SISTEMA DE CONTROLE DE ANIMAÇÕES DE PERSONAGENS PARA O FRAMEWORK GUFF

Márcio da Silva Camilo

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

Orientadora: Aura Conci

Niterói, 2006

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

C183 Camilo, Márcio da Silva.
Sistema de controle de animações de personagens para o framework guff / Márcio da Silva Camilo – Niterói,. RJ : [s.n.], 2006.
143 f.

Orientador: Aura Conci.
Dissertação (Mestrado em Ciência da Computação) - Universidade Federal Fluminense, 2006.

1. Framework (Programa de computador). 2. Computação gráfica. 3. Imagem 3D. 4. Jogos em computador. 5. Animação por computador. 6. Engenharia de software. I. Título.

CDD 005.3

SISTEMA DE CONTROLE DE ANIMAÇÕES DE PERSONAGENS PARA O FRAMEWORK GUFF

Márcio da Silva Camilo

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

Banca Examinadora

_____

Profª. Aura Conci – IC/UFF

_____

Prof. Otton Teixeira da Silveira Filho – IC/UFF

_____

Prof. Esteban Walter Gonzalez Clua – PUC-Rio

Niterói, 2006.

Agradecimentos

Agradeço em primeiro lugar a Deus Todo-Poderoso, a Nosso Senhor Jesus Cristo e ao Divino Espírito Santo. A Nossa Senhora e todos os Santos e Santas. Ao meu querido Anjo da Guarda e a todos os Guias Espirituais que velam por mim.

Agradeço à minha família, pois tudo o que sou devo a eles.

Agradeço a minha orientadora Aura Conci e aos demais membros da banca pelas valiosas observações.

Finalmente agradeço aos amigos que me deram apoio, em especial: Viviane Fernandes, Wilson Tadeu e Carlos Magno.

Resumo

Frameworks para jogos 3D possuem um módulo de sistema de controle de animações de personagens, baseado em um baixo nível de abstração, responsável pelas animações, e em um alto nível de abstração relacionado com a inteligência e o comportamento dos personagens.

O baixo nível do sistema de controle de animações é responsável por prover animações e reproduzi-las de forma correta e eficiente, de acordo com os comportamentos resultantes do alto nível.

Esse trabalho tem como objetivo apresentar a implementação do baixo nível para o *framework* Guff. Esta implementação foi baseada em técnicas utilizadas em jogos 3D atualmente comercializados, mas apresenta também uma nova abordagem utilizando um estágio intermediário de configuração que permite ao projetista de jogo controlar as animações manipulando as características de uniformidade (baseada na distância entre *keyframes*), evolução e *timing*. Esse estágio permite também a configuração de transições entre seqüências de animação e a associação entre seqüências de animações a comportamentos que o personagem apresentará durante o jogo.

Foi desenvolvida, também, uma abordagem preliminar para a camada do alto nível, baseado em uma máquina de estados finitos, visando mostrar como a camada do baixo nível é eficaz em responder as necessidades da camada do alto nível.

Abstract

3D Games frameworks have a module of character animation system based on a low level layer for dealing with animations and a high level layer related with the intelligence and behavior of characters.

The low level animation system is responsible for providing a way of obtaining animations and to play them efficiently and correctly, according to the behaviors resulting of the high level animation system.

This paper presents the implementation of the low level animation layer of the Guff framework's Character Animation System. This implementation was done based on techniques used in nowadays commercialized 3D games, presenting also a new approach, using an intermediate configuration stage which allows the game designer to control animations handling its characteristics of uniformity (based on the distance between keyframes), evolution and timing. This stage also allows the setting of transitions between animation sequences and the behavior and animation sequences association.

Also, a first approach to a high level layer, based in a finite state machine, was developed to show how the low level layer can support the high level layer's demands.

Sumário

Sumário iv
Lista de Figuras vi
Lista de Tabelas x
1. Introdução 11
2. Simulação d animações em 3D para jogos em 2D 13
2.1 Jogos *side scrolling* 13
2.2 Jogos com visualização isométrica 17
2.3 Jogos *first person shooter* 20
2.3 Conclusão do Capítulo 21
3. Animação de personagens para jogos em 3D 22
3.1 Evolução do processo de animação *keyframing* para jogos em 3D 30
3.2 Animações baseadas em interpolação de posições de vértices 33
3.2.1 O formato Model Quake 2 35
3.2.2 O formato Model Quake 3 39
3.3 Animação baseada em hierarquia 42
3.3.1 Interpolações de translações e rotações 46
3.3.1 O formato MilkShape 3D 47
3.4 Técnicas de *Skinning* 50
3.4.1 O formato Model Doom 3 53
3.3 Conclusão do Capítulo 57
4. O sistema de controle de animações de personagens do *framework* Guff 58
4.1 O *framework* Guff 58
4.2 O sistema de controle de animações de personagens 60
4.3 Comparações com outros *frameworks* e *game engines* 64
4.4 Conclusão do Capítulo 66
5. Configuração da seqüência de animação pelo número de *intermediate frames* 68
5.1 Controle baseado na uniformidade entre *keyframes* 72
5.2 Controle baseado em evolução 75
5.3 Controle de *timing* adaptativo baseado na taxa de *frames* por segundo da aplicação 79
5.4 Conclusão do Capítulo 82
6. Módulo de transição de seqüências de animação 84
6.1 Tipos de transições 85
6.2 Posicionamento 86
6.3 Mesclagem 88
6.4 Conclusão do Capítulo 94
7. Alto nível do sistema de controle de animações de personagens 96
7.1 Alto nível do sistema de controle de animações de personagens do *framework* Guff 97
7.2 Conclusão do Capítulo 100
8. Resultados Experimentais 101
8.1 Testes e Resultados 101
8.2 Conclusão do Capítulo 107
9. Conclusão 108
9.1 Dificuldades 109

9.2 Trabalhos Futuros 109
Referências Bibliográficas 111
Apêndice A 119
A.1 Estágio de Design 119
A.2 Estágio de Configuração 121
A.3 Estágio de Processamento 126
Apêndice B 129
B.1 Curvas Paramétricas 129
B.1.1 Curvas Bézier 131
B.1.2 Curvas de Hermite 132
B.1.3 *Spline* de Catmull-Rom 133
B.1.4 B-Splines uniformes cúbicas 134
B.2 Parametrização de rotações 135
B.2.1 Ângulos de Euler 135
B.2.2 Axis-Angle 139
B.2.3 Quatérnios 140

Lista de Figuras

Figura 2.1: Imagens que representam as animações de um personagem em jogo 2D 13
Figura 2.2: “Sonic II”, com *parallax scrolling* criando a ilusão de profundidade 14
Figura 2.3: *Golden Axe*, movimentação em três direções permitindo a simulação de 3 dimensões 15
Figura 2.4: *Shining in the darkness*, representação em primeira pessoa 16
Figura 2.5: Phantasy Star 3, jogo com visualização do alto 17
Figura 2.6: *Tiles* retangulares e *tiles* angulares (losangos) 17
Figura 2.7: Ultima8, perspectiva isométrica dando a ilusão de 3 dimensões 18
Figura 2.8: Posições do personagem em um jogo isométrico 18
Figura 2.9: Wolfenstein 3D, um jogo em 2.5 D 20
Figura 3.1: Tie Fighter. Objetos em 3 dimensões com sombreamento constante 21
Figura 3.2: Malha de polígonos e vértices representando uma chaleira 23
Figura 3.3: Malha de polígonos e vértices de um modelo de um personagem, aramado em (a) e texturizado em (b) 23
Figura 3.4: Sistema de controle de animações de personagens em duas camadas 26
Figure 3.5: Abordagem de *keyframing* em dois estágios 29
Figura 3.6: Interpolação de posição de vértices 33
Figura 3.7: Interpolação linear e a *Spline* de Catmull-Rom 33
Figura 3.8: Modelo md2 34
Figura 3.9: Textura de modelo md2 37
Figura 3.10: Um modelo md2 animado 37
Figura 3.11: Hierarquia do modelo md3 39
Figura 3.12: Modelo md3 animado 41
Figura 3.13: Esqueleto (estrutura hierárquica de juntas) 42
Figura 3.14: *Forward Kinematics* e *Inverse Kinematics* 44
Figura 3.15: Modelo baseado no formato MilkShape 3D 48

