deslocamento. É utilizada na avaliação de comportamento (vide seção 4.2.4), como um complemento ao movimento descrito;

- **VELOCIDADE (Simples)** - A velocidade é utilizada juntamente com a direção para compor o movimento complementar (vide seção 4.2.4), descrito parcialmente pela direção. Normalmente é positiva, mas valores negativos só têm o efeito de inverter o sentido do movimento. Os atributos *direção* e *velocidade* compoem, efetivamente, o **vetor velocidade** do movimento complementar.

- **COR (Composto)** - Utilizada unicamente na exibição do ator, para distingüi-lo de outros. Não é necessária na descrição nem na avaliação do comportamento. Cabe ressaltar que apesar de não ser preciso utilizá-la, pode ser alterada no decorrer da animação, para efeitos visuais. Também armazenada sob forma composta, mas representando cada um dos componentes **R (X),G (Y) e B (Z)** [WAT 89] de uma cor.

4.1.4 Herança de Atributos

Devido à modelagem das classes ser hierárquica, ao se criar uma classe derivada de outra, a nova classe herdará todos os atributos da original. Desta forma, é possível se aproveitar os atributos já existentes e simplesmente acrescentar novos quando for necessário, não sendo preciso criá-los novamente.

Como exemplo, considere-se a figura 4.3, demonstrando uma modelagem de veículos aéreos, compreendendo aviões e helicópteros. Todas as classes derivadas de *veículo aéreo* herdam todos os atributos de um "veículo aéreo". Os aviões, por exemplo, se subdividem em *comerciais* e *caças*. Um ator do tipo *comercial* possuirá posição, orientação, ..., altitude máxima e velocidade máxima - herdados de *veículo aéreo* - envergadura da asa e comprimento - herdadas de *aviões* - e seus próprios atributos, *no. de passageiros e autonomia de vôo*. Já para um caça não interessa o número de passageiros que ele é capaz de transportar (pois geralmente é apenas um ou dois), mas o *tipo de armamento* que contém, etc.

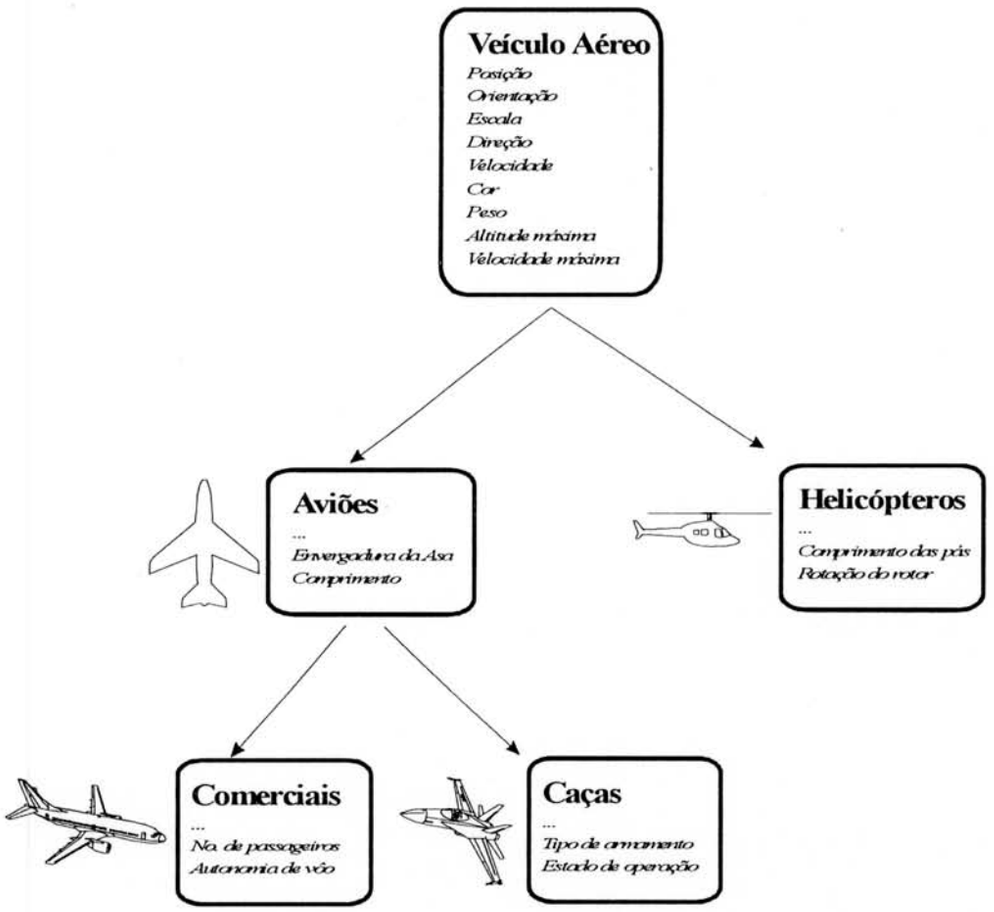


Figura 4.3 - Exemplo de modelagem com classes

4.2 Comportamento

O **comportamento** é o elemento mais importante da abordagem, pois é através de cada descrição de comportamento que as ações/movimentos dos atores são especificados. É, portanto, a base da animação, sem o qual a existência desta não tem sentido algum. Os comportamentos são criados para uma determinada classe, o que significa que somente atores desta classe específica podem apresentar estes comportamentos (e, naturalmente, atores de classes derivadas desta, pois herdam os atributos). Um comportamento é descrito com o auxílio de três estruturas de controle:

- Descrição de comandos (ou procedural) - através de uma linguagem de programação simplificada (vide seção 4.2.1), torna possível a descrição detalhada de cada comportamento;

- Especificação de restrições - permite restringir o movimento definido pelo nível de comando, através de instruções específicas (vide seção 4.2.2). Estas instruções são do tipo "evitar uma aproximação excessiva de um determinado ator", "ir para o mais longe possível de determinado ator", etc;

- Tratamento de eventos - permite identificar situações diversas - tais como proximidade ou colisão entre atores - que requeiram alguma ação imediata do ator (vide seção 4.2.3).

Além da classe de atores para o qual foi criado, também é associado um nome a cada comportamento, a fim de facilitar sua posterior identificação pelo usuário e também para a operação de troca de comportamento (vide seção 4.2.1.5) durante a animação.

4.2.1 Descrição de comandos

A fim de facilitar a descrição do comportamento, foi criada uma linguagem de programação simplificada, apresentando os seguintes tipos de instruções:

- Atribuições - permitem alterar os valores numéricos dos atributos;

- Iteradores - permitem a construção de repetições de blocos de comandos (*loops*);

- Condicionais - têm o objetivo de realizar tomadas de decisões no decorrer do comportamento;

- Troca de comportamento - causam a execução imediata de algum outro comportamento, cancelando o atualmente sendo executado pelo ator;

- Comportamentos primitivos - são assim denominados porque representam ações básicas, pré-programadas, que podem ser desempenhadas pelos atores;

- Composições - possibilitam a especificação direta de hierarquias de movimentação dentro de um comportamento, através de comportamentos primitivos e atribuições;

- Comunicação com outros atores - se dá através do envio de "mensagens", através das quais se pode passar dados (atributos ou valores númericos simplesmente) de um para outro;

- Criação de novo ator - permite que durante a execução de um comportamento seja possível a criação de um novo ator da mesma ou de outra classe de atores;

- Eliminação do ator - possibilita que o sistema deixe de considerar o ator como componente da animação, desaparecendo inclusive da visualização.

Comandos das classes acima são utilizados para descrever o comportamento básico de uma classe de atores, ou melhor, um comportamento que pode ser associado a um ator de determinada classe.

4.2.1.1 Atribuição

O comando de atribuição tem como objetivo a alteração do valor numérico de um determinado atributo. A finalidade desta alteração depende do contexto no qual este atributo está sendo utilizado. Exemplo: uma alteração do atributo *POSIÇÃO* é uma indicação clara de movimento, mas uma alteração de um atributo não-básico - como *PESO, MASSA, POTÊNCIA* - geralmente terá outra finalidade para o comportamento, como controlar alguma condição, por exemplo.

A sintaxe de um comando de atribuição é

```
<ATRIBUTO> = <EXPRESSÃO>
```

onde:

- `<ATRIBUTO>` - indica o nome de um atributo da classe para a qual o comportamento está sendo descrito. No caso de um atributo composto, pode-se especificar somente uma componente a ser alterada, através da forma `<ATRIBUTO>.x` ou `<ATRIBUTO>.y` ou `<ATRIBUTO>.z`;

- `<EXPRESSÃO>` - informa o novo valor deste atributo. Valores compostos são representados por `[<Expressão X>, <Expressão Y>, <Expressão Z>]`. Uma expressão pode incluir qualquer conjunto de operações matemáticas válidas, incluindo algumas funções pré-definidas (vide seção 5.5.2). O tipo da expressão (simples ou composto) deve coincidir com o tipo do atributo ao qual está sendo atribuída.

Exemplos de atribuições válidas:

- `posição = [ 10, -10, 20 ]`
- `posição.x = -50.7`
- `velocidade = velocidade + 5`
- `escala.x = escala.x * 0.2`

Exemplos de atribuições inválidas:

- `posição = sin(45)*cos(45)` - **O atributo é composto e a expressão é simples**
- `escala.x = [2, 2, 2]` - **O atributo é simples (componente X da escala) e a expressão é composta**

4.2.1.2 Iterador: PARA...

O comando **PARA...** é um dos comandos que possibilitam a construção de repetições, gerando uma variação incremental para um atributo. É equivalente a um comando do tipo ***for*** em uma linguagem como C ou Pascal. A sintaxe de um comando *Para* é

Para **<ATRIBUTO> = <Expr. Inicial>** *até* **<Expr. Final>** *passo* **<Expr. Passo>**
...<Comandos>...
Fim

onde:

- **<ATRIBUTO>** - atributo que será atualizado a cada iteração;
- **<Expr. Inicial>** - valor inicial que será atribuído ao atributo;
- **<Expr. Final>** - valor final que será atribuído ao atributo;
- **<Expr. Passo>** - valor que indica o deslocamento numérico a ser utilizado em cada iteração, isto é, quanto se deve adicionar ao (ou subtrair do) atributo a cada passo;
- **<Comandos>** - comandos a serem executados a cada iteração;

Uma observação importante é que o comando *Para* suporta tanto atributos simples como compostos. A única restrição que deve ser observada é que se o atributo é simples, todas as expressões têm que ser simples e se o atributo é composto, todas as expressões também têm de ser compostas.

Exemplos da utilização de iteradores *Para...*:

- Faz componente X da posição variar de 0 até 20, de um em um. A cada passo, adiciona 15 graus à orientação do ator em torno do eixo Y:

```
Para Posição.x = 0 até 20 passo 1
    Orientação.y = Orientação.y + 15
Fim
```

- Faz velocidade variar de 10 até 1, de 2 em 2 unidades A cada passo, atribui ao componente X da escala, o valor da velocidade; ao componente Y da escala, o valor da velocidade subtraído de 2; ao componente Z da escala, o valor da velocidade acrescido de 5:

```
Para Velocidade = 10 até 1 passo -2
  Escala.x = Velocidade
  Escala.y = Velocidade - 2
  Escala.z = Velocidade + 5
Fim
(Um conjunto equivalente é:)
Para Velocidade = 10 até 1 passo -2
  Escala = [Velocidade,Velocidade-
  2,Velocidade+5]
Fim
```

- Realiza um deslocamento horizontal, enquanto faz a cor do ator variar de vermelho vivo à ciano, subtraindo, a cada passo, 2 unidades da componente R e adicionando 2 unidades às componentes G e B:

```
Para Cor =[255,0,0] até [0,255,255] passo[-
  2,2,2]
  Posição.x = Posição.x + 5
Fim
```

4.2.1.3 Iterador: ENQUANTO...

O comando **ENQUANTO**, juntamente com o comando PARA, possibilita a repetição condicional de um conjunto de comandos. É análogo ao comando **while** de uma linguagem como C ou Pascal. Sua sintaxe é

```
Enquanto <CONDIÇÃO>
... <COMANDOS> ...
Fim
```

onde:

- **<CONDIÇÃO>** - expressão numérica que controla a lógica da repetição. A cada iteração, a expressão é avaliada. Se o seu valor numérico for diferente de zero (verdadeira), o bloco de comandos é executado. Caso contrário (a condição não foi satisfeita), a repetição é finalizada e o primeiro comando após o bloco de comandos será executado;
- **<COMANDOS>** - comandos a serem executados a cada iteração;

Exemplos da utilização de iteradores *Enquanto*:

- Enquanto a posição X do ator for inferior a 20, adiciona 2 a este atributo e gira o ator em 4 graus no eixo Z.

```
Enquanto Posição.x < 20

  Posição.x = Posição.x + 2

  Orientação.z = Orientação.z + 4

Fim
```

- Enquanto a velocidade do ator for superior a 15, subtrair 1 do valor desta e aumentar a escala do ator em 2 unidades para X, 1.5 para Y e 3 para Z:

```
Enquanto Velocidade > 15

  Velocidade = Velocidade - 1

  Escala = Escala + [1, 1.5, 3]

Fim
```

- Iteradores infinitos, isto é, que nunca cessam de repetir os comandos:

```
Enquanto 1      Enquanto Posição.x > Posição.x-1

  ...             ...

Fim             Fim

Enquanto Posição.y > -10

  ...

  Posição.y = 0;

Fim
```

- Iteradores nulos, isto é, que nunca executarão o bloco de comandos:

```
Enquanto 3>2    Enquanto Cor <> Cor

...             ...

Fim             Fim
```

4.2.1.4 Condicional: SE...

A instrução condicional **SE** tem o objetivo de prover um mecanismo de tomada de decisões durante a execução do comportamento. É análoga a um comando **if...then** de uma linguagem como C ou Pascal. A sintaxe do comando SE é:

```
Se <CONDIÇÃO> então:
  ...<COMANDOS>...
Fim
```

onde:

- <CONDIÇÃO> - expressão condicional avaliada na execução do comando. Se seu valor for diferente de zero (verdadeira), o bloco de comandos será executado. Caso contrário, será ignorado. É interessante observar que o comando SE funciona como um comando ENQUANTO de uma única iteração;
- <COMANDOS> - conjunto de comandos a ser executado se a expressão for verdadeira.

Exemplos da utilização de condicionais:

- Se a escala do ator, no eixo X, for maior que 3, então subtrai uma unidade deste atributo:

```
Se Escala.x > 3 então:
  Escala.x = Escala.x - 1
Fim
```

- Se a velocidade do ator exceder 10 unidades e se estiver subindo (isto é, direção é o eixo Y) então diminui a velocidade e inverte a direção:

```
Se Velocidade > 10 e Direção = [0,1,0] então:
  Velocidade = Velocidade - 2
  Direção = -Direção
Fim
```

- Se o "peso" (um atributo simples, por exemplo) do ator exceder 45 unidades, então executa iteração para simular uma queda. Senão, executa um movimento horizontal:

```
Se Peso > 45 então:
  Para Posição.y = Posição.y até Posição.y - 50
  Passo 1
  Fim
Fim
Se Peso <= 45 então:
  Para Posição.x = Posição.x até Posição.x + 5
  Passo 1
Fim
```

4.2.1.5 Troca de comportamento

Cada ator só pode executar um único comportamento de cada vez. Este comando serve, então, para realizar a troca do comportamento corrente para algum outro, em momentos determinados. Esses momentos podem ser simplesmente o final de uma movimentação como também podem ser causados por condições específicas, avaliadas em tempo de execução. Cabe ressaltar que o comando realmente faz uma substituição, isto é, não é apenas uma suspensão temporária do comportamento atual.

O comportamento atual não será mais executado pelo ator, a menos que este outro comportamento realize uma nova troca, de volta. A sintaxe do comando TROCA é

Troca para <COMPORTAMENTO>

onde <COMPORTAMENTO> é um identificador, um nome de comportamento válido.