Figura 3.16: Fissura entre partes do corpo na região da junta 49
Figura 3.17: Polígonos servindo como ligação elástica entre as partes do corpo que se encontram em uma junta 50
Figura 3.18: Distorção inconveniente da “liga” na região da junta 50
Figura 3.19: Vértices influenciados por mais de uma junta por pesos 51
Figura 3.20: Resolução do problema de distorções indesejáveis 51
Figura 3.21: O modelo md5 dividido em um arquivo de malhas e vários arquivos de seqüências de animação 53
Figura 3.22: Estrutura do arquivo de malhas 53
Figura 3.23: Estrutura do arquivo de seqüência de animação 54
Figura 3.24: Modelo criado no formato md5 55
Figura 4.1: Estados do framework Guff 58
Figura 4.2: Ciclo principal de aplicação baseada no framework Guff 59
Figura 4.3: NameSpaces do toolkit do framework Guff 59
Figura 4.4: O estágio intermediário de configuração no baixo nível do sistema de personagens baseado em *keyframing* 61
Figura 4.5: O estágio intermediário de configuração e suas funções 62
Figura 5.1: Comparação entre três abordagens de tratamento de *intermediate frames*, em (a) eles são gravados como *keyframes*, em (b) são representados por um valor constante (α) para cada par de *keyframes*, e em (c) são representados por valores variáveis (α, β, γ) para cada par de *keyframes* 68
Figura 5.2: Distância entre dois *keyframes* 73
Figura 5.3: Contribuição de todos os vértices para o cálculo da distância 73
Figura 5.4: *Slow in* e *Slow out* 75
Figura 5.5: Curva senoidal 76
Figura 5.6: Curva parabólica 77
Figura 5.7: Curva B-*spline* uniforme cúbica 77
Figura 5.8: A taxa de *frames* por segundo interfere no fator de *timing* fazendo variar a velocidade da seqüência de animação 81
Figura 6.1: Conceitos de seqüência atual, seqüência seguinte e fase de mesclagem 84
Figura 6.2: A questão do posicionamento distante em *frames* alinhados 86

Figura 6.3: Remapeamento do sistema de coordenadas 86
Figura 6.4: Mesclagem com interpolação linear 88
Figura 6.5: Efeito “vai suave e de repente salta” 90
Figura 6.6: Efeito “vai sauve então reposiciona e então volta” 91
Figura 6.7: Efeito “bate e volta” 93
Figura 7.1: Alto nível de um sistema de controle de animações de personagens 96
Figura 7.2: Máquina de estados finitos do estágio reativo 98
Figura 8.1: Personagem Archvile do jogo DOOM3 101
Figura 8.2: Distâncias de segmentos da seqüência de animação attack2 do personagem Archvile do jogo DOOM3 101
Figura 8.3: Combinação dos controles de equalização e evolução 103
Figura 8.4: Variação de *frames* por segundo (f/s) e Fator de *timing* (*timing*) com 7 cópias do personagem 104
Figura 8.5: Variação de *frames* por segundo (f/s) e Fator de *timing* (*timing*) com 10 cópias do personagem 104
Figura 8.6: Resultados obtidos de transições da animação “walk” para as animações “evade_left”, “pain_head1”, “sight2” e “attack1” ((a), (b), (c) e (d) respectivamente) 105
Figura A1: Modelo sendo importado para o 3D Studio 119
Figura A2: Modelo sendo exportado para o formato md5 119
Figura A.3: Tela inicial do Guff Animation Configuration 120
Figura A.4: Janela “Character Configuration” 121
Figura A.5: Tela “Behavior Configuration” 122
Figura A.6: A janela “State Configuration” 123
Figura A.7: A janela “Animation Controls Configuration” 124
Figura A.8: A janela “Transition Configuration” 125
Figura A.9: Classe “Character” subordinada ao *namespace* Character 125
Figura A.10: Relação entre os arquivos de *script* do sistema de controle de animações de personagens do *framework* Guff 126
Figura B.1. Bézier de terceira ordem 131
Figura B.2. Curva de Hermite 132
Figura B.3. *Spline* de Catmull-Rom 133

Figura B.4. Ângulos de Euler 135
Figura B.5. *Gimbal lock* 137
Figura B.6. Distorção causada pela interpolação linear de uma rotação 137
Figura B.7. Deslocamento angular do vetor r de valor θ ao retor no eixo n 138

Lista de Tabelas

Tabela 3.1: Seqüências de animação utilizadas pelo modelo md2 35
Tabela 4.1: Comparação entre os sistemas de controle de animações de personagens de bibliotecas, *game engines* e o *framework* Guff 65
Tabela 5.1: Significados de movimento informados através do *timing* da seqüência de animação 79

1. Introdução

A representação de animações em 3 dimensões nos computadores não é uma tarefa trivial, uma vez que tenta recriar movimentos e fenômenos com o máximo de realismo tendo que lidar com as limitações do *hardware* disponível.

Nos jogos de computador em 3D essa dificuldade torna-se mais crítica uma vez que as animações são atualizadas em tempo real concorrendo com várias outras tarefas durante a execução do jogo, a uma taxa de *frames* por segundo que deve ser convincente para o usuário final.

É muito comum atualmente que um conjunto de ferramentas e funções seja utilizado para criação de jogos 3D. *Frameworks* e *game engines* permitem que tarefas sejam agrupadas de forma a permitir eficiência no estágio de desenvolvimento evitando implementações redundantes. Eles também podem possibilitar uma melhor utilização dos recursos durante o estágio de execução do jogo, permitido um desempenho mais eficiente do jogo.

De uma maneira geral, *Frameworks* e *game engines* de criação de jogos de computador incluem algum sistema de criação e gerenciamento de personagens. Esse sistema pode ser subdivido em duas camadas de abstração: um alto nível de abstração e um baixo nível de abstração. O alto nível é responsável por determinar comportamentos de respostas a eventos do ambiente onde o personagem está inserido e também a sua própria inteligência. O baixo nível engloba as tarefas de obtenção (criação e armazenamento) de animações, o processamento das animações requeridas pelo alto nível e a reprodução dessas animações.

Este trabalho tem como primeiro objetivo mostrar como jogos de computador representam animações de personagens em 3 dimensões. Além disso, apresenta-se os princípios e técnicas em que se baseou o projeto de implementação do sistema controle de animações de personagens do *framework* Guff [1]. Tal projeto teve como objetivo principal equipar o

framework Guff com a capacidade de obter, gerenciar e reproduzir animações de personagens.

Este projeto pode ser dividido em 5 partes, que foram implementadas pelo autor e servem de base para esta dissertação.

- Implementação de leitura e reprodução de Animações baseadas nos formatos de personagem e animações da *game engine* do jogo Doom3;
- Desenvolvimento do sistema de interpolação e controles de uniformidade, evolução e *timing*;
- Desenvolvimento do módulo de transições entre as seqüências de animação;
- Implementação da máquina de estados para o comportamento dos personagens (como uma abordagem preliminar para a camada de alto nível do sistema de controle de animações de personagens);
- Desenvolvimento do *software* configurador de interpolações, transições e da máquina de estados para o comportamento dos personagens;

Nos dois capítulos que se seguem, há uma pesquisa de como jogos lidam com a representação em 3 dimensões. Primeiramente são mostradas as abordagens que permitiram que jogos 2D simulassem animações em 3 dimensões (capítulo 2). Depois são mostradas técnicas mais atuais que permitem a utilização de modelos em 3 dimensões em jogos baseados em *keyframing* (capítulo 3).

O sistema de controle de animações de personagens do *framework* Guff é apresentado no capítulo 4. E nos capítulos 5, 6 e 7 são mostradas as funcionalidades que este sistema oferece para configuração das camadas de baixo nível e alto nível do sistema de controle de animações de personagens. O capítulo 8 mostra testes realizados e resultados obtidos pelo sistema. A conclusão deste trabalho é apresentada no capítulo 9. No Apêndice A encontra-se uma descrição dos estágios de *design*, configuração e desenvolvimento para personagens no *framework* Guff.

2. Simulação de animações em 3D para jogos em 2D

Um personagem é qualquer entidade com movimentos e vontade própria com a qual o usuário interage. O próprio usuário está representado por um personagem (seja ele visível ou não) no jogo [2].

Animar significa dar vida a um corpo [3]. Em um filme, jogo ou desenho animado, uma seqüência de animação é uma forma de representação do movimento de um personagem. Seqüências de animação utilizam o princípio de *moving pictures* (movimento de imagens) [4], isto é, imagens estáticas são mostradas rapidamente de forma que o cérebro humano não seja capaz de perceber o intervalo entre elas, criando a ilusão do movimento.

Neste capítulo é apresentado um histórico de como os jogos 2D evoluíram passo a passo para jogos 3D e também como o sistema de controle de animações de personagens acompanhou esta evolução. Jogos para consoles dedicados bem como jogos para arquiteturas APPLE, MSX, PC, Macintosh, entre outros, são indiferenciadamente chamados de jogos de computador. Contudo, dadas as dificuldades de obter informações sobre jogos em consoles e em certas arquiteturas, este estudo baseou-se principalmente em jogos para PC.

2.1 Jogos side scrolling

Jogos *side scrolling* são os jogos em que um personagem principal, comandado pelo usuário, percorre um cenário em uma direção principal (embora o movimento em outras direções seja possível), interagindo com outros personagens e com o ambiente na busca de um objetivo. Como o personagem apresenta estados e se movimenta durante o jogo, é necessário criar alguma forma de gerenciar o estado do personagem, seu posicionamento no ambiente e uma forma de representação que dê ao usuário a idéia de

movimento. Alguns exemplos famosos são: “Pitfall”, “Alex Kid”, “Mario Bros”, “Sonic”.

Geralmente, os personagens são representados por estruturas de dados que contém informações sobre posição do personagem no ambiente, estado do personagem (agachado, em pulo, ferido, invulnerável, etc), representações de imagens 2D do personagem para cada estado (chamadas *sprites* [4] [4]), entre outras informações de controle. A animação do personagem é feita alternando-se as *sprites* do personagem de forma contínua representando sua movimentação. A figura 2.1 mostra uma seqüência de *sprites* armazenadas para a representação de animações de um personagem de um jogo 2D.

Figura 2.1. Imagens que representam as animações de um personagem em um jogo 2D [4]

As *sprites* podem ser consideradas um sistema rudimentar de controle de animações de personagens para jogos 2D. A lógica do jogo e as informações contidas na estrutura de controle de um personagem determinam o comportamento deste personagem. Uma imagem é escolhida dentro do conjunto predeterminado de *sprites* armazenadas para representar o estado atual do personagem em tempo de execução do jogo [4].

As primeiras tentativas de criação da “sensação” de 3 dimensões em jogos de computador, começaram com os jogos em 2D. Desde a década de 1980, alguns jogos de computador já agregavam características que tentavam passar ao usuário a idéia de 3 dimensões. Merecem destaque as técnicas de *parallax scrolling* [5] [6], movimentação em mais de uma dimensão e representação de visão em primeira pessoa.

O *parallax scrolling* é uma técnica que permite que partes do cenário sejam animadas em velocidades diferentes possibilitando assim a impressão

de que o personagem está em primeiro plano e que há planos (camadas) subseqüentes atrás dele. Essa técnica é utilizada em jogos s*ide scrolling* [7], como, por exemplo, o jogo “Sonic II”, mostrado na figura 2.2. Observe-se na figura que 4 planos são representados: dos personagens, do solo, do mar e do céu. Esta técnica foi utilizada pela primeira vez em animações dos estúdios Disney (“O velho Moinho” de 1937 e “Branca de Neve e os sete anões” de 1939) com o nome de “câmera multiplano” [8].

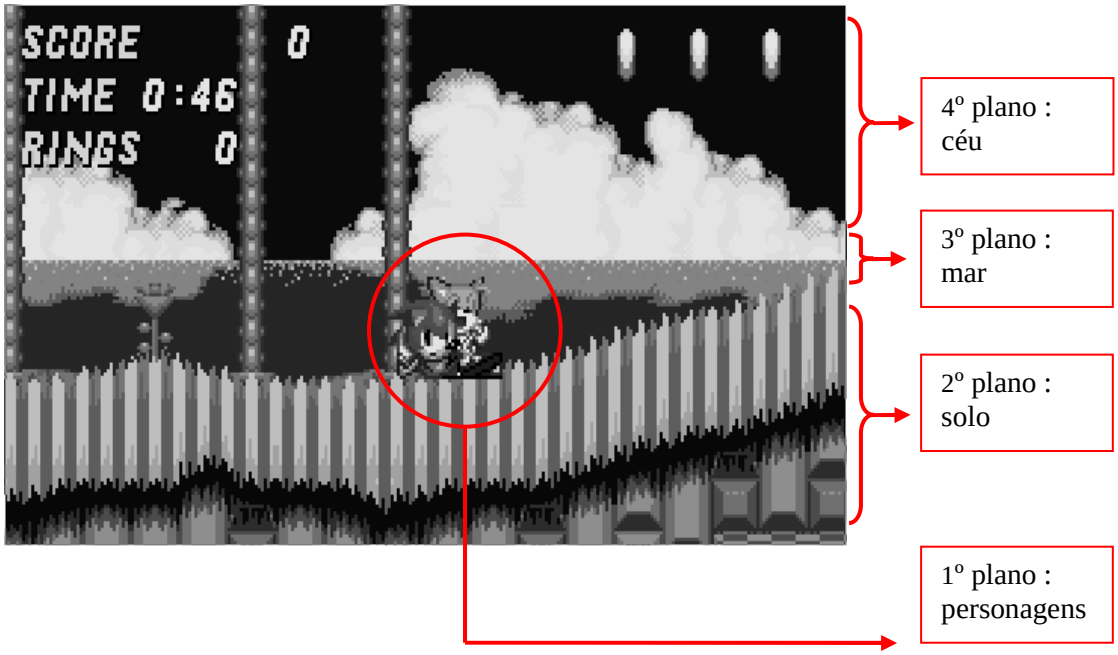

Figura 2.2. “Sonic II”, com *parallax scrolling* criando a ilusão de profundidade

Ainda em jogos do tipo *Side Scrolling*, uma técnica muito utilizada é a movimentação do personagem em mais de uma dimensão. Estendendo a possibilidade de movimentação do personagem, não só se podem ver os planos mais distantes como também “caminhar” até eles. Alguns exemplos de jogos que utilizam essa técnica são “Golden Axé”, “Double Dragon”, “Altered Beast”, “Final Fight”, entre outros.