Exemplos da utilização de comandos de troca:

- Enquanto o ator não atingir a posição 100 em X, avança. Ao atingí-la, troca para comportamento que eleva ator até 15 unidades em Y:

```
(Comportamento: "Avança")
Enquanto 1
  Se Posição.x > 100 então:
     Troca para Eleva
  Fim
  Posição.x = Posição.x + 5
Fim
(Comportamento: "Eleva")
Para Posição.y = 0 até 15 Passo 1
Fim
```

4.2.1.6 Comportamentos Primitivos

Comportamentos primitivos são movimentos gerados algoritmicamente, pré-programados, que podem ser utilizados dentro de uma especificação de comportamento. Cada tipo de comportamento primitivo tem uma aplicação específica, mas nada impede o usuário de combiná-los como quiser, seja em uma especificação

sequencial ou em uma composição (vide seção 4.2.1.7). É importante observar que todos os comportamentos primitivos são cinemáticos. Os seguintes comportamentos primitivos são suportados:

a) **MRU - Movimento Retilíneo Uniforme**

O MRU é um movimento em que a velocidade e a direção são constantes [ALV 79]. Esta última deve ser especificada de forma vetorial. Um ponto importante a observar é que o MRU é um movimento infinito, só pode ser cancelado por um evento que gere uma troca de comportamento (vide seção 4.2.3). A sintaxe do comando é

MRU (Dir = **<DIREÇÃO>**, *Vel =* **<VELOCIDADE>**onde:

- **<DIREÇÃO>** - vetor de direção desejado para a movimentação. Enxerga-se este parâmetro como **a direção e o sentido** da reta definida por este vetor;
- **<VELOCIDADE>** - indica a velocidade, ou seja, o deslocamento que será adicionado à posição do ator, na direção especificada. Velocidades negativas invertem o sentido do movimento. Este deslocamento é adicionado uma vez a cada passo.

Cálculo do MRU para cada passo:

1. Transforma-se o parâmetro *DIREÇÃO* em um vetor unitário (se este não o for), dividindo cada componente sua pelo seu módulo:

$$módulo = \sqrt{DIREÇÃO_x^{\,2} + DIREÇÃO_y^{\,2} + DIREÇÃO_z^{\,2}}$$

$$DIREÇÃO_x = \frac{DIREÇÃO_x}{módulo}$$

$$DIREÇÃO_y = \frac{DIREÇÃO_y}{módulo}$$

$$DIREÇÃO_z = \frac{DIREÇÃO_z}{módulo}$$

2. Multiplica-se cada componente normalizado pelo parâmetro *VELOCIDADE* e adiciona-se aos componentes da *Posição do ator:*

$$Posição_x = Posição_x + (DIREÇÃO_x \times VELOCIDADE)$$
$$Posição_y = Posição_y + (DIREÇÃO_y \times VELOCIDADE)$$
$$Posição_z = Posição_z + (DIREÇÃO_z \times VELOCIDADE)$$

Exemplos de utilização do MRU:

- Deslocamento horizontal - ao invés de se utilizar um iterador *Para* ou *Enquanto* para atualizar a posição X, pode-se fazer uso do comando MRU (figura 4.4a):

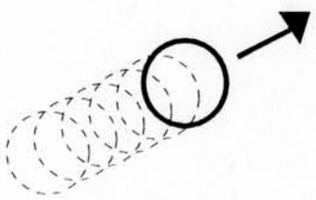
```
MRU ( Dir = [1,0,0], Vel = 1)
```

Direção = [1,0,0]
Velocidade = 1

Figura 4.4a - Movimento Retilíneo Uniforme (1)

- Deslocamento horizontal ascendente, isto é, incremento simultâneo dos componentes (da posição) X e Y (figura 4.4b):

```
MRU ( Dir = [1,1,0], Vel = 0.2)
```



Direção = [1,1,0]
Velocidade = 0.2

Figura 4.4b - Movimento Retilíneo Uniforme (2)

- Deslocamento horizontal, mas em sentido contrário (sentido negativo do eixo X) (figura 4.4c):

```
MRU ( Dir = [-1,0,0], Vel = 1)
```

ou

```
MRU ( Dir = [1, 0,0], Vel= -1)
```

Direção = [-1,0,0]
Velocidade = 1

Figura 4.4c - Movimento Retilíneo Uniforme (3)

b) **MRUV - Movimento Retilíneo Uniformemente Variado**

A idéia do MRUV é semelhante ao MRU, porém agora com um parâmetro adicional, que é a **aceleração** [ALV 79]. Esta tem como objetivo alterar a velocidade do ator em cada instante de tempo, para mais ou para menos. Com isso, o ator pode acelerar rapidamente ou vice-versa. O MRUV também é executado ininterruptamente, até que algum evento realize uma troca de comportamento. A sintaxe do MRUV é semelhante à do MRU:

MRU (Dir = <DIREÇÃO>, *Vel* = <VELOCIDADE>, *Acel* = <ACELERAÇÃO> onde:

- <DIREÇÃO> - vetor de direção desejado para a movimentação. Enxerga-se este parâmetro como **a direção e o sentido** da reta definida por este vetor;

- <VELOCIDADE> - indica a velocidade inicial, ou seja, o deslocamento que será inicialmente adicionado à posição do ator, na direção especificada. A cada passo, a este deslocamento é adicionada a aceleração;

- <ACELERAÇÃO> - indica o quanto a velocidade deve variar de instante para instante. Acelerações positivas aumentam a velocidade, negativas diminuem. Deve-se ressaltar que a aceleração pode inclusive inverter o sentido do movimento, como será demonstrado nos exemplos.

Cálculo do MRUV para cada passo:

1. Transforma-se o parâmetro *DIREÇÃO* em um vetor unitário (se este não o for), dividindo cada componente sua pelo seu módulo:

$$módulo = \sqrt{DIREÇÃO_x{}^2 + DIREÇÃO_y{}^2 + DIREÇÃO_z{}^2}$$

$$DIREÇÃO_x = \frac{DIREÇÃO_x}{módulo}$$

$$DIREÇÃO_y = \frac{DIREÇÃO_y}{módulo}$$

$$DIREÇÃO_z = \frac{DIREÇÃO_z}{módulo}$$

2. Multiplica-se cada componente normalizado pelo parâmetro *VELOCIDADE* e adiciona-se aos componentes da *Posição do ator:*

$$Posição_x = Posição_x + (DIREÇÃO_x \times VELOCIDADE)$$
$$Posição_y = Posição_y + (DIREÇÃO_y \times VELOCIDADE)$$
$$Posição_z = Posição_z + (DIREÇÃO_z \times VELOCIDADE)$$

3. Adiciona-se o parâmetro *ACELERAÇÃO ao parâmetro VELOCIDADE*:

$$VELOCIDADE = VELOCIDADE + ACELERAÇÃO$$

Exemplos de utilização do MRUV:

- Deslocamento horizontal acelerado (figura 4.5a):

```
MRUV (Dir = [1,0,0], Vel = 1, Acel = 0.5)
```

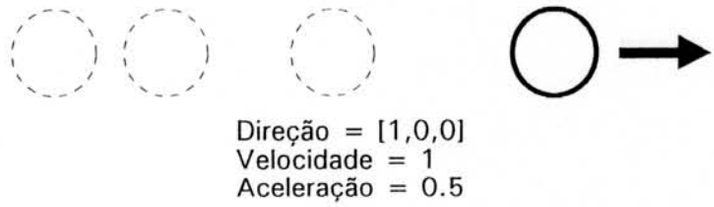


Figura 4.5a - Movimento Retilíneo Uniformemente Variado (1)

- Deslocamento horizontal acelerado, cujo sentido se inverte após 3 instantes de tempo (figura 4.5b):

```
MRUV (Dir = [1,0,0], Vel = 3, Acel = -1)
```

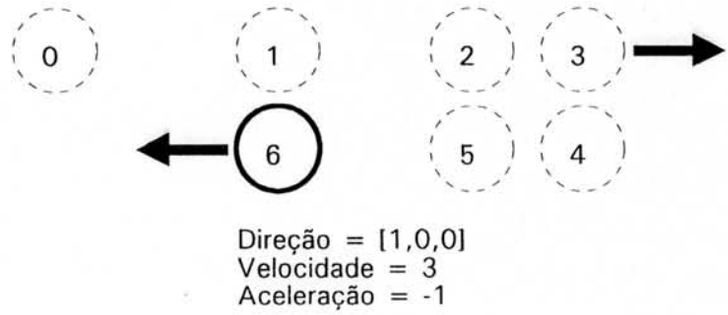


Figura 4.5b - Movimento Retilíneo Uniformemente Variado (2)

c) **MCU - Movimento Circular Uniforme**

O movimento circular uniforme [RAM 90] tem como objetivo simplificar a especificação e utilização de trajetórias circulares. Permite o controle preciso do raio da trajetória a ser criada e a orientação desta em relação aos eixos coordenados, bem como a velocidade de deslocamento. Assim como os movimentos retilíneos, também é um movimento com duração infinita. A sintaxe do MCU é

MCU (Dir = <DIREÇÃO>*, Vel =* <VELOCIDADE>*, Raio =* <RAIO> onde:

- <DIREÇÃO> - indica o eixo ao redor do qual a trajetória será gerada. Isto permite uma grande flexibilidade na orientação da trajetória circular, ou seja, o usuário não está limitado a deslocamentos nos planos (XY, XZ e YZ). Pode-se também imaginar a direção como o vetor normal ao plano sobre o qual o deslocamento se dará;
- <VELOCIDADE> - indica a velocidade angular (ω), isto é, em quantos graus a posição angular do ator é incrementada a cada instante. A posição angular representa o ângulo do arco (entre 0 e 359) no qual o ator está em relação à circunferência. A velocidade é constante;
- <RAIO> - informa o raio da circunferência imaginária que é a trajetória a ser percorrida.

Cálculo do MCU para cada passo:

1. Na primeira vez que for executar o movimento, inicializa variável *ang* com zero, representando a posição angular do ator em relação à circunferência.
2. Transforma-se o parâmetro *DIREÇÃO* em um vetor unitário (se este não o for), dividindo cada componente sua pelo seu módulo:

$$módulo = \sqrt{DIREÇÃO_x{}^2 + DIREÇÃO_y{}^2 + DIREÇÃO_z{}^2}$$

$$DIREÇÃO_x = \frac{DIREÇÃO_x}{módulo}$$

$$DIREÇÃO_y = \frac{DIREÇÃO_y}{módulo}$$

$$DIREÇÃO_z = \frac{DIREÇÃO_z}{módulo}$$

3. Calcula-se a velocidade linear (v), com base na velocidade angular (ω) e no raio (R):

$$v = \omega \times R$$

4. Calcula-se o deslocamento para as coordenadas X, Y e Z, mas como se considera que a circunferência está sobre o plano XZ (isto é, em torno do eixo Y), o deslocamento para Y será zero.

$$\Delta x = -sin(ang) \times v$$
$$\Delta y = 0$$
$$\Delta z = \cos(ang) \times v$$

5. Até aqui tem-se o deslocamento específico para uma circunferência sobre o plano XZ. A fim de se orientar corretamente esta circunferência, será simulada uma rotação do eixo (isto é, da direção), semelhante àquela que é realizada no processo de visualização tridimensional, para conversão de sistemas de referência [FOL 90]. Esta rotação, na verdade, será executada sobre os deslocamentos já calculados. Primeiramente, deve-se calcular o **eixo de rotação**, através de um produto vetorial entre o eixo Y ([0,1,0]) e a direção especificada pelo usuário:

$$\vec{e} = [0 \quad 1 \quad 0] \times DIREÇÃO$$

6. A seguir, calcula-se o ângulo de rotação necessário (β) para se levar um ponto do eixo Y até o eixo especificado. Esse ângulo é calculado através do produto escalar entre os dois vetores:

$$\beta = \arccos\left(\frac{[0 \quad 1 \quad 0] \cdot DIREÇÃO}{\left|[0 \quad 1 \quad 0] \cdot DIREÇÃO\right|}\right)$$

7. Através da matriz de rotação de um ponto em torno de um eixo (M_e) [ROG 90], rotaciona-se os deslocamentos já calculados em torno do eixo obtido no item 5, com o ângulo de rotação obtido no item 6:

$$[\Delta x \quad \Delta y \quad \Delta z] = [\Delta x \quad \Delta y \quad \Delta z] \times M_e$$

8. Acrescenta-se à posição do ator os deslocamentos, já orientados corretamente:

$$Posição_x = Posição_x + \Delta x$$
$$Posição_y = Posição_y + \Delta y$$
$$Posição_z = Posição_z + \Delta z$$

As figuras 4.6a, 4.6b e 4.6c apresentam vários exemplos de MCU, cada um com uma orientação distinta:

```
MCU (Dir = [0,0,-1], Vel = 15, Raio = 10)
```

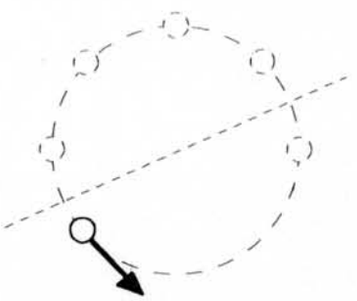


Figura 4.6a - Movimento Circular Uniforme (1)

```
MCU (Dir = [0,1,0], Vel = 15, Raio = 10)
```

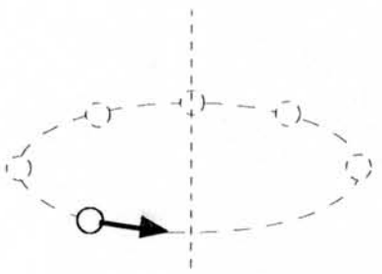


Figura 4.6b - Movimento Circular Uniforme (2)

```
MCU (Dir = [-1,-1,0], Vel = 15, Raio = 10)
```

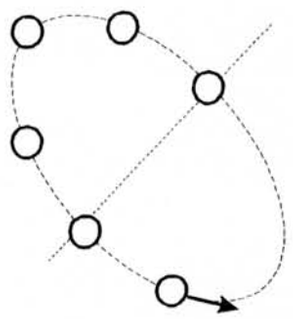


Figura 4.6c - Movimento Circular Uniforme (3)

d) **Movimento em Espiral**

O movimento em espiral é uma extensão do movimento circular uniforme. A trajetória gerada é uma circunferência cujo raio diminui ou aumenta um pouco a cada passo. Este decréscimo/acréscimo ao raio é que gera a trajetória em espiral. O movimento em espiral utiliza os mesmos parâmetros que o MCU e o parâmetro *Passo*, que controla o quanto o raio diminui/aumenta a cada instante. É um movimento finito, pois quando o raio atingir o valor mínimo/máximo (parâmetro *Final*) será finalizado. Sua sintaxe é

ESPIRAL (Dir = <DIREÇÃO>, *Vel =* <VELOCIDADE>, *Raio =* <RAIO>, *Final =* <MÍNIMO>, *Passo =* <PASSO>onde:

- <DIREÇÃO> - tem o mesmo significado empregado no MCU, isto é, a orientação da trajetória gerada;
- <VELOCIDADE> - como no MCU, indica a velocidade angular. Porém, a medida que o raio diminui/aumenta, o deslocamento efetivo se tornará cada vez menor/maior, pois o ator terá que percorrer a mesma "distância angular" no mesmo tempo;
- <RAIO> - informa o raio inicial da circunferência imaginária que é a trajetória a ser percorrida;

- <FINAL> - informa o raio final (mínimo ou máximo) da circunferência imaginária que é a trajetória a ser percorrida;
- <PASSO> - informa o valor que será acrescentado ao raio a cada passo. Se o passo for negativo, o raio diminuirá cada vez mais e o parâmetro *Final* indicará o raio mínimo. Se o passo for positivo, o raio aumentará cada vez mais e o parâmetro *Final* indicará o raio máximo.

Observação: O movimento espiral é calculado exatamente como o MCU (vide item "c"), porém no final o raio é atualizado com o passo, conforme mencionado acima.

As figuras 4.7a, 4.7b e 4.7c apresentam vários exemplos de movimentos em espiral, cada um com um passo diferente:

```
ESPIRAL (Dir = [0,0,-1], Vel = 15, Raio = 20,
  Final = 5, Passo = -0.5 )
```

Figura 4.7a - Movimento em Espiral (1)

```
ESPIRAL (Dir = [0,0,-1], Vel = 15, Raio = 5,
  Final = 20, Passo = 1 )
```

Figura 4.7b - Movimento em Espiral (2)

```
ESPIRAL (Dir = [0,0,-1], Vel = 15, Raio = 1,
  Final = 20, Passo = 0.5 )
```

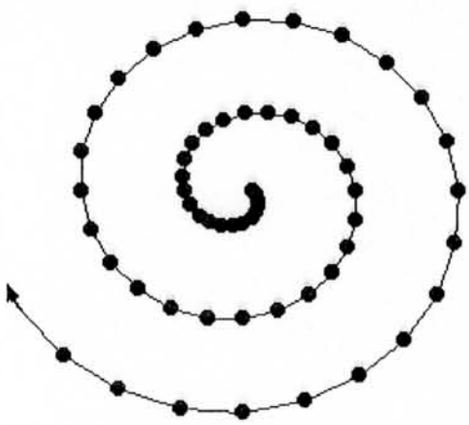


Figura 4.7c - Movimento em Espiral (3)

e) **Movimento de Lançamento Oblíquo**

O movimento de lançamento oblíquo [RAM 90] visa simular a trajetória percorrida por um projétil disparado do solo (ou não). Esta trajetória, como se sabe, é uma parábola, cuja aparência depende de três fatores: o *ângulo* que a direção inicial de lançamento forma com o solo, a *velocidade inicial* do projétil e a *aceleração gravitacional* sofrida por este durante o vôo. Não é considerado o atrito do ar. O movimento de lançamento termina quando o corpo atingir novamente o solo. Sintaxe:

LANÇAMENTO (Giro = <GIRO>, *Vel* = <VELOCIDADE>, *Ang* = <ÂNGULO>, *G* = <G>onde:

- <GIRO> - representa a direção de lançamento em torno do eixo Y. É um ângulo que varia entre 0 (coincidindo com o eixo X) e 359 graus;
- <VELOCIDADE> - indica a velocidade inicial de lançamento. Quanto maior, mais chances um ator tem de atingir uma maior altura ou percorrer uma maior distância;
- <ÂNGULO> - informa o ângulo que a direção de lançamento forma com o solo. Na prática, os parâmetros *giro* e *ângulo* juntos compõem a direção de lançamento (figura 4.8a). O ângulo também contribui para uma maior altitude ou maior alcance;
- <G> - aceleração gravitacional, utilizada a cada instante para compor a direção resultante, diminuindo pouco a pouco a velocidade vertical do ator e eventualmente levando este de volta ao solo. Quanto maior, mais rápida será a queda do ator e menor será sua ascensão;

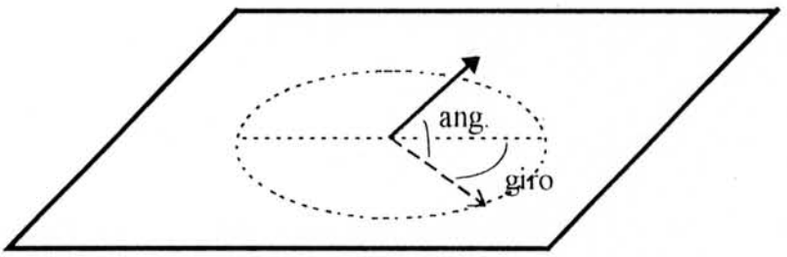


Figura 4.8a - Parâmetros *giro* e *ângulo* no movimento de lançamento oblíquo

Cálculo do movimento de lançamento oblíquo a cada passo:

1. Com base no ângulo de giro (α), no ângulo de lançamento (β) e na velocidade inicial (v), calcula-se primeiramente a direção de lançamento, representada por Δx, Δy e Δz:

$$\Delta x = v \times \cos(\alpha) \times \cos(\beta)$$
$$\Delta y = v \times sin(\beta)$$
$$\Delta z = v \times sin(\alpha) \times \cos(\beta)$$

2. A cada iteração, adiciona-se à posição do ator os deslocamentos atuais:

$$Posição_x = Posição_x + \Delta x$$
$$Posição_y = Posição_y + \Delta y$$
$$Posição_z = Posição_z + \Delta z$$

3. A seguir, subtrai-se do deslocamento vertical a aceleração gravitacional especificada. Isto faz com que o ator suba cada vez menos e eventualmente começará a descer:

$$\Delta y = \Delta y - G$$

4. O movimento de lançamento oblíquo termina quando o ator chegar ao solo, isto é, quando houver contato de sua *bounding box* com o plano Y = 0.

As figuras 4.8b, 4.8c e 4.8d apresentam vários exemplos de movimentos de lançamento oblíquo, cada um com parâmetros diferentes:

```
LANÇAMENTO (Giro = 0, Vel = 20, Ang = 20,  G =
5)
```

Figura 4.8b - Movimento de Lançamento Oblíquo (1)

```
LANÇAMENTO (Giro = 0, Vel = 30, Ang = 45,  G =
9.8)
```

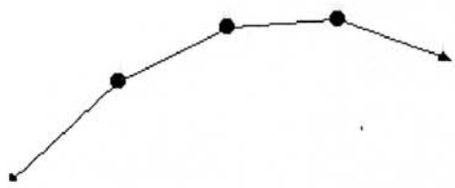


Figura 4.8c - Movimento de Lançamento Oblíquo (2)

```
LANÇAMENTO (Giro = 0, Vel = 20, Ang = 80,  G =
3)
```

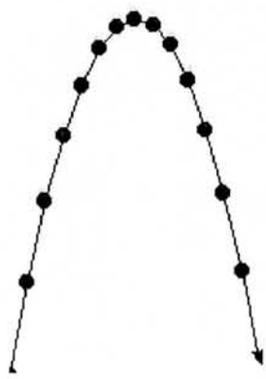


Figura 4.8d - Movimento de Lançamento Oblíquo (3)

4.2.1.7 Composições de Comportamentos

O comando de composição permite a especificação de hierarquias de comportamento, combinando primitivos e comandos de atribuição. Isto é, pode-se criar movimentos compostos de um ou mais primitivos, alterando-se atributos diretamente no decorrer destes. Sua sintaxe é:

```
Composição
  ...<COMANDOS>...
Fim
```

<COMANDOS> compreende um conjunto de comandos de atribuição ou comandos de movimento primitivo (MRU, MRUV, MCU, ESPIRAL, LANÇAMENTO). A execução dos comandos será realizada de forma hierárquica, isto é, a cada passo serão calculadas as posições tomando-se como base a ordem em que os comandos foram especificados.

Exemplos da utilização de composições:

- Movimento horizontal (MRU), no qual o ator descreve uma circunferência em torno do eixo X (MCU) (figura 4.9a)

```
Composição
  MRU (Dir = [1,0,0], Vel = 5)
  MCU (Dir = [1,0,0], Vel = 15, Raio = 15)
Fim
```

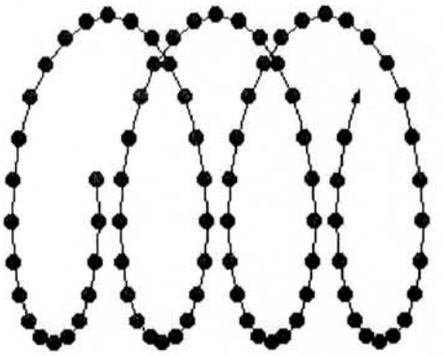


Figura 4.9a - Composição de MRU e MCU

- Movimento em espiral decrescente no plano XZ, onde o ator sofre uma ascensão (coordenada Y) e simultaneamente gira em torno de seu eixo Z (figura 4.9b)

```
Composição

  ESPIRAL (Dir = [0,1,0], Vel = 15, Raio = 20,
  Final = 2, Passo = -0.5)

  Posição.y = Posição.y + 1

  Orientação.z = Orientação.z + 8

Fim
```

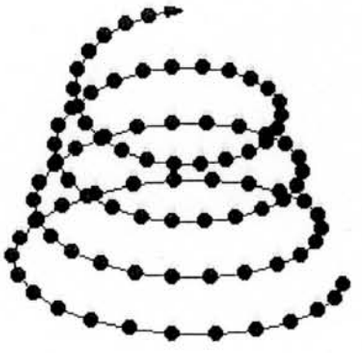


Figura 4.9b - Composição de ESPIRAL e atribuições

4.2.1.8 Envio de mensagens

O comando de envio de mensagens é fundamental, pois permite a comunicação efetiva entre atores. Uma *mensagem* consiste em um conjunto de parâmetros, que podem ser valores numéricos (constantes) ou atributos. Para efeitos de tornar a comunicação mais inteligível para o usuário, podem ser enviadas *strings* de caracteres também. A sintaxe do comando é

Enviar: **<MENSAGEM>** *para* **<CLASSE>**

onde:

- **<MENSAGEM>** - representa a mensagem a ser enviada, composta por um conjunto de parâmetros separados por vírgulas. Como mencionado anteriormente, estes parâmetros podem ser constantes, atributos ou *strings*;
- **<CLASSE>** - indica a **classe de atores** que receberá a mensagem. É importante observar que todos os atores desta classe receberão a mensagem, uma vez que na descrição de comportamento não se tem conhecimento de quantos atores de cada classe foram criados, ou seja, não se pode identificar um ator específico. Pode-se especificar a classe **TODOS**, representando todos os atores existentes no sistema, independentemente de suas classes;

O tratamento de uma mensagem recebida por um ator só ocorre se o comportamento associado a este contiver um evento de recebimento de mensagens (vide seção 4.2.3.4). Este tratamento dependerá de como o evento foi descrito para o sistema. Obviamente, o ator que enviou a mensagem não a receberá, mesmo que tenha enviado para atores que sejam da mesma classe que a sua.

Exemplos de comunicação entre atores (envio de mensagens):

- Envia uma mensagem contendo a *string* "Aviso" e a posição atual do ator para todos os atores da classe *Veículos*

```
Enviar: "Aviso",Posição para Veículos
```

- Envia uma mensagem contendo a *string* "Atenção" e a direção e velocidade atuais do ator para todos os atores da classe *Aves*

```
Enviar: Direção,Velocidade,"Atenção" para Aves
```

4.2.1.9 Criação de ator

Este comando permite a criação controlada um novo ator de uma determinada classe de atores. Permite, por exemplo, que um novo ator surja em determinado instante da animação, quando alguma condição for satisfeita. O posicionamento do novo ator criado sempre se dá em relação à posição atual do ator que o criou. Nenhum de seus outros atributos é inicializado. A sintaxe de um comando de criação é

```
Cria <CLASSE> em <DIREÇÃO>,<DISTÂNCIA>
  executando <COMPORTAMENTO>
```

onde:

- <CLASSE> - indica o tipo de ator que será criado, isto é, a classe do novo ator. Note-se que esta classe não precisa ser necessariamente a mesma do ator que está gerando o novo ator;
- <DIREÇÃO> - especifica em que direção (vetorial) o novo ator será criado. A posição inicial do novo ator é sempre relativa à posição daquele que cria;
- <DISTÂNCIA> - indica a distância do novo ator em relação ao criador. Este parâmetro é utilizado, juntamente com o parâmetro *direção* e a posição do ator que está criando, para calcular a posição inicial do novo ator;
- <COMPORTAMENTO> - informa o comportamento inicial que o novo ator deve executar. O início deste comportamento pode ser utilizado para inicializar os demais atributos deste novo ator, uma vez que só a sua posição é ajustada automaticamente. O ator que

está criando pode, também, enviar uma mensagem para o recém-criado, passando alguma informação ou instrução adicional.

Exemplos de criação de novos atores:

```
Cria Veículos em [0,1,0],5 executando Avanço

Cria Aves em [1,1,0],3.2 executando Vôo

Cria Cubos em [-0.4,0.2,0.1],8 executando
  Espera
```

4.2.1.10 Eliminação de ator

Existem momentos em uma animação em que é necessário que um ator desapareça, deixe de participar da animação. O comando *Elimina* faz com que o comportamento sendo executado seja cancelado e ator deixe de existir. É um comando direto, sem parâmetros, exceto quando utilizado como resposta de um evento (vide seção 4.2.3):

```
Elimina ator
```

4.2.2 Especificação de Restrições

Uma estrutura de controle adicional que pode ser utilizada na especificação de um comportamento é a **restrição**. Essa utilização não é obrigatória, pode-se modelar comportamentos somente com a descrição procedimental se assim se desejar. Porém, como se verá a seguir, as restrições são um componente importante para alguns tipos de comportamento. As restrições funcionam como indicações de condições que devem ser respeitadas no decorrer do movimento. Estas condições se dividem em três tipos: **perseguir, aproximar-se e evitar aproximar-se de uma classe de atores**, ou seja, de qualquer ator que seja desta classe. A cada restrição também é

associada uma **prioridade**. Esta indica a ordem em que as restrições devem ser avaliadas; quanto maior a prioridade, mais importante é a restrição.

4.2.2.1 Perseguir

A restrição de perseguição faz com que o ator tente se movimentar na direção de todos atores que sejam da classe desejada. Devido à forma como é implementada (vide seção 4.4.2), deve-se tomar cuidado para não gerar situações conflitantes, como, por exemplo, onde um ator tem que se aproximar de dois outros que estejam à mesma distância e em lados opostos. Esse tipo de situação causa uma anulação do movimento gerado pela restrição, pois o ator tem que se aproximar dos dois, mas não pode favorecer nem um nem o outro. Esta restrição é especificada da seguinte forma:

```
<PRIORIDADE> - Perseguir: <CLASSE>
```

onde:

- `<PRIORIDADE>` - indica a prioridade desta restrição em relação às demais. Se for a única, a prioridade é irrelevante;
- `<CLASSE>` - informa a classe que deve ser perseguida pelo ator. Todos os atores desta classe serão considerados. Pode-se indicar a classe **TODOS** para o sistema considerar todos os atores presentes.

4.2.2.2 Aproximar

A restrição de aproximação tem o mesmo sentido da restrição de perseguição, porém agora acrescenta-se um elemento adicional a ser considerado, que é a distância mínima a ser mantida entre o ator e os alvos. Novamente, deve-se procurar evitar as situações conflitantes. Esta restrição é especificada da seguinte forma:

<PRIORIDADE> -*Aproximar:* <CLASSE> *até* <DISTÂNCIA>

onde:

- <PRIORIDADE> - indica a prioridade desta restrição em relação às demais. Se for a única, a prioridade é irrelevante;
- <CLASSE> - informa a classe cujos atores devem ser aproximados pelo ator. Todos os atores desta classe serão considerados. Assim como na restrição de perseguição, pode-se indicar a classe **TODOS**;
- <DISTÂNCIA> - especifica a distância mínima que deve ser mantida entre o ator e os demais. Esta distância será mantida na medida do possível, pois dependendo do comportamento e de outras restrições, pode não ser possível mantê-la.

4.2.2.3 Evitar

Esta restrição tem o sentido inverso da restrição de aproximação, pois tenta evitar que o ator siga na direção dos atores da classe especificada. É importante observar que não é garantido que esta restrição **desvie** o ator de seu rumo, mas simplesmente tente evitar ao máximo a aproximação. Situações conflitantes também devem ser cautelosamente consideradas. A distância é interpretada de forma diferente, conforme exemplificado abaixo.

<PRIORIDADE> - *Evitar:* <CLASSE> *até* <DISTÂNCIA>

onde:

- <PRIORIDADE> - indica a prioridade desta restrição em relação às demais. Se for a única, a prioridade é irrelevante;

- <CLASSE> - informa a classe cujos atores devem ser evitados pelo ator. Todos os atores desta classe serão considerados. podendo-se informar a classe **TODOS**;

- <DISTÂNCIA> - especifica a distância máxima na qual a restrição será considerada. Isso especifica, na prática, a área de influência de um ator em relação a outro. Com a escolha correta da distância, pode-se evitar que atores que estejam muito distantes sejam evitados, pois não interferem realmente no movimento do ator. Esta interpretação da distância ajuda a evitar conflitos.

4.2.3 Tratamento de Eventos

Os eventos são a terceira e última estrutura de controle que define um comportamento. Nada mais são do que condições especiais que exigem a atenção do ator: proximidade de outro ator, colisão com outro ator, surgimento de um ator ou recebimento de uma mensagem. Cada evento, caso ocorra, requer uma resposta, isto é, o que deve ser feito para tratar a situação que se apresentou. Esta resposta pode ser: uma atribuição, a troca do comportamento atual, o envio de uma mensagem, a criação de um novo ator e a eliminação de um ator. Assim como nas restrições, os eventos são avaliados de acordo com a prioridade de cada um.

O tratamento de eventos sob um formalismo mais rigoroso não é absolutamente essencial para o protótipo, uma vez que buscou-se simplificar ao máximo o funcionamento do sistema. Recomenda-se a consulta ao artigo de Kalra e Barr [KAL 92], onde uma forma de animação orientada a eventos descrita formalmente é apresentada.