O personagem pode se mover para frente e para trás (na direção horizontal do vídeo que é naturalmente utilizada em jogos *side scrolling* representando o eixo x), pode se mover para cima e para baixo (na direção vertical do vídeo caracterizando profundidade, ou seja, o eixo z), e finalmente pode saltar (caracterizando o movimento no eixo y). Na figura 2.3 pode ser

visto o jogo “Golden Axe”, que apresenta a movimentação de um personagem em mais de uma dimensão.

É interessante observar que, de maneira geral, os jogos que utilizam essa técnica não apresentam animações diferenciadas para quando o personagem caminha em profundidade (eixo z) como o personagem de frente para a tela quando “desce” o eixo z ou de costas no sentido contrário, porque os resultados nesse caso parecem distorcidos, uma vez que não utilizam a técnica de *bitmap scalling* mostrada mais à frente. Mesmo assim, alguns jogos como “Indiana Jones e o Templo da Perdição”, por exemplo, utilizam essa abordagem.

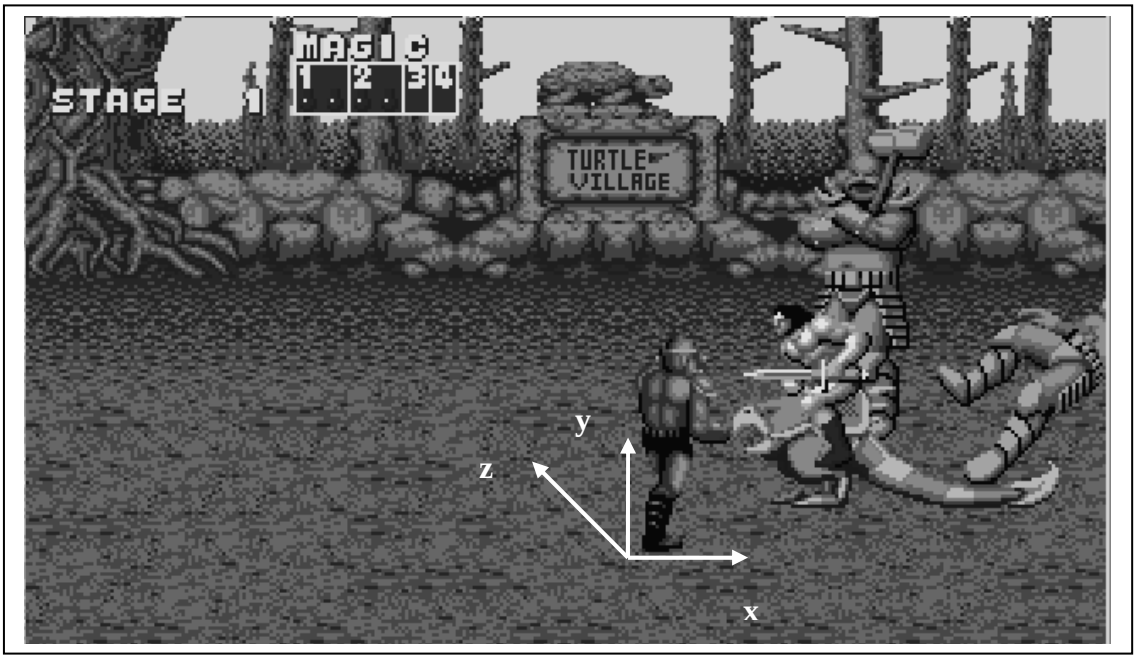

Figura 2.3. *Golden Axe*, movimentação em três direções permitindo a simulação de 3 dimensões

Alguns jogos permitem a visualização do jogador não em terceira pessoa (onde o seu personagem é focalizado), mas em primeira pessoa (onde a visão do jogador é baseada no ponto de vista do personagem), também conhecidos como *first person adventures*. Neste caso, utilizando um desenho em perspectiva com ponto de fuga é possível dar ao jogador a impressão de estar em um ambiente de 3 dimensões. Todas as possíveis visualizações do

ambiente são pré-geradas e armazenadas e na medida em que o personagem percorre o cenário, essas imagens são mostradas. Como exemplos podem ser citados “Shining in the Darkness” (mostrado na figura 2.4) e “Eye of the Beholder”, ambos RPGs.

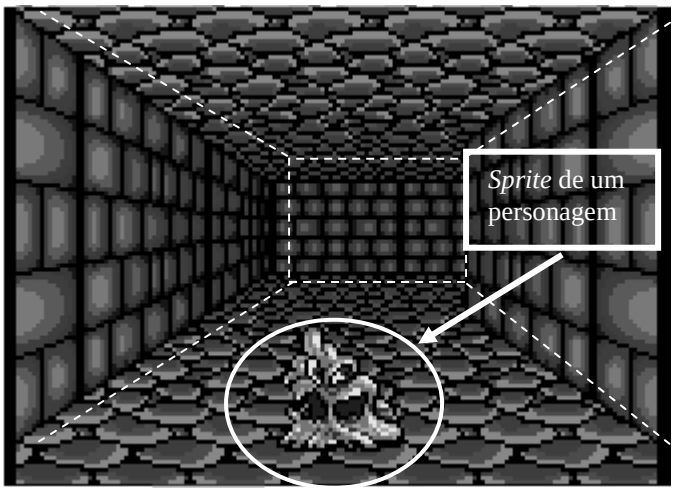

Figura 2.4. *Shining in the darkness*, representação em primeira pessoa

Via de regra, nos jogos *first person adventures*, o personagem principal nunca é visto durante o jogo (às vezes ele pode ser visto em seqüências de animação não manipuladas pelo usuário, conhecidas como *cut-scenes*). Os demais personagens, sobretudo inimigos, são mostrados com poucas ou apenas uma *sprite*, em geral, sobrepostas às imagens representativas do cenário. O uso de mais de uma *sprite* é interessante para realizar pequenas animações, como por exemplo, de um personagem inimigo atacando.

2.2 Jogos com visualização isométrica

Outra categoria de jogos que permite uma visualização diferenciada é o tipo de jogo conhecido como “isométrico”. Nesta categoria de jogo, há uma representação em terceira pessoa, mas ao contrário dos jogos *side scrolling*, há uma simulação de uma câmera com visualização da cena em perspectiva isométrica criando a ilusão de 3 dimensões. Exemplos são os jogos Ultima8 e Diablo. Eles guardam muitas semelhanças com um outro tipo de jogo *adventure* chamado “jogo com visualização do alto”, no qual os personagens

são vistos de cima e podem se movimentar em duas dimensões, como pode ser visto na figura 2.5, mostrando uma seqüência do jogo “Phantasy Star 3”.

A perspectiva isométrica simula a representação em 3 dimensões fazendo os objetos inclinados de 30° graus do plano formado pelos eixos x e z. Essa perspectiva é conhecida como **perspectiva perfeita**, pois as formas não apresentam distorções. Ao contrário dos jogos com visualização do alto, em que os mapas são feitos com peças de cenário (*tiles*) retangulares, nos jogos isométricos os *tiles* são losangos [9], como pode ser visto na figura 2.6.