4.2.3.1 Proximidade

O evento de proximidade visa detectar se o ator está muito próximo de algum outro, da classe especificada. Também funciona como um recurso para se evitar colisões, por exemplo, no caso da restrição de evitar se mostrar insatisfatória. Deve ser informada a distância mínima a partir da qual o evento deve ocorrer. Descrição:

```
<PRI> - Proximidade: <CLASSE> até <DISTÂNCIA> -
  - <RESPOSTA>
```

onde:

- <PRI> - indica a prioridade deste evento em relação aos demais. Se for o único, a prioridade é irrelevante;
- <CLASSE> - informa a classe cujos atores podem gerar eventos para este ator. Qualquer ator desta classe pode ser considerado. Pode-se especificar a classe **QUALQUER**, que representa qualquer classe. Dessa forma, qualquer ator de qualquer classe pode gerar o evento;
- <DISTÂNCIA> - especifica a distância mínima a partir da qual o evento será gerado. Este parâmetro evita que eventos de proximidade sejam gerados entre atores muito distantes;
- <RESPOSTA> - determina a resposta para o evento, caso este ocorra. Se a resposta for a eliminação de um ator, deve ser especificado se o ator que deve ser eliminado é o que sofreu o evento (*Elimina ator*) ou o que causou o evento (*Elimina alvo*).

4.2.3.2 Colisão

O evento de detecção de colisão tem como objetivo determinar se um ator colidiu com outro, isto é, se houve contato entre eles. Este contato é testado verificando-se se houve intersecção entre as *bounding boxes* dos atores em questão. O

algoritmo não é preciso, porém mais simples. Como exemplo de um algoritmo alternativo com mais precisão, cita-se [MOO 88]. O evento permite tratar colisões entre atores como se desejar, inclusive ignorando-as. É especificado da seguinte forma:

```
<PRI> - Colisão: <CLASSE> -- <RESPOSTA>
```

onde:

- <PRI> - indica a prioridade deste evento em relação aos demais. Se for o único, a prioridade é irrelevante;
- <CLASSE> - informa a classe cujos atores podem gerar eventos para este ator. Sua interpretação é idêntica à do evento de proximidade;
- <RESPOSTA> - determina a resposta ao evento, análoga ao evento de proximidade.

4.2.3.3 Presença

Este tipo de evento só é gerado no momento em que um ator é criado. Detecta, portanto, o surgimento de novos atores. Pode ser restrito a uma classe específica ou ser genérico, com a utilização da classe fictícia TODOS. Definido como:

```
<PRI> - Presença: <CLASSE> -- <RESPOSTA>
```

onde:

- <PRI> - indica a prioridade deste evento em relação aos demais. Se for o único, a prioridade é irrelevante;
- <CLASSE> - informa a classe cujos atores podem gerar eventos para este ator. Sua interpretação é idêntica à dos eventos anteriores;
- <RESPOSTA> - determina a resposta ao evento, análoga aos eventos anteriores.

4.2.3.4 Recepção de mensagem

Para que um determinado ator possa entender as mensagens que recebe, deve se especificar no seu comportamento um evento de recepção de mensagem. Este tem a finalidade de descrever como a mensagem deve ser interpretada e qual a ação (resposta) que deve ser tomada se a mensagem desejada for recebida. Naturalmente, cada mensagem diferente deve ser tratada em um evento separado. Um evento de recepção é definido como:

```
<PRI> - Mensagem: <MENS> de <CLASSE> --
  <RESPOSTA>
```

onde:

- `<PRI>` - indica a prioridade deste evento em relação aos demais. Se for o único, a prioridade é irrelevante;
- `<MENS>` - especifica a mensagem que deve ser recebida pelo ator para o evento ocorrer. É importante ressaltar que se a especificação da mensagem recebida representar uma porção da mensagem original, isto é, se contiver um subconjunto dos parâmetros recebidos (na ordem correta), o evento será gerado;
- `<CLASSE>` - informa a classe cujos atores podem gerar eventos para este ator. Sua interpretação é idêntica à dos eventos anteriores;
- `<RESPOSTA>` - determina a resposta ao evento, análoga aos eventos anteriores.

Exemplo: um ator envia a mensagem **"Atenção",Posição,Escala** para outros. Se um desses outros apresentar em seu comportamento um evento de recebimento de mensagem, cujo parâmetro *mensagem* seja **"Atenção",Posição**, o evento ocorrerá.

4.2.4 Movimento Complementar

Este movimento adicional é gerado se ambos os atributos *Direção* e *Velocidade* de um ator não forem nulos. O objetivo é prover um mecanismo para simplificar a especificação de certos tipos de movimentações. O resultado prático deste movimento complementar é a adição de valores a cada uma das coordenadas armazenadas no atributo *Posição* do ator. Estes valores são calculados da seguinte maneira:

1. Transforma-se o atributo *Direção* em um vetor unitário (se este não o for), dividindo cada componente sua pelo seu módulo

$$módulo = \sqrt{Direção_x^{\,2} + Direção_y^{\,2} + Direção_z^{\,2}}$$

$$Direção_x = \frac{Direção_x}{módulo}$$

$$Direção_y = \frac{Direção_y}{módulo}$$

$$Direção_z = \frac{Direção_z}{módulo}$$

2. Multiplica-se cada componente normalizado pelo atributo *Velocidade* e adiciona-se aos componentes da *Posição*

$$Posição_x = Posição_x + (Direção_x \times Velocidade)$$
$$Posição_y = Posição_y + (Direção_y \times Velocidade)$$
$$Posição_z = Posição_z + (Direção_z \times Velocidade)$$

Este movimento complementar é, na verdade, um MRU (vide seção 4.2.1.6, item "c"), pois tanto a direção quanto a velocidade são constantes. É claro que durante o comportamento pode-se variar tanto um quanto o outro e é nesta propriedade que reside a sua flexibilidade.

4.3 Cenas

Uma **cena** especifica a disposição inicial dos atores e seus comportamentos, bem como a posição de onde eles serão observados. É, portanto, um conjunto de atores e parâmetros de visualização. Estes parâmetros de visualização são utilizados na câmera sintética (vide seção 4.3.2), a fim de gerar a imagem da cena.

4.3.1 Atores

Cada ator criado é, na verdade, uma **instância** de uma classe de atores, possuindo todos os atributos definidos nesta última. Os atributos de cada ator, porém, podem ser inicializados pelo usuário como for desejado. Um ator será representado na visualização como o objeto tridimensional descrito pela geometria de sua classe. Esta não pode ser alterada, mas parâmetros como rotação, escala e cor podem contribuir para diferenciar o ator de outros de sua classe (obviamente, se isto for necessário).

Um ator criado recebe uma identificação, que é o nome de sua classe acrescido de um índice, que o diferencia de outros atores da mesma classe. Exemplo: considerando-se a classe **Cubos**, os atores desta classe serão denominados **Cubo #1, Cubo #2, Cubo #3,** ...

Para um ator participar efetivamente da animação, é necessário que algum comportamento seja associado a ele. Dessa forma, qualquer comportamento definido para a sua classe (ou para as classes de onde sua classe foi derivada) pode ser utilizado.

4.3.2 Câmera Sintética

Cada cena a ser visualizada deve prover ao sistema a informação da posição e da orientação do observador. Como a visualização é feita através de um

pacote de câmera sintética [FOL 90], deve também estar presente a informação de *zoom* (ou abertura da lente).

Todos os atores ativos (isto é, que não foram cancelados pelo comando *Elimina*) são desenhados, representados pela geometria de sua classe, nas coordenadas presentes em seu atributo *Posição*, em escala definida por seu atributo *Escala* e em orientação definida por seu atributo *Orientação*. A cor de desenho é aquela armazenada no atributo *Cor* do ator.

4.4 Avaliação do Comportamento

A partir dos elementos que compõem uma cena, isto é, atores e seus respectivos comportamentos, é possível descrever agora como se dá efetivamente a avaliação de comportamento.

Primeiramente, deve-se imaginar cada ator como uma unidade de processamento independente, executando um único processo, que é o seu comportamento corrente. A avaliação do comportamento na cana é, teoricamente, uma simulação de vários processos sendo executados simultaneamente, isto é, um sistema de multiprogramação, onde cada elemento de processamento executa o seu processo, independentemente dos demais.

A cada passo da simulação, isto é, para cada quadro de animação a ser gerado, é necessário avaliar o comportamento de cada ator, a fim que seja gerado o seu movimento. Se o sistema fosse realmente multiprogramável, a avaliação se daria simplesmente executando-se o mesmo algoritmo em cada elemento de processamento. Como o sistema foi implementado em um ambiente monoprogramado, fez-se necessária a utilização de um controle de multitarefa, onde vários processos são executados concorrentemente, ou seja, um de cada vez.

A avaliação de comportamento é realizada para cada ator, de acordo com os seguintes passos:

1. Execução do comando corrente

2. Avaliação das restrições

3. Cálculo do movimento complementar (se houver)

4. Tratamento dos eventos

4.4.1 Execução do comando corrente

Quando um comportamento é associado a um ator, na definição de uma cena, é criada juntamente com este último uma estrutura de controle, a qual armazena o estado de execução do comportamento. A informação mais importante armazenada no estado é a **posição,** na lista de instruções do comportamento, que contém o comando a ser executado (ou que está sendo executado) pelo ator. Esta posição é inicializada com zero, indicando o primeiro comando da lista de instruções. Por seu significado se assemelhar a um contador de programa (ou *program counter*), doravante será denominada *BC (behavior counter).* Também são armazenadas estruturas adicionais, a fim de controlar a execução de iteradores, composições e primitivos.

É importante observar que cada ator, mesmo que compartilhe a descrição do comportamento com outro, armazena o BC independentemente dos demais, reforçando a idéia de processos concorrentes. Porém, resta ainda a explicação de como é controlada a execução dos comandos, pois não faz sentido executar de uma só vez toda a lista de instruções do comportamento de cada ator.

Com o objetivo de solucionar esta questão, convencionou-se separar os comandos em dois grupos:

- comandos imediatos
- comandos de iteração

Comandos imediatos são aqueles que podem ser executados diretamente, tais como: atribuição, condicional, troca de comportamento, envio de mensagem, criação e eliminação de ator. Este tipo de comando é executado continuamente, isto é, enquanto forem encontrados comandos imediatos, continua-se a avaliar o comportamento do ator.

Exemplo: a seguinte seqüência de comandos será executada de uma só vez.

```
Posição.x = Posição.x + 2

Orientação.z = Orientação.z - 8

Se Posição.x > 75 então:

  Envia "Atenção",Posição para Alvos

Fim
```

Já comandos de iteração são aqueles que podem criar uma repetição, isto é, definem um bloco de comandos que deve ser executado continuamente. Os comandos de iteração são: iteradores Para... e Enquanto..., comportamentos primitivos e composições. Para este tipo de comando, o sistema executa **uma única vez** o bloco de comandos que o compõe, para evitar que o ator utilize um excesso de tempo de processamento. Somente no próximo quadro gerado é que o bloco de comandos será novamente executado.

Como exemplo, considere-se a seguinte lista de comandos: a cada quadro, todos os comandos no interior do iterador Para... serão executados, pois são imediatos.

```
Para Posição.x = Posição.x até Posição.x + 10

  Orientação.z = Orientação.z - 8

  Se Posição.x > 75 então:

     Envia "Atenção",Posição para Alvos
```

```
    Fim

Fim
```

4.4.2 Avaliação das restrições

As restrições, conforme mencionado na seção 4.2.2, são consideradas de acordo com a prioridade de cada uma. O objetivo deste processo todo é gerar um vetor de deslocamento resultante, que será posteriormente combinado com o atributo *Direção* do ator para gerar o movimento final.

O processamento das restrições segue os seguintes passos:

1. Inicializa o vetor de deslocamento resultante ($\vec{R}$) com um vetor nulo;
2. Combina este vetor com o resultante de cada restrição especificada;
3. Normaliza o vetor resultante final e combina este ao atributo *Direção* do ator, realizando uma soma vetorial:

$$Direção_x = Direção_x + \vec{R}_x$$
$$Direção_y = Direção_y + \vec{R}_y$$
$$Direção_z = Direção_z + \vec{R}_z$$

4.4.2.1 Cálculo da resultante para restrições de perseguição/aproximação

Este cálculo é realizado de forma quase idêntica para as restrições de perseguição e aproximação. A única diferença é que a restrição considera a distância mínima a ser mantida entre o ator e os demais. A resultante é calculada da seguinte forma:

1. Inicializa resultante local ($\vec{r}$) com um vetor nulo;

2. Para cada ator (denominado *alvo*) que seja da classe especificada na restrição (ou para todos os atores, se for utilizada a classe especial TODOS):

   2.1 Calcula distância (*d*) entre o ator atual e o alvo;

   2.2 Calcula um vetor ($\vec{a}$), da posição do ator atual até o alvo;

   2.3. Se restrição for aproximação, verifica se *d* é menor ou igual à distância mínima especificada na restrição. Em caso afirmativo, não se deve aproximar mais o ator do alvo, pois este já está suficientemente próximo. O processo pula para o próximo alvo (passo 2).

   2.4. Atualiza a resultante local, usando a distância como peso para ponderação. Dessa forma, quanto maior a distância, mais o ator será atraído para este alvo, o que condiz com a definição da restrição, pois se o ator está próximo de um alvo e distante de outro alvo, é mais importante ele tentar se aproximar daquele que está mais distante, uma vez que esta condição com o alvo mais próximo já foi satisfeita.

$$\vec{r}_x = \vec{r}_x + \vec{a}_x \times d$$
$$\vec{r}_y = \vec{r}_y + \vec{a}_y \times d$$
$$\vec{r}_z = \vec{r}_z + \vec{a}_z \times d$$

3. Normaliza a resultante local

4. Atualiza a resultante global, utilizando a **prioridade** da restrição como um ponderador. Isso faz com que as restrições que tenham maior prioridade contribuam proporcionalmente mais para a resultante final:

$$\vec{R}_x = \vec{R}_x + \vec{r}_x \times prioridade$$
$$\vec{R}_y = \vec{R}_y + \vec{r}_y \times prioridade$$
$$\vec{R}_z = \vec{R}_z + \vec{r}_z \times prioridade$$

4.4.2.2 Cálculo da resultante para restrições do tipo evitar

Já as restrições de fuga (evitar) realizam um tratamento diferente da distância, pois esta agora representa a distância máxima em que a restrição deve ser avaliada. A resultante agora é calculada da seguinte forma:

1. Inicializa resultante local ($\vec{r}$) com um vetor nulo;

2. Para cada ator (denominado *alvo*) que seja da classe especificada na restrição (ou para todos os atores, se for utilizada a classe especial TODOS):

   2.1 Calcula distância (*d*) entre o ator atual e o alvo;

   2.2 Calcula um vetor ($\vec{a}$), da posição do alvo até o ator atual. Como agora deseja-se evitar o alvo - e não aproximar-se - este vetor é calculado de forma inversa;

   2.3. Se *d* for superior à distância máxima especificada na restrição, o alvo está suficientemente distante do ator e não deve ser considerado. Avalia-se então o próximo alvo (passo 2).

   2.4. Calcula-se o peso para ponderação, computando-se a razão entre a distância máxima especificada na restrição e a distância calculada (*d*). Isso faz com que a importância deste alvo aumente à medida que o ator se aproxima dele, o que é lógico, pois quanto mais próximos, mais necessária é a fuga.

$$peso = \frac{distância}{d}$$

   2.5 Atualiza-se a resultante local, utilizando o peso calculado no passo anterior e a direção obtida no passo 2.2. Como a direção agora aponta na direção oposta ao alvo, o que ocorre efetivamente é que o ator tentará evitá-lo.

$$\vec{r}_x = \vec{r}_x + \vec{a}_x \times peso$$
$$\vec{r}_y = \vec{r}_y + \vec{a}_y \times peso$$
$$\vec{r}_z = \vec{r}_z + \vec{a}_z \times peso$$

3. Normaliza a resultante local

4. Atualiza a resultante global, utilizando a **prioridade** da restrição como um ponderador. Isso faz com que as restrições que tenham uma prioridade maior contribuam proporcionalmente mais para o resultante final:

$$\vec{R}_x = \vec{R}_x + \vec{r}_x \times prioridade$$
$$\vec{R}_y = \vec{R}_y + \vec{r}_y \times prioridade$$
$$\vec{R}_z = \vec{R}_z + \vec{r}_z \times prioridade$$

4.4.3 Cálculo do movimento complementar

O movimento complementar, calculado conforme mencionado na seção 4.2.2, é computado somente após a execução dos comandos e avaliação das restrições. A avaliação das restrições pode alterar o atributo *Direção*, pois como visto na seção 4.4.2, este é combinado com o vetor resultante, obtido das restrições especificadas.

Então, na prática, a **execução** das restrições só ocorre agora, quando for efetivamente calculado o deslocamento da *Posição* em função dos atributos *Direção* e *Velocidade*.

4.4.4 Tratamento dos eventos

Os eventos são tratados como **exceções** à execução normal de um comportamento, isto é, em qualquer momento um pode ocorrer. Se a resposta do evento não for uma troca de comportamento ou uma eliminação do próprio ator, o comportamento continua a sua execução normal logo após a execução desta resposta. Se for uma troca de comportamento, o comportamento atual é cancelado e o especificado será executado. Se for uma eliminação do próprio ator, este é simplesmente marcado como cancelado e não será mais considerado pelo sistema.

5. PROTÓTIPO DE AMBIENTE PARA A ABORDAGEM PROPOSTA

5.1 Visão Geral do Ambiente

A fim de comprovar a funcionalidade da abordagem, foi desenvolvido um protótipo, com as seguintes características:

- utilização de uma interface gráfica, para todas as tarefas;
- facilidades para a criação de classes de atores, comportamentos e edição de cenas;
- importação de vários formatos de geometria tridimensional;
- capacidade de visualizar a animação ou parte dela, sem ser necessário um processo de *rendering* sofisticado;
- possibilidade de gravação da animação quadro a quadro, a fim de facilitar a comunicação com outros sistemas de animação e/ou *rendering;*

Pelos objetivos propostos pode-se perceber que a ênfase no protótipo (bem como na abordagem) é a especificação da animação, isto é, dos comportamentos. Porém, a gravação de animações permite que praticamente qualquer sistema de síntese de imagens possa ser utilizado, o que facilita a geração de animações cujas imagens apresentem mais realismo.

O protótipo foi desenvolvido em ambiente *Windows*, o que ajuda a padronizar e tornar mais intuitiva a sua utilização, uma vez que o ambiente é conhecido e dominado por um grande número de usuários.

Sua operação se dá basicamente em três telas principais: edição de classes de atores, edição de comportamentos e edição de cenas. A seguir serão detalhados os procedimentos possíveis em cada uma.

5.2 Edição de Classes de Atores

A figura 5.1 apresenta a tela para edição de classes de atores. Nesta tela podem ser vistas as várias áreas de trabalho, a saber:

- área para criação/manipulação de classe e geometria (1);
- área para criação/edição/remoção de atributos (2);
- área de visualização tridimensional da geometria (3);
- área de controles para visualização (4).

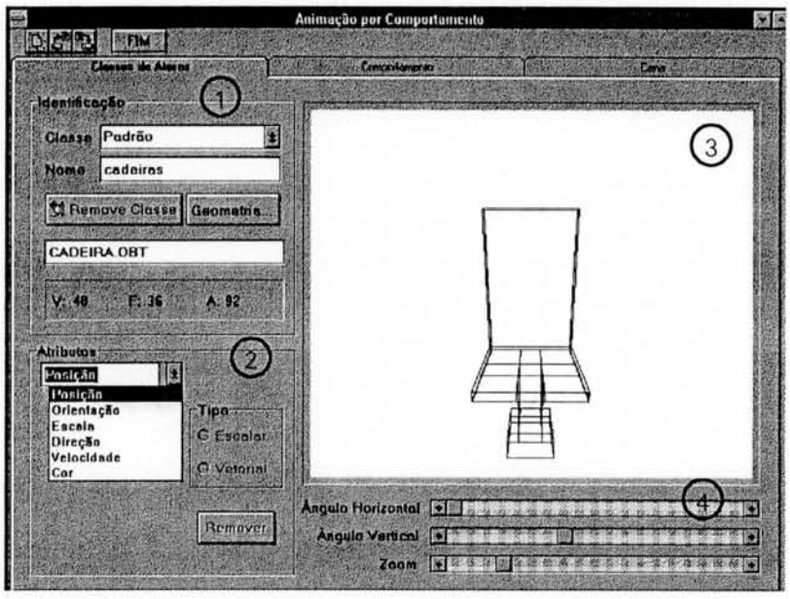


Figura 5.1 - Tela para edição de classes de atores

5.2.1 Criação / Manipulação de Classes e Geometria

A figura 5.2 apresenta a área de edição / manipulação de classes e geometria. Para criar uma nova classe, primeiramente seleciona-se a classe da qual a nova derivará, através da lista *Classe*. A criação de uma nova classe é feita simplesmente digitando-se o nome da classe na área de edição *Nome*. Será pedida uma confirmação e a classe será inicializada com os atributos daquela de onde foi derivada.

É fundamental associar-se à nova classe um arquivo que descreva a geometria desejada, pressionando-se o botão ***Geometria***. Este fará com seja apresentada uma caixa de diálogo para seleção de arquivo, onde serão exibidos os arquivos de geometria encontrados no diretório corrente. Uma vez lido com sucesso, o objeto tridimensional aparece na área de visualização e a informação do arquivo, número de vértices (V), faces (F) e arestas (A) é atualizada nesta área.

A remoção de uma classe de atores é feita pressionando-se o botão ***Remove Classe***. Esta operação tem que ser confirmada pelo usuário.

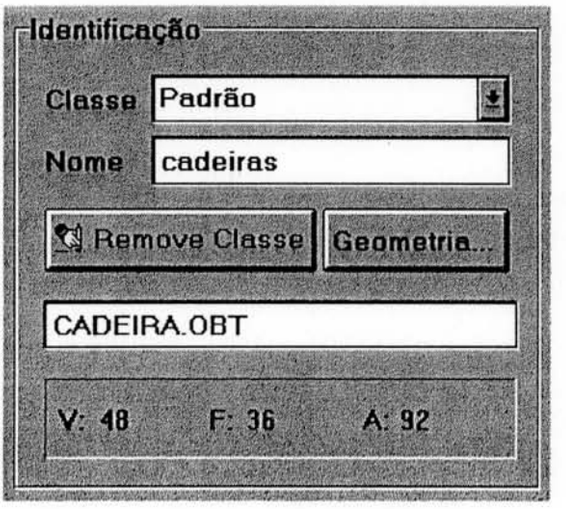


Figura 5.2 - Área para criação / manipulação de classes e geometria

5.2.2 Criação / Manipulação de Atributos

A figura 5.3 apresenta a área de criação / manipulação de atributos. Para se criar um novo atributo, digita-se o seu nome na área de edição *Atributo*. Se for digitado o nome de um atributo já existente, este poderá ser editado. O atributo ainda pode ser selecionado na lista que aparece logo abaixo. A edição de um atributo consiste na definição de seu tipo: simples ou composto, o que é feito pelos botões de seleção *Simples* e *Composto*.

É importante salientar que os atributos que foram herdados **não podem ser removidos nem editados**, pois sua definição não pertence a esta nova classe. Somente os atributos criados aqui podem ser manipulados.

A remoção de um atributo (não herdado) é feita pressionando-se o botão ***Remover***. Será solicitada uma confirmação do usuário.

Figura 5.3 - Área para criação / manipulação de atributos

5.2.3 Controles para Visualização da Geometria

A figura 5.4 apresenta os controles para a visualização da geometria. O objetivo destes é permitir uma visualização tridimensional flexível, porém simples de se obter. O objeto é visto de uma determinada distância de sua origem, distância esta especificada pelo parâmetro *Zoom*. Os controles de ângulo horizontal e vertical especificam coordenadas polares em torno do objeto. Dessa forma, fica fácil observar a geometria sob qualquer ângulo desejado.

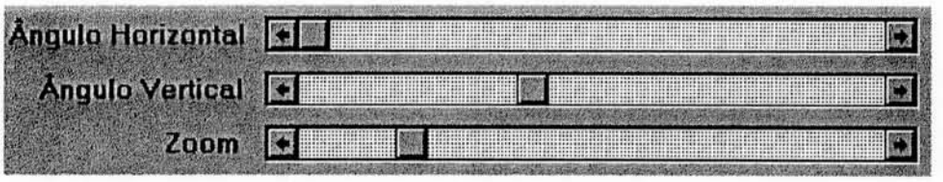


Figura 5.4 - Controles para visualização da geometria de uma classe

5.3 Edição de Comportamentos

A figura 5.5 apresenta a tela para edição de comportamentos. Nesta tela podem ser vistas as várias áreas de trabalho, a saber:

- área para criação/seleção/remoção de comportamento (1);
- área para seleção de comandos (2);
- área para seleção de comandos de controle (3);
- área para seleção de movimentos primitivos (4);
- área para edição da descrição do comportamento (5);
- área para edição / visualização das restrições (6);

- área para edição / visualização dos eventos (7).

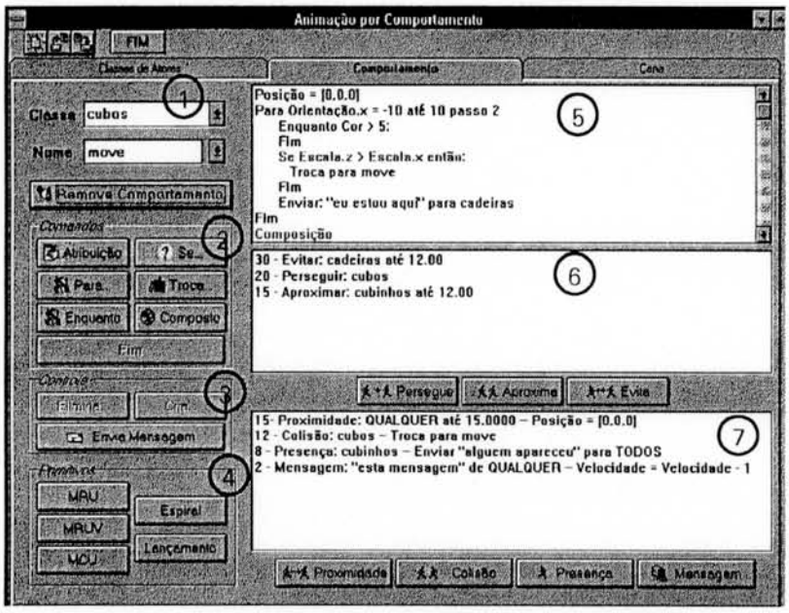


Figura 5.5 - Tela para edição de comportamentos

5.3.1 Criação, Seleção e Remoção de Comportamentos

Esta área (figura 5.6) é utilizada nas funções de criação de um novo comportamento, seleção de um comportamento já existente e eliminação de um comportamento.

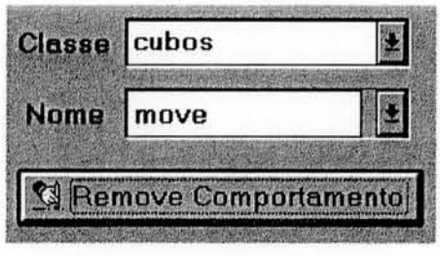


Figura 5.6 - Área para criação e manipulação de comportamentos

Para se criar um novo comportamento, seleciona-se a classe desejada na lista *Classe* e digita-se o nome do comportamento, na área de edição *Nome*. O sistema pede uma confirmação e cria um novo comportamento, vazio.

A seleção de um comportamento existente se dá através da digitação do seu nome, na mesma área de edição, ou através de seleção na lista presente nesta área.

O botão ***Remove Comportamento*** elimina o comportamento da memória, sempre exigindo uma confirmação do usuário.

5.3.2 Inclusão e Edição de Comandos no Comportamento

A inclusão de comandos em um comportamento se dá através da seleção de um botão de comando. Este comando pode estar na área de comandos normais, de controle ou primitivos (figuras 5.7, 5.8 e 5.9).

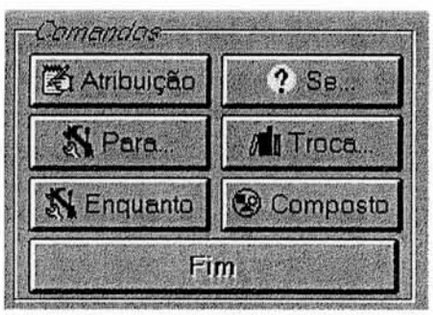


Figura 5.7 - Área para seleção de comandos

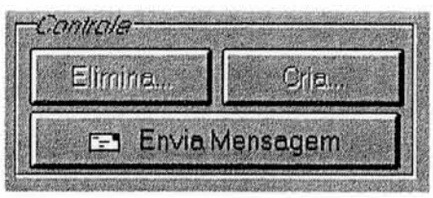


Figura 5.8 - Área para seleção de comandos de controle

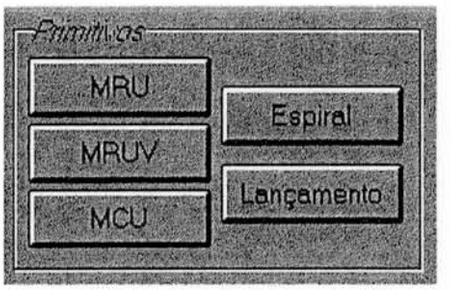


Figura 5.9 - Área para seleção de movimentos primitivos

Cada comando selecionado, independentemente da área onde está localizado, apresenta uma janela com campos a serem preenchidos pelo usuário, com o intuito de gerar a instrução correspondente na área para visualização da descrição de comportamentos. Como exemplo, a figura 5.10 apresenta a janela de edição de um iterador *Para...*, a figura 5.11 apresenta a janela de edição de um comando para envio de mensagens e a figura 5.12 apresenta a janela de edição dos parâmetros do movimento primitivo de lançamento oblíquo.

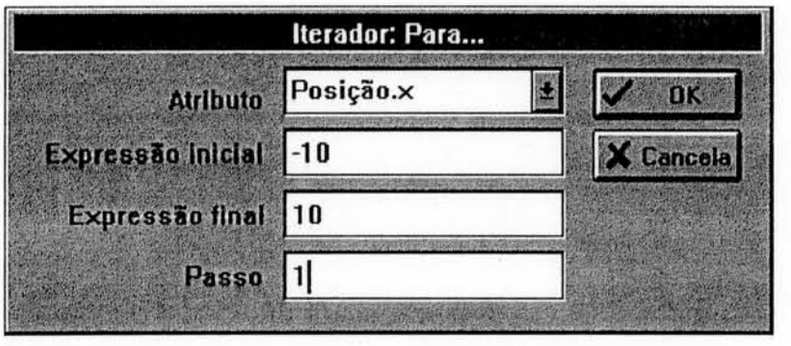


Figura 5.10 - Janela de edição de um iterador *Para...*

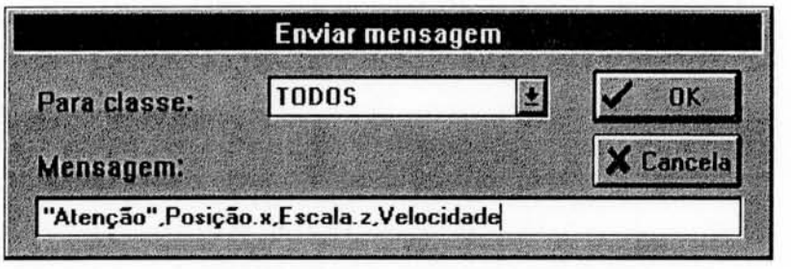


Figura 5.11 - Janela de edição para envio de mensagem

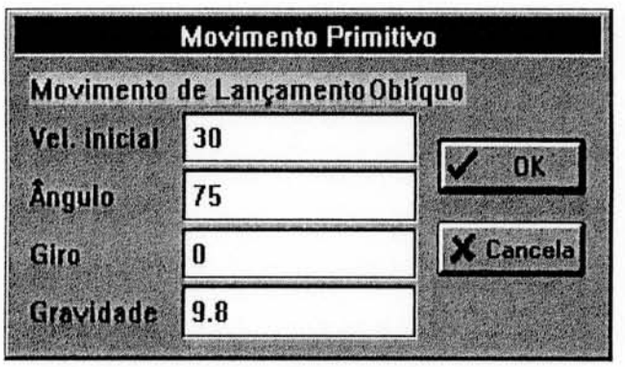


Figura 5.12 - Janela de edição dos parâmetros de um lançamento

Uma vez confirmados os parâmetros de um comando (através do botão ***OK*** na janela de edição), a instrução correspondente será apresentada na área para visualização da descrição (figura 5.13). Os comandos serão indentados, na medida em que estiverem aninhados dentro de um comando de iteração, condicional ou composição. Para finalizar um comando composto (*Para...*, *Enquanto...* ou composição), deve-se pressionar o botão ***Fim***, que marcará o final do bloco de comandos.

```
Posição = [0,0,0]
Para Orientação.x = -10 até 10 passo 2
    Enquanto Cor > 5:
    Fim
    Se Escala.z > Escala.x então:
      Troca para move
    Fim
    Enviar: "eu estou aqui" para cadeiras
Fim
Composição
```

Figura 5.13 - Área para visualização da descrição dos comportamentos

Na área de visualização da descrição, cada tipo de comando é exibido com uma cor diferente (para facilitar a leitura) e cada um pode ser selecionado com o *mouse* (ou com as setas). A tecla *DEL* permite apagar o comando selecionado e a tecla *ENTER* (ou um duplo clique do *mouse*) permite editar os parâmetros do comando. Dessa forma, minimiza-se erros de digitação por parte do usuário, pois ele nunca entra diretamente com as instruções na lista e, para todo parâmetro digitado, é feita uma verificação de consistência de valores.