Figura 2.5. Phantasy Star 3, jogo com visualização do alto

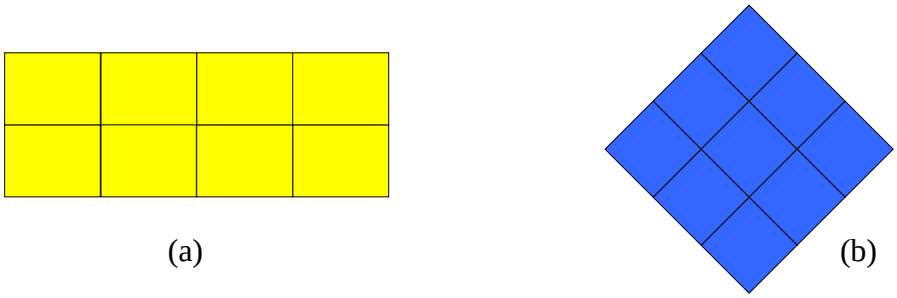

Figura 2.6. *Tiles* retangulares e *tiles* angulares (losangos)

Em termos de movimentação, os jogos isométricos se assemelham muito aos jogos com visualização do alto. Uma diferença importante entre eles é que os jogos isométricos não só dão a impressão de possuírem uma terceira dimensão (eixo y) como de fato, às vezes, o personagem pode pular e também

subir em objetos caracterizando a movimentação no eixo y. Uma cena do jogo Ultima8 é mostrada na figura 2.7.

Uma questão importante acerca dos jogos isométricos é que diferentemente dos outros tipos de jogos, não só o ambiente ao redor do personagem sofre algum tipo de tratamento para simular a existência de 3 dimensões, também os personagens possuem uma representação mais propícia para uma simulação 3D. Embora ainda utilizando o conceito de *sprites*, o personagem passa a contar com conjunto mais completo de imagens que o representam em diferentes posições para cada uma das direções em que o movimento é possível, como pode ser visto na figura 2.8.

Figura 2.7. Ultima8, perspectiva isométrica dando a ilusão de 3 dimensões

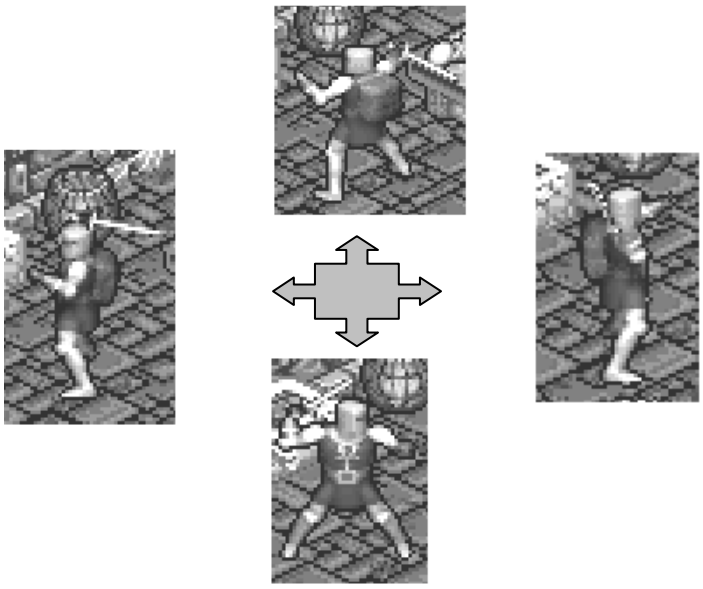

Figura 2.8. Posições do personagem em um jogo isométrico

2.3 Jogos first person shooter

O passo seguinte na representação de 3 dimensões foi dado pelos jogos *Wolfstein 3D* e *Doom* da IdSoftware [10]. Esses jogos baseiam-se em importantes características que permitem uma simulação bastante realista, para os jogos da época, de representação em 3 dimensões [6]. Sendo jogos *first person shooter*, onde o usuário vê através do ponto de vista do personagem, os cenários são criados através de *ray casting* [6] permitindo a visualização de muros texturizados.

Além disso, as técnicas de *bitmap scalling* e representação de *sprites* como nos jogos isométricos, permitem um grau de simulação em 3 dimensões superior ao que foi visto em outros jogos. A técnica de *bitmap scalling* está baseada na idéia de que quando um objeto está próximo do observador ele parece maior e quando está distante ele parece menor (permitindo a percepção de profundidade). A representação de *sprites* em várias direções permitiu que o objeto pudesse ser visto de pontos diferentes apresentando faces diferentes, uma técnica que pode ser considerada uma variação da técnica de *impostors* [11]. Como o número de *sprites* é limitado em geral em posições bem definidas como: frontal, lateral esquerdo, lateral direito e traseira, um giro ao redor de um personagem mostra mudanças brusca entre essas *sprites* direcionais. Dessa forma, o personagem ainda não possui uma representação completa em 3 dimensões.

Jogos deste tipo popularizaram a expressão muito comum na época, de jogos em 2.5 dimensões. O que significa: jogos que simulam muito bem as 3 dimensões mas ainda não as realizam de fato. Uma idéia corrente na época é que, se em um jogo um personagem ou o ambiente não pode ser visto em todas as posições possíveis (e mais analiticamente, se eles não contém uma completa representação em 3 dimensões), então não podem ser considerados jogos em 3D. Na figura 2.9 uma imagem do jogo *Wolfenstein 3D* pode ser vista.

Figura 2.9. Wolfenstein 3D, um jogo em 2.5 D

Embora apresentem a mesma forma de visualização que os jogos *first person adventures*, os *first person shooters* são considerados uma categoria diferente pelas novas técnicas de simulação de 3D que apresentaram. E também por um conceito de jogabilidade mais dinâmico do que o de seus antecessores, com o uso do mouse para mirar e atirar e com possibilidades mais ricas de representação de ambientes e de personagens.

2.3 Conclusão do Capítulo

Alguns gêneros de jogos de computadores 2D já apresentavam técnicas de representação de cenários e personagens que permitiam ao usuário uma sensação da representação de 3 dimensões.

Tais técnicas permitiram uma gradual mudança na forma em que os jogos representam um mundo virtual e seus personagens partindo do simplesmente 2D, passando por formas cada vez apuradas de representação de 3 dimensões, como por exemplo os chamados jogos 2.5 D.

3. Animação de personagens para jogos em 3D

Neste capítulo será mostrado como os jogos em 3D, com o enfoque especial nos personagens, são criados.

Cumpre observar que doravante serão utilizadas as seguintes denominações para profissionais da área de criação de jogos: modelador (*model designer*), o profissional responsável pelo projeto e criação de modelos e seqüências de animação de personagens; projetista de jogo (*game designer*), o profissional responsável por projetar a idéia do jogo e seu funcionamento; desenvolvedor de jogo (*game programmer*), o profissional responsável pela implementação dos módulos de jogo.

Uma geração de jogos seguinte aos jogos 2.5D, adotou uma representação mais adequada a 3 dimensões para cenários e personagens. Primeiramente de formas não texturizadas (representados em *wireframes* ou com sombreamento constante) e mais adiante com texturas representando materiais dos objetos. Um exemplo pode ser visto na figura 3.1, onde o jogo Tie Fighter exibe espaçonaves completamente representadas em 3 dimensões e com sombreamento constante.

Figure 3.1. Tie Fighter. Objetos em 3 dimensões com sombreamento constante

Esses jogos foram considerados os primeiros jogos de fato em 3D. A representação através de arranjos de malhas com polígonos e vértices foi a abordagem preferida entre os jogos substituindo as estruturas de controle de *sprites* popularizada com os jogos 2D. No jogo Quake, por exemplo, os personagens foram representados com malhas de polígonos e vértices e as *sprites* foram utilizadas apenas para representação de explosões [12].