5.3.2 Inclusão e Edição de Restrições ao Comportamento

Com o objetivo de simplificar a visualização do comportamento como um todo, foi criada uma área especial para a exibição e edição das restrições (figura 5.14). Essa área apresenta cada restrição em uma linha, indicando a prioridade da restrição (o primeiro número), o tipo de restrição e os parâmetros (se houver).

Figura 5.14 - Área para exibição e edição de restrições

As restrições são inseridas através dos três botões localizados abaixo da lista: ***Persegue, Aproxima e Evita***. Cada botão, ao ser pressionado, exibe na tela uma janela para edição dos parâmetros da restrição e, caso seja confirmado, atualiza a lista de restrições. Como exemplo, a figura 5.15 mostra a janela para edição de uma restrição de aproximação:

Figura 5.15 - Janela para edição de uma restrição de aproximação

Assim como na edição dos comportamentos, as restrições podem ser selecionadas com o *mouse* ou com as setas do teclado, removidas com a tecla *DEL* e editadas com um duplo clique ou *ENTER*.

5.3.3 Inclusão e Edição de Eventos para o Comportamento

Os eventos também são exibidos em uma área especial, com o mesmo objetivo da exibição de restrições: facilitar a visualização do comportamento (figura 5.16). Essa área, de forma idêntica à utilizada para as restrições, exibe cada evento em uma linha, indicando a prioridade do evento, o tipo de evento, os parâmetros do evento e o comando de resposta ao evento.

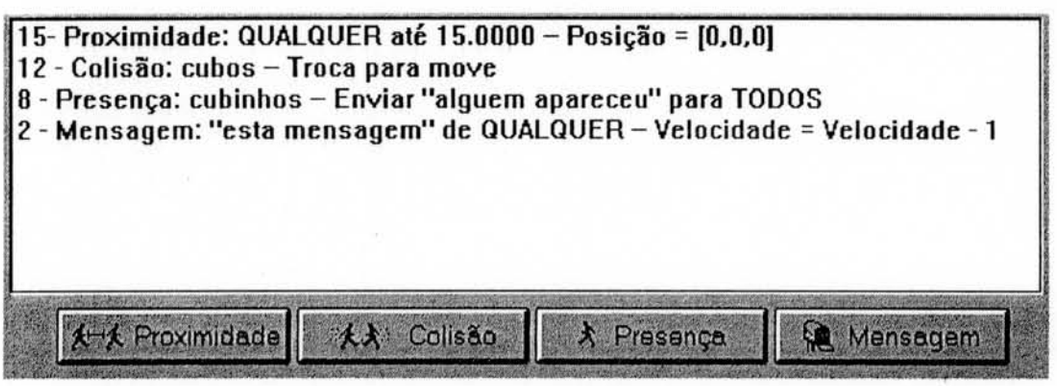


Figura 5.16 -Área para exibição e edição de eventos

Acrescenta-se novos eventos pela pressão de um dos quatro botões localizados abaixo da área de visualização. Cada botão apresenta uma janela para edição (figura 5.17) dos parâmetros relacionados ao evento. Estes parâmetros são: o ator desejado, a prioridade do evento, distância (se houver), mensagem (se houver) e o tipo de resposta quando o evento ocorrer. A resposta em si é definida pelo botão ***Resposta***, o qual exibirá a janela para edição do comando correspondente ao tipo de resposta.

Figura 5.17 - Janela para edição dos parâmetros de um evento de proximidade

A edição e remoção de eventos é realizada de forma idêntica à de restrições e comandos.

5.4 Edição de Cenas

A edição de cenas é feita na tela apresentada na figura 5.18. Esta tela possui diversas áreas, cada uma com uma função diferenciada:

- área para criação, seleção e remoção de atores (1);
- área para edição de valores dos atributos do ator (2);
- área para visualização tridimensional da cena (3);

- área para controle da visualização (4);
- área para controle e gravação da animação (5);
- área de controle da geração e exibição da pré-visualização (6).

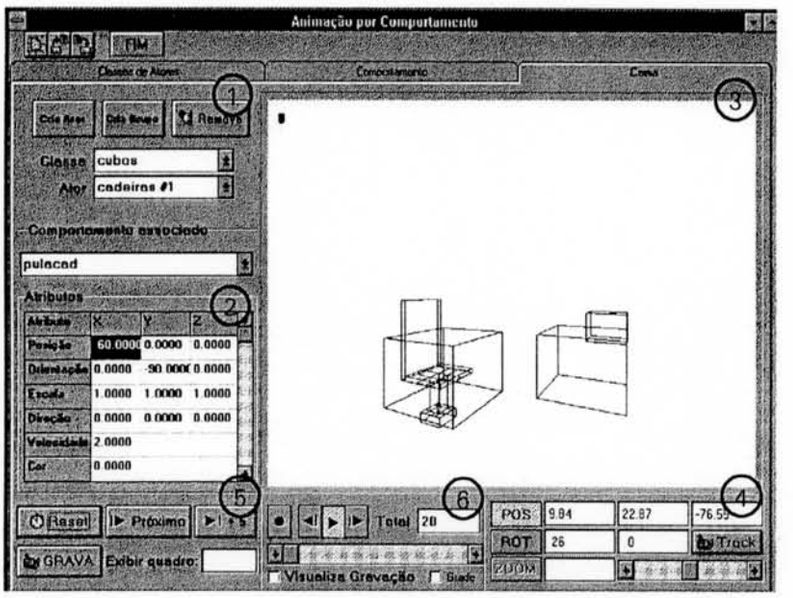


Figura 5.18 - Tela para edição de cenas

5.4.1 Criação, Seleção e Remoção de Atores

A fim de compor uma seqüência de animação, deve-se primeiramente criar os atores necessários e associar os comportamentos desejados a estes. A figura 5.19 apresenta a área para a manipulação de atores.

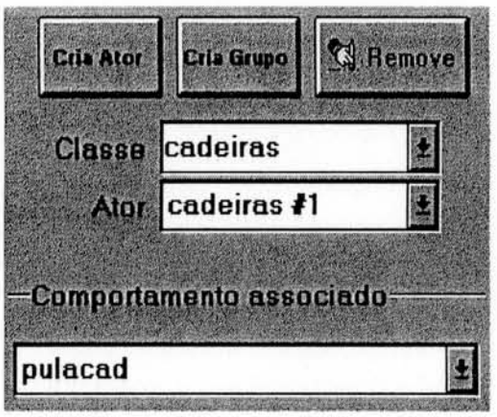


Figura 5.19 - Área para criação, seleção e remoção de atores

Para se criar um ator, seleciona-se a classe desejada na lista *Classe* e pressiona-se o botão ***Cria Ator***. Uma instância desta classe será então criada e exibida na área de visualização. A lista *Ator* agora possuirá o nome deste novo ator. A lista *Comportamento associado* permite agora a seleção do comportamento inicial deste ator, inicialmente informando que este não tem nenhum comportamento associado. Obviamente, só serão listados os comportamentos que foram criados para a classe do ator ou para as classes das quais esta foi derivada.

Seleciona-se o ator desejado através da lista *Ator*. Este será exibido em cor vermelha e com linhas tracejadas na área de visualização. O botão ***Remove*** elimina o ator selecionado, pedindo uma confirmação para o usuário.

Com o propósito de facilitar a criação de grupos de atores de uma mesma classe, pode-se utilizar o botão ***Cria Grupo***, o qual exibe a tela mostrada na figura 5.20.

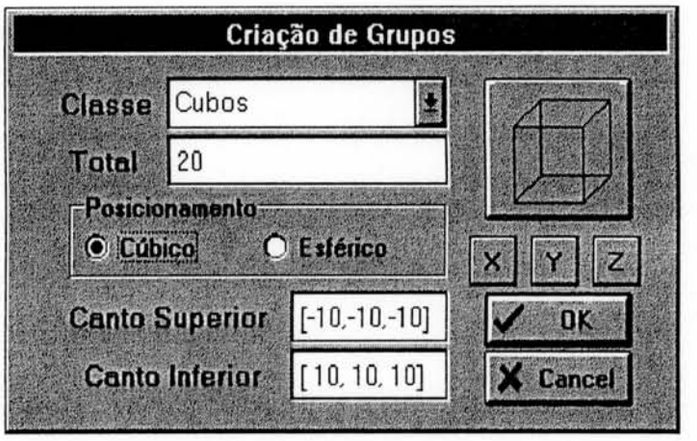


Figura 5.20 - Tela para criação de grupos

Para se criar um grupo, deve-se indicar a **classe de atores** deste grupo, o número de elementos a serem criados e o tipo de posicionamento destes elementos.

- Na lista *Classe*, seleciona-se a classe desejada.
- O campo de edição *Total* indica o número de elementos que deverão ser criados.
- Em caso de posicionamento **cúbico**, selecionado na caixa *Posicionamento*, os atores serão dispostos na superfície ou interior de um cubo, cujas dimensões serão especificadas nos campos *Canto Superior* e *Canto Inferior*, representando dois cantos opostos do cubo.
- Já no posicionamento **esférico**, os atores serão dispostos na superfície ou interior de uma esfera, cujas dimensões serão especificadas através dos campos *Centro* e *Raio* (substituindo os campos *Canto Superior* e *Canto Inferior*).
- O posicionamento dos atores dentro da área especificada é puramente aleatório, porém respeitando o volume ocupado por cada um, de forma a não gerar colisões entre eles.
- No canto superior direito da janela, será apresentada uma imagem representando o tipo de posicionamento: um cubo ou uma esfera.

Abaixo desta há três botões: **X, Y e Z**, os quais controlam quais coordenadas serão geradas. Por exemplo, se somente o botão X estiver selecionado, os atores serão gerados sem deslocamento no eixo Y e Z.

- O botão ***OK*** confirma a criação do conjunto e o botão ***Cancela*** retorna à área principal sem executar a função.

5.4.2 Edição dos Valores de Atributos

Esta área, apresentada na figura 5.21 permite a edição dos valores de todos os atributos do ator selecionado. Este valores serão aqueles utilizados pelo sistema no início da seqüência de animação, porém um vez que são alterados internamente no decorrer desta, esta área sempre mantém os valores informados. Dessa forma, é fácil se fazer vários testes sem precisar ajustar todos os atributos novamente.

A edição é simples: pressiona-se o botão do *mouse* sobre o componente do atributo desejado e digita-se o novo valor, pressionando-se *ENTER* para confirmar a alteração. Devido à maneira como foi implementado, os atributos que não são compostos exibem uma área em branco nos componentes Y e Z. Esta área é ignorada pelo sistema, somente o componente X é considerado como o valor simples.

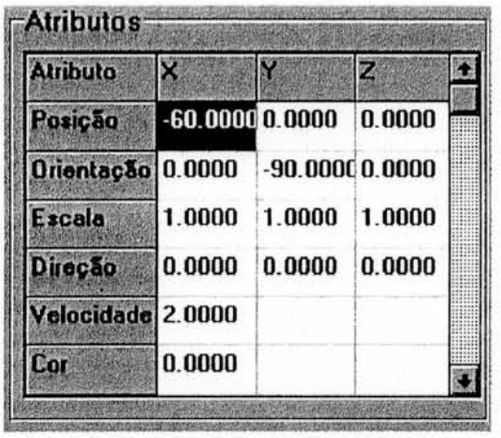


Figura 5.21 - Área para edição dos valores de atributos

5.4.3 Controle dos Parâmetros de Visualização

O controle da câmera sintética é realizado nesta área (figura 5.22). Aqui pode-se controlar a posição, orientação e *zoom* da câmera.

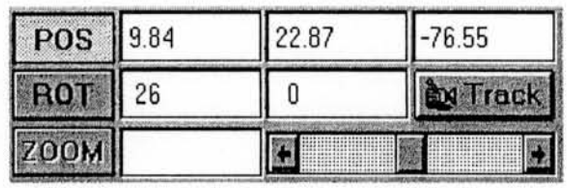


Figura 5.22 - Área de controle da visualização

Os botões ***POS e ROT***, uma vez selecionados, permitem a manipulação das coordenadas do observador e dos ângulos de orientação da câmera diretamente na área de visualização:

- Se o botão ***POS*** estiver ativo, ao se pressionar o botão esquerdo do *mouse* na área de visualização, deslocamentos horizontais do *mouse* alterarão a coordenada X do observador e deslocamentos verticais, a coordenada Z. Se o botão direito estiver pressionado, o deslocamento vertical do *mouse* alterará a coordenada Y do observador.

- Se o botão ***ROT*** estiver ativo, ao se pressionar o botão esquerdo do *mouse* na área de visualização, deslocamentos horizontais do *mouse* alterarão o ângulo horizontal da câmera e se o botão direito estiver pressionado, deslocamentos verticais alterarão o ângulo vertical da câmera.

Este tipo de controle é mais preciso que aquele utilizado na visualização de geometria, na tela de definição de classe de atores. Este último tem o único propósito de o usuário conseguir "enxergar" a geometria de cada ator sem ter que criar uma cena completa.

As caixas de edição à direita do botão ***POS*** permitem a entrada direta das coordenadas X, Y e Z do observador. Aquelas localizadas à direita do botão ***ROT*** servem para a especificação do ângulo horizontal (também chamado *heading*, variando de 0 a 359 graus) e do ângulo vertical (ou *tilt*, mesma variação) da câmera. A que está à direita do botão ***ZOOM*** especifica o ângulo de abertura a ser utilizado (entre 5 e 175 graus, normalmente 90 graus). Este ainda pode ser ajustado através da barra de rolagem presente na mesma linha.

O botão ***Track*** permite ativar a opção de *tracking*, a qual faz com que a câmera mantenha-se sempre orientada na direção do ator selecionado. Dessa forma, o ator sempre será visível, não importando para onde se movimente. Para desativá-la, basta selecionar o mesmo ator e pressionar novamente o botão. Se for selecionado um novo ator e pressionado o botão, este será agora o alvo escolhido.

5.4.4 Controle e gravação da animação

Através desta área (figura 5.23), é possível se exibir qualquer quadro desejado da animação. O controle é bastante simples: o botão ***Reset*** volta a animação para o início, o quadro 0; o botão ***Próximo*** gera e exibe o próximo quadro; o botão denominado ***+5*** avança cinco quadros na animação.

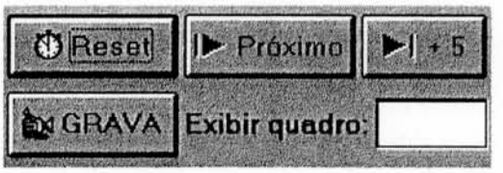


Figura 5.23 - Área para controle e gravação da animação

O usuário pode ainda digitar o número do quadro desejado na caixa de edição *Exibir quadro*. É importante ressaltar que cada mudança de quadro exige novamente a avaliação do comportamento para aquele instante, a fim de se gerar a

visualização correta. Essa avaliação obriga a repetição da seqüência desde o início, até o quadro desejado. O número do quadro atual é sempre exibido no canto superior esquerdo da área de visualização tridimensional.

A janela para gravação dos quadros (figura 5.24) é exibida pressionando-se o botão *GRAVA*.

No campo *Arquivo-base* deve ser digitado o nome que será utilizado como base para formar o nome dos arquivos. A este nome será acrescentado o número de cada quadro, utilizando 4 dígitos (exemplo: quad0000, quad0001, ...). Este nome deve possuir no máximo 4 caracteres, devido à limitação no tamanho de nome dos arquivos no MS-DOS.[1]

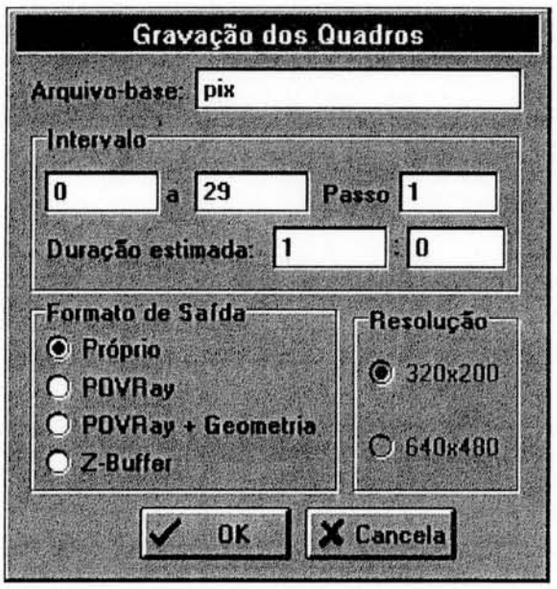


Figura 5.24 - Tela de gravação dos quadros

A área denominada *Intervalo* permite a especificação do intervalo de quadros que devem ser gravados, bem como o passo a ser utilizado entre eles. Este passo é um valor em ponto flutuante. A caixa de edição *Duração estimada* exibe a

[1] Esta limitação inexiste se estiver sendo utilizado o ambiente Windows ™ 95.

duração que a seqüência tem em segundos e centésimos de segundo, respectivamente. É interessante observar que qualquer valor digitado em qualquer um dos campos de edição faz com que o sistema recalcule automaticamente os demais. Por exemplo, se a duração é aumentada de 1 para 2 segundos, o passo será automaticamente corrigido pela metade do seu valor (a fim de gerar o dobro de quadros).

Estes controles foram criados porque o sistema não tem nenhum mecanismo de sincronização, nem de controle por tempo. Desta forma, o usuário pode, pelo menos, determinar a duração exata das animações que cria.

Os quadros podem ser gravados em quatro formatos de saída:

- Formato próprio: armazena os quadros apenas com a informação de posicionamento da câmera, ângulos, *zoom* e posição, orientação, escala e cor de cada ator (vide seção 5.5.3). Nenhuma informação referente à geometria é armazenada, exceto o nome do arquivo utilizado.

- POVRay: armazena cada quadro como um arquivo *.POV, utilizado pelo *software* de *ray tracing* denominado Persistence Of Vision (POV). Este sistema é de domínio público e pode ser facilmente utilizado para gerar imagens com um elevado grau de realismo[2]. A geometria de cada ator será indicada como um objeto POV (vide seção 5.5.3) a ser incluído na leitura.

- POVRay + Geometria: armazena cada quadro como o formato anterior, mas inclui a geometria de cada ator no arquivo *.POV gerado, o que o torna substancialmente mais longo, mas não necessita de nenhum arquivo externo (vide seção 5.5.3).

- Z-Buffer: armazena cada quadro como uma imagem sintetizada através de um subsistema de *Z-Buffer* [PIN 92], utilizando o formato de *bitmaps* do Windows (BMP). Os quadros podem ser gravados em duas resoluções: 320 x 200 pixels e 640 x 480 pixels.

---

[2] O pacote POV-Ray pode ser encontrado via ftp anônimo em caracol.inf.ufrgs.br.

5.4.5 Controle e exibição da pré-visualização

A função de pré-visualização (figura 5.25) foi criada para que o usuário pudesse ter uma idéia do movimento gerado, sem ter que gravar os quadros um a um e gerar as imagens com um sistema de *rendering*. Esta função **grava** cada imagem em uma área de memória e depois as exibe, conseguindo-se grande velocidade.

Figura 5.25 - Área de controle e exibição da pré-visualização

Os controles da função de pré-visualização são os seguintes:

- indicar o número de quadros a serem gravados: digita-se o valor desejado no campo *Total*;
- iniciar o processo de gravação: pressiona-se o botão mais à esquerda desta área. Após alguns instantes, o sistema devolve o controle para o usuário. Se a opção *Visualiza Gravação* estiver selecionada, cada quadro gerado será exibido enquanto é gravado;
- exibir a seqüência gravada: pressiona-se o botão central (semelhante à uma tecla de *play* de um videocassete). Toda a seqüência gravada será exibida, com o máximo de velocidade possível, o que dependerá da rapidez do computador que está sendo utilizado;
- visualizar um quadro específico: pode-se avançar ou retornar um quadro de cada vez, pressionando-se os botões à direita e à esquerda do botão de *play*, ou ainda posicionar a barra de rolagem logo abaixo na posição desejada;

- a opção *Grade* exibe uma grade tridimensional no plano XZ (y=0), de forma a facilitar o posicionamento espacial dos atores e câmera.

5.5 Especificação de Expressões

No protótipo, em todo e qualquer campo que exija a digitação de uma expressão, pode-se pressionar o botão esquerdo do *mouse* duas vezes, rapidamente, e será exibida a janela para edição de expressões (figura 5.26), a qual visa simplificar o processo de entrada de expressões.

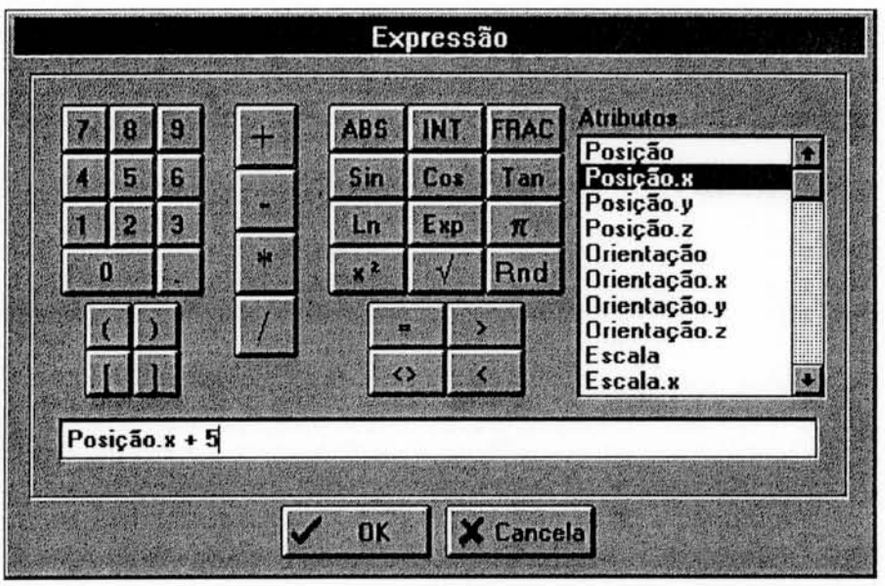


Figura 5.26 - Janela para edição de expressões

Através desta janela, a expressão pode ser entrada sem digitação, utilizando-se os botões presentes, a saber:

- dígitos (**0..9 e "."**) - inserem o dígito ou o ponto decimal na área de edição;

- parênteses ("(","**)**") - permitem o agrupamento de operações que devem ser executadas como um todo;
- colchetes ("**[**","**]**") - permitem a definição de um valor composto;
- operadores matemáticos (+, -, *, /) - inserem a operação correspondente na expressão;
- operações vetoriais ("." **e** "*") - quando os operandos são grandezas vetoriais, estas duas operações realizam produto escalar e vetorial, respectivamente;
- operadores relacionais (=, >, <, <>) - inserem a operação correspondente na expressão;
- lista de atributos - permite a visualização e escolha de um atributo para ser inserido na expressão;
- funções - todas as funções (exceto π) exigem um argumento entre parênteses e realizam a operação especificada. O argumento pode ser qualquer expressão cujo resultado seja simples. A função **Rnd** retorna um número aleatório entre 0 e o valor do argumento.

Ao se pressionar o botão ***OK***, a sintaxe da expressão será verificada e esta será recusada se houver algum erro. Dessa forma, o usuário só consegue informar ao sistema expressões sintaticamente corretas.

5.6 Exemplo de movimento modelado por comportamento

Com o intuito de demonstrar o funcionamento do método, nesta seção é apresentado um exemplo prático de um movimento gerado. Primeiramente, define-se as classes de atores que serão utilizadas:

```
Classe Outdoors
Derivada de Classe Padrão
```

```
Eventos:
Prior. 10 - Mensagem "cuidado!" de qualquer -
                    Troca para Desvia
```

- Comportamento *Desvia* - definido para a classe *Cadeiras,* executa um movimento de lançamento, porém com uma direção de 45 graus. Tem o objetivo de desviar o ator de outro que está se aproximando. Será ativado pelo comportamento *Espera.*

```
Comportamento Desvia
Classe: Cadeiras

  Enquanto 1:
     Lançamento (Giro=45, Vel=30, Ang=80, G=5)
  Fim

Eventos:
Nenhum
```

A cena é definida com dois atores: um da classe *Outdoors* e um da classe *Cadeiras.* O comportamento *Salta* é associado ao ator *Outdoors #1* e o comportamento *Espera* ao ator *Cadeiras #1.* A figura 5.27 apresenta os atores, já com suas posições iniciais ajustadas.

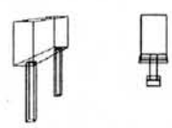


Figura 5.27 - Quadro inicial da animação de exemplo

Até o quadro 9, o ator *Outdoors #1* descreverá o movimento de lançamento (figura 5.28). A partir daí, o evento de proximidade ocorrerá e o ator *Cadeiras #1* iniciará o seu movimento de lançamento, para evitar a colisão. A figura 5.29 apresenta o estado da animação no quadro 15.

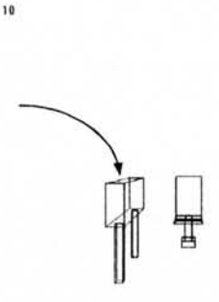


Figura 5.28 - Quadro #10 da animação de exemplo

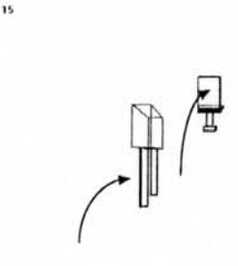


Figura 5.29: Quadro #15 da animação de exemplo

6. ANÁLISE DO PROTÓTIPO

Com o intuito de validar a abordagem proposta, será feita uma análise do protótipo desenvolvido, seguindo os mesmos critérios utilizados na comparação das abordagens e sempre tentando relacionar o tipo de resultado obtido nesta abordagem com os que seriam resultado das demais.

6.1 Generalidade

Este critério, como visto anteriormente, determina se o método é aplicável a vários tipos de situações. É um critério difícil de ser avaliado, uma vez que pode-se imaginar um grande número de situações diferentes. Porém, será feita uma comparação através de exemplos selecionados, julgados mais relevantes entre os principais tipos de animação modelada tridimensional.

• **Animação Paramétrica (*key-frame*)**: apesar de não trabalhar com a especificação de quadros (ou tempos), necessários à uma animação por quadros-chave, o método é capaz de simular este tipo de animação, pois pode-se especificar o valor dos atributos básicos (*Posição, Orientação, Escala, ...*) em qualquer parte de um comportamento. A única dificuldade é justamente a sincronização.

• **Animação de objetos cuja forma muda durante o processo:** a única maneira de alterar a aparência de um ator **durante a animação** é modificando-se os seus atributos básicos. Ou seja, não é possível alterar a geometria do ator, pelo menos não diretamente. Se isso for absolutamente necessário, pode-se, por exemplo, eliminar o ator e criar um novo, de uma classe com uma geometria diferente, na mesma posição. Porém o método não suporta geometrias com hierarquia, tais como objetos articulados.

• **Animação calculada através de dinâmica:** o processo de obtenção das novas posições de cada ator se dá, como visto, através de *cinemática direta*, pois a partir de vetores de velocidade são obtidos os *deslocamentos* para cada coordenada. Logo,

animações necessitando de análise dinâmica do movimento (direta ou inversa) não podem ser criadas com este método.

Apesar das limitações expostas nos exemplos acima, considera-se que o método ainda assim apresenta uma boa generalidade, o que pode ser constatado através do exemplo mostrado na seção 5.6.

6.2 Resultados

O critério de resultados é bastante subjetivo, pois algumas pessoas podem considerar o movimento não-satisfatório, ao passo que para outras é perfeito. Esta análise dificilmente pode ser feita com um rigor técnico muito elevado. Assim, de forma semelhante à utilizada na análise de generalidade, serão expostos os pontos principais onde foram detectados problemas na abordagem:

- Detecção de colisões por *bounding box*: apesar de simples, este tipo de detecção pode considerar colisões onde realmente não houve contato, especialmente se o ator tiver uma forma um pouco irregular. Isto é agravado ainda mais se o ator estiver rotacionado, pois a *bounding box* não é rotacionada junto.

- Avaliação das restrições: a forma como as restrições são avaliadas pode gerar situações indesejáveis, como: uma restrição anular a outra, ou o movimento final não ser aquele que o usuário desejaria obter. Porém, cabe lembrar que a imprevisibilidade deve ser um fator presente em animação definida por comportamentos.

- Eficiência do método: a abordagem implementada é, de forma geral, bastante eficiente. A avaliação dos comportamentos, apesar de ser um processo de interpretação, é rápida. Os processos mais lentos são, com certeza, as avaliações de restrições e eventos. Por exemplo, um evento de colisão testa **todos** os atores da classe especificada. Este é um algoritmo cuja complexidade é $O(n^2)$, isto é, cresce quadraticamente com o número de elementos envolvidos.

6.3 Interatividade

Com relação ao critério de interatividade, foi criada uma *interface* gráfica, buscando facilitar o uso e simplificar processos de digitação. Por exemplo, a descrição dos comportamentos, apesar de ser através de comandos, evita ao máximo a digitação, a não ser para a entrada de parâmetros (atributos, expressões, etc). Com isso, o usuário dificilmente comete erros de digitação. A separação de comandos, restrições e eventos também facilita a visualização do comportamento como um todo.

Contudo, apesar desses pontos positivos, a *interface* tornou-se um pouco complexa, devido ao grande número de funções oferecidas, o que fugiu um pouco da idéia inicial, que era criar um ambiente facilmente operável. Ainda assim, considera-se que o sistema utiliza razoavelmente bem os recursos de interação com o usuário. Deve-se ressaltar que este tipo de abordagem, sem uma *interface* gráfica, seria extremamente difícil de se utilizar, porque exigiria o conhecimento detalhado da sintaxe da linguagem criada.

6.4 Simplicidade

Conforme mencionado na seção anterior, a *interface* tornou-se um pouco mais sofisticada que aquela inicialmente imaginada. De qualquer forma, está claramente presente a simplicidade em se obter determinados tipos de movimento, bastando lembrar dos comportamentos primitivos, por exemplo. Porém, isto não quer dizer que a abordagem é necessariamente simples de se utilizar em qualquer caso. Dependendo do tipo de movimento que se deseja obter, sua definição pode-se tornar bastante complexa, exigindo várias descrições de comportamento, restrições, eventos, manipulação de atributos, troca de mensagens, etc.

6.3 Interatividade

Com relação ao critério de interatividade, foi criada uma *interface* gráfica, buscando facilitar o uso e simplificar processos de digitação. Por exemplo, a descrição dos comportamentos, apesar de ser através de comandos, evita ao máximo a digitação, a não ser para a entrada de parâmetros (atributos, expressões, etc). Com isso, o usuário dificilmente comete erros de digitação. A separação de comandos, restrições e eventos também facilita a visualização do comportamento como um todo.

Contudo, apesar desses pontos positivos, a *interface* tornou-se um pouco complexa, devido ao grande número de funções oferecidas, o que fugiu um pouco da idéia inicial, que era criar um ambiente facilmente operável. Ainda assim, considera-se que o sistema utiliza razoavelmente bem os recursos de interação com o usuário. Deve-se ressaltar que este tipo de abordagem, sem uma *interface* gráfica, seria extremamente difícil de se utilizar, porque exigiria o conhecimento detalhado da sintaxe da linguagem criada.