Jogos 3D não se restringem, atualmente, a um único gênero. Existem diversos gêneros atualmente que apresentam representações 3D [13] como: *side scrolling adventure*, *first person shooters*, *third person shooters*, jogos de luta, *adventures* em primeira terceira pessoa e isométricos, entre outros.

Um sólido é um subconjunto fechado e limitado no espaço euclidiano tridimensional: E^3 [14]. Esta definição indica que um objeto deve ser representado em 3D (espaço tridimensional euclidiano) mesmo que uma ou duas dimensões sejam desprezíveis com relação a uma terceira dimensão. Além disso, o objeto deve ser fechado o que significa que deve apresentar um contorno e deve ser limitado, não apresentando nenhuma dimensão que seja infinita.

Um objeto pode ser representado pela descrição dos polígonos (também chamados faces) que definem a topologia deste objeto. As faces por sua vez podem ser definidas pelos vértices (suas coordenadas tridimensionais) que as compõem. Dessa forma, com a descrição de vértices e faces se pode definir uma malha (conjunto de dois ou mais faces que compartilham vértices) que representará o objeto. A figura 3.2 mostra uma malha de polígonos e vértices representando uma chaleira.

O triângulo é o polígono mais simples e de uma forma geral, é utilizado para representação de faces. A representação de faces por triângulos ocasiona menor uso de memória, menor tempo de renderização e se adapta a qualquer tipo de contorno [14]. Além disso, APIs gráficas utilizam triângulos como primitivas de renderização possibilitando modos de renderização otimizados

como *triangle fans* e *triangle strips* [11][15][16]. A figura 3.3 (a) mostra uma malha de polígonos e vértices representando um personagem.

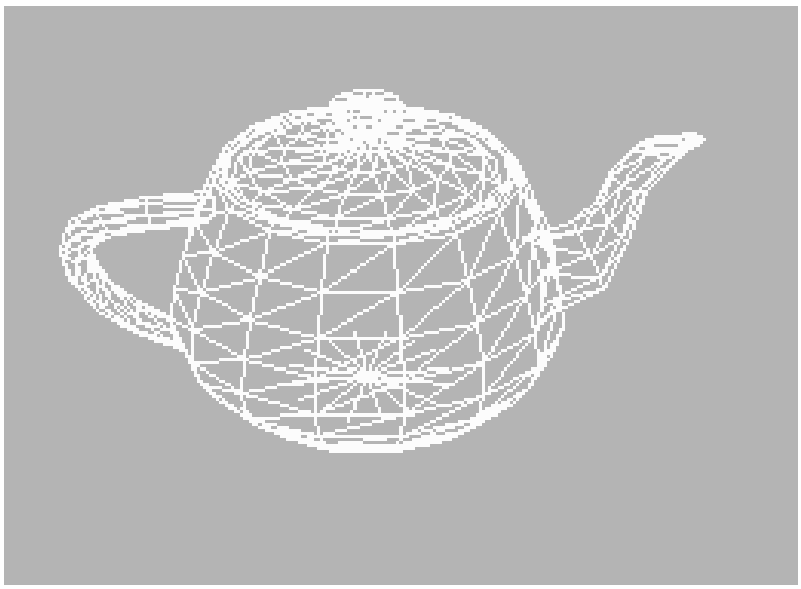

Figura 3.2. Malha de polígonos e vértices representando uma chaleira

Eventualmente as normais de polígonos e normais nos vértices também podem fazer parte da representação de um objeto e são usadas como informações de mapeamento de textura para o modelo de iluminação utilizado. Também podem ser encontradas informações para texturização dos polígonos formadores da superfície do objeto. Na maioria das vezes há uma indicação de arquivo 2D para a textura e informações de mapeamento de textura. A figura 3.3 (b) mostra um personagem texturizado.

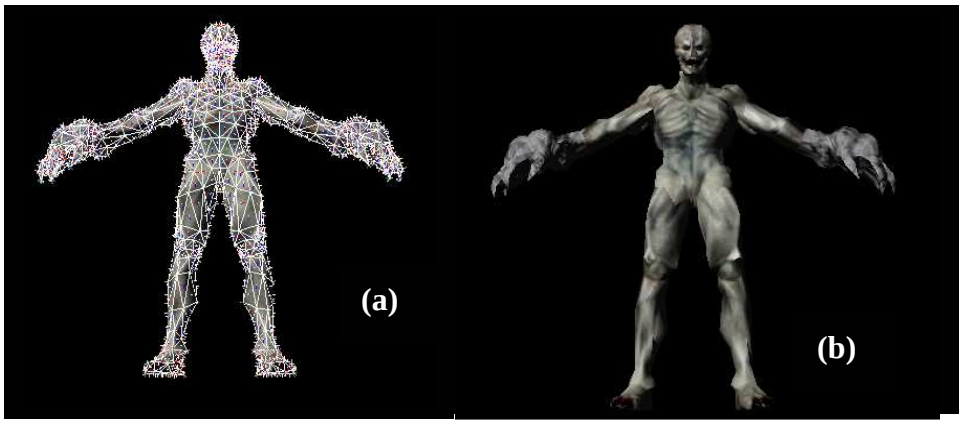

Figura 3.3. Malha de polígonos e vértices de um modelo de um personagem, aramado em (a) e texturizado em (b)

A mudança na forma de representação dos modelos dos personagens trouxe a baila outras questões como: estruturas de dados apropriadas à grande quantidade de dados que deve ser armazenada e manipulada, os métodos de animação a serem adotados e também como os personagens devem ser gerenciados durante a execução do jogo passaram a fazer parte das preocupações de equipes de criação e desenvolvimento de jogos.

Com o passar do tempo alguns conceitos e práticas foram se sedimentando no que se refere à representação de personagens em jogos 3D. Em um primeiro estágio, para cada jogo era criado um completo sistema de visualização 3D, em geral com acesso direto ao *hardware* da máquina como pode ser visto em [6] e [12]. Várias funções eram codificadas em linguagem *assembly* para permitir uma performance otimizada e acesso a primitivas do processador, constituindo soluções *ad hoc* para cada módulo do jogo [1].

Dois fatores modificaram esta forma de desenvolvimento. Em primeiro lugar as empresas de criação de jogos perceberam que muitas funcionalidades criadas para um jogo poderiam ser utilizadas em outros jogos. Essas funcionalidades podem ser agrupadas em bibliotecas possibilitando o reuso e otimizando assim o estágio de desenvolvimento do projeto de criação de um jogo. Esses fatores constituíram o estágio de industrialização dos jogos de computador.

Paralelamente, houve uma modificação na forma com que as aplicações se comunicam com o *hardware*. Com o advento de sistemas operacionais mais robustos com características de acesso protegido ao *hardware*, multi-tarefas e multi-usuários, as aplicações passaram a depender do sistema operacional para se comunicar com o *hardware*. Como exemplo pode-se citar a transição entre os sistemas operacionais DOS e Windows.

Os sistemas operacionais nem sempre permitem uma comunicação entre aplicação e *hardware* satisfatória em termos de desempenho, sobretudo para jogos. Por isso, foram criadas APIs que possibilitam um acesso mais otimizado

ao *hardware* (além de aproveitarem melhor as características de *hardware* mais sofisticadas como placas de vídeo com modos de renderização otimizados por *hardware*). Como essas APIs são padronizadas, o acesso ao *hardware* continua a ser protegido. Alguns exemplos de APIs são OpenGL[15] [16] e DirectX [17] [18].

Com o agrupamento das funcionalidades para desenvolvimento de jogos e a utilização de APIs gráficas surgiu o conceito de *game engine* para jogos de computador. Uma *game engine* (motor de jogo) é um programa que implementa uma determinada arquitetura para jogos. Através de configurações da *game engine* se pode determinar o aspecto do jogo [1]. Uma *game engine* pode ser implementada com base em um *framework.*

Um *framework* para jogos de computador é um conjunto de classes que forma uma estrutura reusável para o desenvolvimento de um jogo de computador [1].

O *framework* determina a estrutura da aplicação criada possibilitando pontos de personalização que dão uma liberdade maior para a criação do jogo do que uma *game engine* [1].

Frameworks e game engines para jogos 3D, atualmente, costumam ter um módulo de sistema de controle de animações de personagens que lida com todo o processo, desde a concepção visual e modelagem dos personagens, sua inteligência e comportamento e sua representação durante a execução do jogo. Sistemas de controle de animações de personagens podem ser visto em duas camadas principais de abstração, como está esquematizado na figura 3.4.

No alto nível, a inteligência e o comportamento do personagem é gerenciado. O nível mais baixo trata da obtenção e reprodução das seqüências de animação do personagem. O alto nível utiliza alguma tecnologia de Inteligência Artificial para cumprir a tarefa de dar uma “alma” aos personagens de jogo. Esse nível gera respostas aos eventos que ocorrem no ambiente onde

o personagem se encontra inserido e que podem afetá-lo de alguma forma [19][20]. Além disso, o alto nível é responsável por dar uma “personalidade” ao personagem expandindo em uma ou mais dimensões [21] as características sentimentais relacionadas à sua reação aos estímulos sofridos, como por exemplo, as dimensões amor-ódio, honestidade-desonestidade, etc.

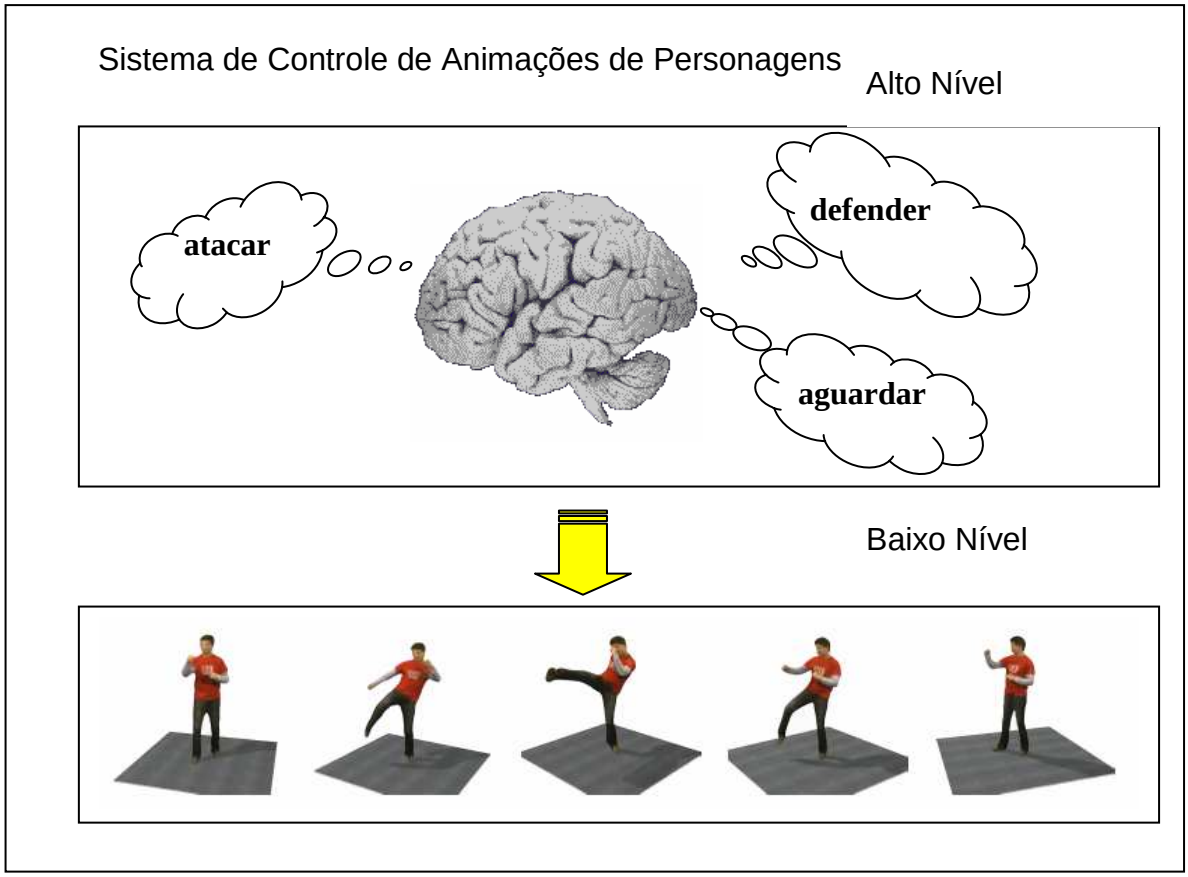

Figura 3.4. Sistema de controle de animações de personagens em duas camadas

Máquinas de estados finitos, lógica fuzzy e modelagem cognitiva baseada em arquiteturas de redes neurais são alguns dos elementos de inteligência artificial sobre os quais geralmente se alicerçam o alto nível [19]. Muitas vezes, o caráter de um personagem é descrito através de algum tipo de linguagem de *script* [19][20]. O *script* determina exatamente que comportamento um personagem irá incorporar dependendo das situações apresentadas durante a execução do jogo.

O baixo nível do sistema de controle de animações de personagens [22][14] responde às requisições do alto nível obtendo as seqüências de animação e as reproduzindo de forma conveniente. O baixo nível pode estar baseado em três tipo de animação: métodos procedurais, *motion capture* e *keyframing* [23][24].

Métodos procedurais são um conjunto de processos que possibilitam a criação de movimentos de personagens e objetos automaticamente durante a execução da aplicação [23][24]. Em geral, a geração dos movimentos é baseada em regras físicas ou naturais bastante precisas (como sistemas de partículas) ou ainda em sistemas de regras comportamentais e sociais (utilizado para representar bandos de animais, por exemplo), nesse caso também chamado de *behavioral animation* [19][22].

Uma vantagem dessa abordagem são os altos níveis de acurácia obtidos de sistemas de regras de movimentos de personagens e objetos, quando bem modelados, muito úteis para processos de simulação. Por outro lado, uma das desvantagens é a falta de interação entre modeladores e a animação propriamente dita. Além disso, pode ser muito oneroso em termos de processamento gerar animações em tempo real de um grande número de personagens, uma vez que a complexidade dos processos interfere diretamente no tempo de execução das aplicações.

Finalmente, em alguns casos, em jogos mais voltados ao entretenimento e com alguma dose de humor, o excesso de perfeição que as animações apresentam podem dar a impressão de que os movimentos são demasiadamente “mecânicos” ou “robóticos”. Devido a suas características, vantagens e desvantagens, os métodos procedurais não são utilizados para personagens individuais.

A animação de personagens individuais é geralmente realizada através de *motion capture* ou *keyframing*. *Motion capture* baseia-se na idéia de gravar os movimentos de um ator ou um objeto e então através de um processamento

dos dados obtidos digitalizá-los para a representação dos movimentos de um personagem.