6.4 Simplicidade

Conforme mencionado na seção anterior, a *interface* tornou-se um pouco mais sofisticada que aquela inicialmente imaginada. De qualquer forma, está claramente presente a simplicidade em se obter determinados tipos de movimento, bastando lembrar dos comportamentos primitivos, por exemplo. Porém, isto não quer dizer que a abordagem é necessariamente simples de se utilizar em qualquer caso. Dependendo do tipo de movimento que se deseja obter, sua definição pode-se tornar bastante complexa, exigindo várias descrições de comportamento, restrições, eventos, manipulação de atributos, troca de mensagens, etc.

7. CONCLUSÕES

O presente trabalho apresentou uma nova abordagem para a obtenção de animação modelada por comportamento, isto é, animação na qual o usuário não sabe exatamente qual será o movimento final gerado, mas apenas tem uma idéia. A abordagem implementada no protótipo apresentou algumas falhas, mencionadas no capítulo 6, porém este protótipo caracteriza-se como um sistema experimental. Nesse contexto, pode ser facilmente estendido, com novas funções, novas formas de avaliação de comportamento, etc.

Considera-se que os objetivos iniciais foram atingidos: o estudo das várias abordagens já existentes e a experimentação de uma nova abordagem, realizada no protótipo. Como sugestões para melhorias do protótipo, cita-se:

- substituição do algoritmo de detecção de colisões, por algum outro que seja mais preciso, ou seja, realizando a detecção através de um método diferente da comparação de *bounding boxes* (lembrando que isto certamente aumentará o tempo de processamento) [MOO 88];
- reformulação da avaliação de restrições, a fim de evitar situações conflitantes ou, em último caso, fornecer uma solução alternativa;
- reformulação do tratamento de eventos, buscando utilizar um formalismo mais rigoroso [KAL 92];
- utilização de um algoritmo cuja complexidade seja inferior àquela do algoritmo utilizado na verificação de proximidade e colisão. Na verdade, isto só serviria para aumentar a velocidade da avaliação de cada comportamento, mas não é absolutamente essencial;
- possibilidade da especificação de movimentos para a câmera sintética, pois atualmente só a função de *tracking* é capaz de trabalhar com a câmera (orientando esta) durante a animação. Uma maneira seria criar uma classe primitiva *Câmera* e criar atores do tipo *câmera* derivados desta. Estes poderiam ser ativados a qualquer instante, ter comportamentos associados a estes, etc.

ANEXO A-1 FORMATO DOS ARQUIVOS

Este anexo apresenta o formato de cada tipo de arquivo utilizado pelo sistema. O sistema utiliza cinco tipos de arquivos:

- descrição de classes de atores: *.ATR;
- descrição de comportamentos: *.BHV;
- descrição de cenas: *.CEN;
- descrição geométrica de objetos: *.OBT;
- saída de quadros: *.FRA, *.POV, *.BMP.

A-1.1 Descrição de classes de atores

```
Ator <Nome da Classe>
Classe <Nome da classe de onde foi derivada>
Geometria <Nome do arquivo>
Atributos:
<Nome do Atributo> <Tipo: S ou C> <Herdado 0/1>
...
Fim Classe
... Próximo ator ...
```

O tipo de cada atributo é indicado com um caractere "**S**" - se o atributo for simples - ou "**C**" - se for composto. Logo a seguir deve aparecer um caractere "**1**", indicando que o atributo foi herdado de outra classe ou um caractere "**0**", indicando que é um atributo desta classe.

A-1.2 Descrição de comportamentos

```
Comportamento <Nome do Comport.>
Classe <Classe de atores a que se aplica>
Descrição:
.. Comandos ...
Restrições:
.. Restrições ...
Eventos
... Eventos ...
Fim Comportamento
```

Os comandos são especificados da forma a seguir, onde **<IND>** indica a indentação do comando, de acordo com o nível de aninhamento deste. Este nível é inicializado em zero e incrementado quando se está aninhando comandos.

```
<Ind> ATRIB <Atributo>;<Valor>
<Ind> PARA <Atributo>;<Inicial>;<Final>;<Passo>
<Ind> ENQUANTO <Expressão>
<Ind> FIM
<Ind> SE <Expressão>
<Ind> TROCA <Nome do Comport.>
<Ind> ENVIA <Mensagem>;<Classe>
<Ind> CRIA <Classe>;<Dir.>;<Dist>;<Comp. inic.>
<Ind> ELIMINA
<Ind> COMPOSICAO
<Ind> MRU <Direção>;<Velocidade>
<Ind> MRUV <Direção>;<Velocidade>;<Aceleração>
<Ind> MCU <Eixo>,<Raio>,<Veloc. angular>
<Ind> ESPIRAL <Eixo>;<Inic>;<Fin>;<Vel>;<Passo>
<Ind> LANÇAMENTO <Giro>;<Vel>;<Ang>;<G>
```

A especificação das restrições é feita através do seguinte formato, onde <**PRI**> indica a prioridade da restrição.

```
EVITA <Pri>;<Dist>;<Classe>
BUSCA <Pri>;<Classe>
APROX <Pri>;<Dist>;<Classe>
```

A definição de eventos é realizada de forma semelhante às restrições, porém acrescenta-se um comando de resposta a cada evento:

```
PROXIM <Pri>;<Dist>;<Classe>
<Comando de Resposta>

COLIS <Pri>;<Classe>
<Comando de Resposta>

PRES <Pri>;<Classe>
<Comando de Resposta>

MENS <Pri>;<Mensagem>;<Classe>
<Comando de Resposta>
```

A-1.3 Descrição das cenas

```
Cena
Observador <X> <Y> <Z>
Orientação <Ang. horiz> <Ang. vert>
Zoom <Ang. abertura>
```

```
Classe: <Classe do ator>
Indice: <Indice do ator>
Comportamento: <Comport. associado>
<Atributo> <Valor> ou
<Atributo> <X> <Y> <Z>
...
Fim Ator
... Proximo ator ...
Fim Cena
```

A-1.4 Descrição geométrica de objetos

```
<No. de vértices>
<X0> <Y0> <Z0>
<X1> <Y1> <Z1>
...
<No. de faces>
<No. vert. face 0> <V0> <V1> ...<COR>
...
<No. de cores>
<No. da cor> <R> <G> <B>
...
```

Cada face possui um certo número de vértices, cujos índices são especificados após este número (**V0, V1, ...**). A informação de faces é necessária para a função de saída dos quadros, onde pode-se solicitar a gravação da geometria (*POVRay + Geometria*). A cada face é também atribuída uma COR (1-256), que representa um índice na tabela de cores do objeto (se esta existir) ou o número "0", indicando que não há tabela de cores. Se houver, esta segue logo após a última face.

A-1.5 Saída de quadros

Existem quatro formas de saída de quadros: *formato próprio, POVRay, POVRay + Geometria* e *Z-Buffer*. A primeira forma grava apenas os parâmetros da câmera e os atributos do ator para cada quadro. Dessa maneira, um outro sistema de *rendering*, por exemplo, pode utilizar somente a informação que necessita.

```
Quadro <No. quadro>
Observador <X> <Y> <Z>
Orientação <Ang. horiz> <Ang. vert>
Zoom <Ang. abertura>

Ator <Nome do arq. que descreve a geometria>
<Atributo> <Valor> ou
<Atributo> <X> <Y> <Z>
...
Fim quadro
```

A segunda e terceira formas são quase iguais. Neste caso, é gravado um arquivo de descrição de cenas para o sistema de *ray tracing* denominado *POVRay*. Este sistema, sendo de domínio público, pode ser facilmente obtido e utilizado para gerar animações completas e com elevada qualidade visual. A diferença entre a segunda e terceira forma é que na segunda é gravada uma referência aos arquivos que descrevem as geometrias e na terceira, esta é gravada no próprio arquivo de descrição, o que torna este maior. A quarta forma grava imagens no formato BMP (vide seção 4.4.4).

• *Formato de gravação* ***POVRay****:*

```
#include "colors.inc"
#include "shapes.inc"
#include "textures.inc"

// Define uma fonte de luz
object {
   light_source { <0 0 0>          // Posição
  color White
```

```
  }
}

camera {
  Position <X> <Y> <Z>          // Observador
  direction <X> <Y> <Z>         // Direção do alvo
  Up <0 1 0>                    // "Up" vector
}

// Cada linha inclui um ator
#include "<Arq. geometria ator #1>.pov"
#include "<Arq. geometria ator #2>.pov"
```

- Formato de gravação **POVRay + Geometria** - é igual até a definição de câmera, então cada ator é definido como:

```
object {
   union {
      triangle {
          <X Y Z>          // Face 0
          <X Y Z>
          <X Y Z>
        }
      triangle {
          <X Y Z>          // Face 1
          <X Y Z>
          <X Y Z>
        }
     ...
     scale <Escala.x Escala.y Escala.z>
     rotate <Orient.x Orient.y Orient.z>
     translate <Posição.x Posição.y Posição.z>
     }
   texture { color <Cor.x Cor.y Cor.z> }
}
```

BIBLIOGRAFIA

[ALV 79] ÁLVARES, Beatriz Alvarenga; DA LUZ, Antônio Máximo Ribeiro. **Curso de Física.** São Paulo: Harper & Row do Brasil, 1979. v. 1.

[BRA 84] BRAITENBERG, V. **Vehicles:** Experiments in Synthetic Psychology. Cambridge: MIT Press, 1984.

[COD 88] CODERRE, B. Modeling Behavior in Petworld. In: **Artificial Life.** Reading: Addison-Wesley, 1988.

[FEI 93] FEIJÓ, Bruno; COSTA, Mônica M.F. Animação Comportamental Baseada em Lógica. In: SIMPÓSIO BRASILEIRO DE COMPUTAÇÃO GRÁFICA E PROCESSAMENTO DE IMAGENS, **6.**, 1993, Recife. **Anais...** Recife: SBC/UFPE, 1993.

[FOL 90] FOLEY, J. D. et al. **Computer Graphics:** Principles and Practice. 2nd. ed. Reading: Addison-Wesley, 1990. 1174p.

[GRE 91] GREEN, Mark. Using Dynamics in Computer Animation: Control and Solution Issues. In: **Making them Move:** Mechanics, Control, and Animation of Articulated Figures. Berkeley, California: Addison-Wesley, 1991. 348p.

[HAU 88] HAUMANN, David; PARENT, Richard E. The behavioral test-bed: obtaining complex behavior from simple rules. **The Visual Computer**, Toronto, p. 14-33, April 1988.

[HEL 90] HELLER, Dan. **XView Programming Manual**. Sebastopol: O'Reilly & Associates, 1990. 642p.

[HOL 75] HOLLAND, John. **Adaptation in Natural and Artificial Systems**. Ann Arbor, MI: University of Michigan Press, 1975.

[KAL 92] KALRA, D; BARR, A. H. Modeling with Time and Events in Computer Animation. In: EUROGRAPHICS, **13.**, 1992, Cambridge. **Proceedings...** Cambridge: Academic Press, 1992.

[KOZ 92] KOZA, John R. **Genetic Programming:** On the programming of computer by means of Natural Selection. Cambridge, MA: MIT Press, 1992.

[LAT 89] LATOMBE, Jean-Claude. Introduction to Robot Motion Planning. In: IEEE VIDEOCONFERENCE ON ROBOTICS, 1989, Cambridge. **Proceedings...** Cambridge: MIT Press, 1989.

[MAG 85] MAGNENAT-THALMANN, Nadia; THALMANN, Dave. **Computer Animation:** Theory and Practice. Berlin: Springer-Verlag, 1985. 248p.

[MOO 88] MOORE, M; WILHEMS, J. Collision Detection and Response for Computer Animation. In: COMPUTER GRAPHICS, 1988, Toronto. **Proceedings...** Toronto: ACM Press, 1988.

[OLA 92] OLABARRIAGA, Sílvia Delgado et al. Integrating Graphic Software - The making of MATE. In: CONFERENCIA LATINO AMERICANA DE INFORMÁTICA, **18.**, 1992, Las Palmas de Gran Canaria. **Actas...** Las Palmas de Gran Canaria: CLEI, 1992. p. 353-361.

[OVE 92] OVERMARS, Mark H. **Forms Library:** A Graphical User Interface Toolkit for Silicon Graphics Workstations. Netherlands: Utrecht University, Dept. of Computer Science, 1992. 120p.

[PIN 92] PINHO, Márcio Serolli; COHEN, Marcelo. Implementação de *Z-Buffer*. [S.l.:s.n.], 1992. Trabalho de pesquisa.

[PUG 84] PUGH, John R. Actors - The Stage is Set. **SIGPLAN Notices**, New York, v. 19, n. 3, Mar. 1984.

[RAM 90] RAMALHO JÚNIOR, Francisco. **Os Fundamentos da Física.** 5.ed. São Paulo: Moderna, 1990. v. 1.

[REY 87] REYNOLDS, Craig W. Flocks, Herds, and Schools: A Distributed Behavioral Model, In: SIGGRAPH, 1987, Chicago. **Proceedings...** Chicago: ACM Press, 1987.

[REY 92] REYNOLDS, Craig W. An Evolved, Vision-Based Behavioral Model of Coordinated Group Motion. In: **From Animals to Animats.** Cambridge, MA: MIT Press, 1992.

[REY 93] REYNOLDS, Craig W. An Evolved, Vision-Based Model of Obstacle Avoidance Behavior. In: ARTIFICIAL LIFE, **3.**, 1993, New York. **Proceedings...** New York: Addison-Wesley, 1993.

[REY 93a] REYNOLDS, Craig W. Evolution of Obstacle Avoidance Behavior: Using Noise to Promote Robust Solutions. In: **Advances in Genetic Programming**. Cambridge: MIT Press, 1993.

[ROG 90] ROGERS, David; ADAMS, J. Alan. **Mathematical Elements for Computer Graphics**. New York: McGraw-Hill, 1990.

[SCH 92] SCHMIDT, Ana Elisa; MUSSE, Soraia Raupp. **PREVIEW:** Sistema de Animação. Porto Alegre: CPGCC da UFRGS, 1992. 47p.(RP-184)

[SIL 92] SILVA, Rodrigo L. **Animaker - Sistema de Animação**. Porto Alegre: Instituto de Informática da UFRGS, 1992. 96p.

[SKI 38] SKINNER, B. F. **The behavior of organism**. New York: Appleton-Century, 1938.

[SUN 93] SUN, Hanqiu; GREEN, Mark. The Use of Relations for Motion Control in an Environment With Multiple Moving Objects. In: GRAPHICS INTERFACE, 1993, Toronto. **Proceedings...** Toronto: ACM Press, 1993.

[TRA 88] TRAVERS, M. Animal Construction Kits. **Artificial Life**. New York: Addison-Wesley, 1988.

[WAT 92] WATT, Alan; WATT, Mark. **Advanced Animation and Rendering Techniques:** Theory and Practice. New York: ACM Press, 1992. 455p.

[WIL 87] WILHELMS, Jane. Toward Automatic Motion Control. **IEEE Computer Graphics and Applications**, Los Alamitos, CA, p. 12-18, April 1987.

[WIL 90] WILHELMS, J ; SKINNER, R. A Notion for Interactive Behavioral Animation Control. **IEEE Computer Graphics and Applications**, Los Alamitos, CA, p. 14-22, May 1990.

[YAE 93] YAEGER, Larry. **Petworld and other subjects in artifical life**. Porto Alegre, UFRGS. 14 out. 1993. (Comunicação via correio eletrônico: larryy@apple.com)

Informática  *UFRGS*

Estudo e Experimentação de Animação Baseada em Comportamento

por

Marcelo Cohen

Dissertação apresentada aos Senhores:

José T. F. Camargo

Dr. José Tarcísio Franco de Camargo (UNICAMP)

Prof. Dr. Ricardo Augusto da Luz Reis

José Valdeni de Lima

Prof. Dr. José Valdeni de Lima

Anatólio Laschuk

Prof. Anatólio Laschuk

Vista e permitida a impressão.
Porto Alegre, 12 / 04 / 96 .

Flávio Rech Wagner

Prof. Dr. Flávio Rech Wagner,
Orientador.

Profa. Dra. Carla Maria Dal Sasso Freitas,
Co-Orientador.

Flávio Rech Wagner

Prof. Flávio Rech Wagner
Coordenador do Curso de Pós-Graduação
em Ciência da Computação - CPGCC
Instituto de Informática - UFRGS