Embora essa técnica apresente ótimos resultados com a alta fidelidade obtida dos movimentos reais capturados, ela nem sempre é uma boa escolha, sobretudo pelo custo do maquinário e dispositivos necessários para realizar gravações de movimentos com qualidade. Vale ressaltar ainda que esses mesmos dispositivos, em geral, são de alguma forma conectados ao corpo do ator ou do objeto cujos movimentos serão capturados, podendo trazer dificuldades à liberdade de movimento dos atores ou objetos. Além disso, o processamento de dados pós-captura não é uma tarefa trivial, podendo por isso também ser mais uma desvantagem [22].

Keyframing é uma técnica de animação baseada na antiga técnica de criação de animações, feitas à mão em desenhos animados da década de 1930 [25]. Era comum nessa época, dividir o processo de criação de animações em 2D em duas etapas. Na primeira, poses principais (chamadas *keyframes*), que definiam o movimento de um personagem em uma cena, eram desenhadas, a mão. Em seguida (que podia ser feito por outro artista) eram desenhadas poses intermediárias (*intermediate frames*) entre as poses principais para que o movimento apresentasse continuidade. Esse processo foi chamado de *inbetweening*.

A técnica de criação de *keyframes* e posterior complementação dos *intermediate frames* foi transportada para a produção de animações em 3D modernas com as devidas adaptações. Uma dessas adaptações diz respeito ao fato de que o processo de *inbetweening* é realizado pelo computador durante a fase de *design* ou posteriormente na fase de processamento, em tempo real.

Atualmente, *keyframing* é uma técnica normalmente realizada em dois estágios: *design* e processamento. No estágio de *design*, os modelos e as seqüências de animação são criados. As seqüências de animação podem ou

não ser estritamente baseadas no comportamento que o personagem exibirá durante a aplicação (ou seja, as seqüências podem ser bastante genéricas ou mais específicas). Seqüências de animação são conjuntos de *keyframes* que mostram ao longo do tempo como o modelo executa um movimento tendo como referência um *keyframe* inicial chamado *baseframe* [26] (*frame* de base).

Durante a execução do jogo, o estágio de processamento das animações entra em cena, respondendo as requisições por animações geradas no alto nível do sistema de controle de animações de personagens, escolhendo e reproduzindo corretamente as seqüências de animação. Geralmente, este estágio é também responsável por gerar *intermediate frames* entre pares de *keyframes*, fazendo assim o papel do processo de *inbetweening.* A Figura 3.5 mostra a abordagem de *keyframing* em dois estágios.

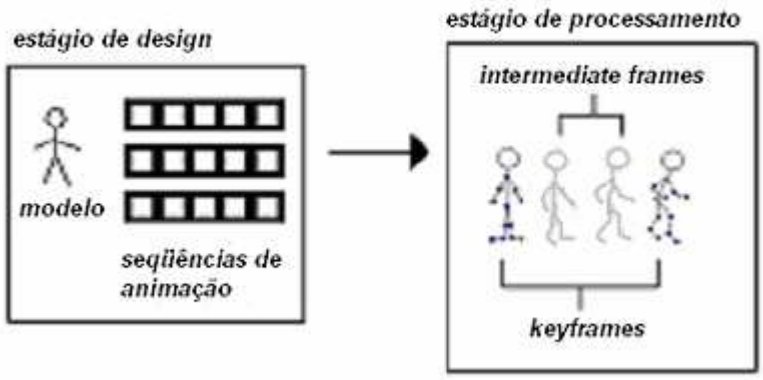

Figure 3.5. Abordagem de *keyframing* em dois estágios

Devida a pouca complexidade e ao baixo custo demandado pelos estágios de *design* e processamento, a técnica de *keyframing* ainda é a forma mais popular de prover animações no baixo nível.

3.1 Evolução do processo de animação keyframing para jogos em 3D

Uma questão a ser considerada no processo de animação é a utilização racional de memória e de processamento. Se por um lado é menos complexo utilizar-se vários modelos armazenados em memória, por outro lado, é menos

restritivo, em termos de memória, criar-se a movimentação do modelo em tempo real, o que pode, por outro lado, comprometer o processamento do sistema como um todo.

Assim é possível identificar duas abordagens que servem como extremos para situar as técnicas de representação das animações utilizadas para *keyframing* em jogos 3D. São elas o *vertex animation* e o *skeletal animation*.

No *vertex animation* cada *keyframe* que o modelo do personagem assume durante o movimento é criada com uma malha de vértices e polígonos completa [14] [27]. A animação se dá através da sobreposição de cada *keyframe* ao *keyframe* anterior respeitando o tempo e taxa e *frames* desejada. O problema dessa técnica é a grande quantidade de memória que será utilizada para guardar todos os *keyframes* de todos os movimentos de cada personagem de uma cena, uma vez que cada *keyframe* é definido por todos os vértices do modelo.

Por outro lado, o *skeletal animation*, é uma técnica em que o personagem é definido através de uma única malha de polígonos e vértices e de regras de hierarquia entre pontos definidos (juntas) de sua estrutura que permitem a criação de relações de hierarquia entre partes do modelo. Essa abordagem pode, em princípio, permitir que qualquer novo movimento seja gerado pelo computador durante o processamento. Para que movimentos estranhos não sejam criados pelo computador é preciso que sejam definidos certos limites de movimentação entre as juntas (chamados graus de liberdade das juntas [28]).

Na prática nenhuma dessas duas abordagens é utilizada plenamente. As soluções encontradas pelos desenvolvedores de jogos têm sido formas híbridas entre as duas.

A primeira solução encontrada foi à adoção parcial do sistema *vertex animation* utilizando interpolação entre *keyframes*. Como não se pretende guardar todas as poses para cada movimento de um modelo de personagem,

os *keyframes* de um personagem são criados e durante o processamento algum tipo de interpolação é utilizado para a representação dos *intermediate frames* (*inbeetwing*) para a composição da animação.

Esta abordagem, conhecida como *vertex blending*, utiliza a memória para guardar os *keyframes, mas* utiliza também a CPU para descobrir os *intermediate frames*. Os pontos altos são a facilidade de implementação e coesão visual, isto é, os personagens não são vistos pelo usuário como pedaços “colados”. As desvantagens são o consumo de memória que ainda é grande e a falta de flexibilidade na geração de novas animações não pensadas pelo modelador no estágio de *design*.

Uma outra solução, utilizada também em jogos mais antigos, foi a definição hierárquica dos modelos, *hierarchical animation*. Ela é baseada na técnica de *skeletal animation,* mas somente no que tange a definição de partes do corpo, pois em geral, da mesma forma que o método anterior, as poses que definem um movimento são geradas na fase de *design*. Contudo, não é necessário armazenar a malha de cada pose, somente as informações de rotação e translação de cada junta, as quais serão aplicadas em cada parte do modelo que for subordinado a essa junta. Rotações e translações podem ser interpoladas para possibilitar uma animação mais suave.

A melhor utilização de memória por essa abordagem a torna muito atrativa. Contudo surgem dois problemas: a necessidade de um método mais sofisticado para interpolação de rotações (como o uso de quatérnios e a interpolação linear esférica) e também a falta de coesão visual entre as partes do modelo. Isso acontece nas juntas onde existe a cisão de duas partes do corpo. Quando o personagem, por exemplo, gira o seu antebraço, a junta da posição do cotovelo pode mostrar a fissura entre essas duas partes do modelo.

Com o passar do tempo soluções que uniam as vantagens de cada uma dessas abordagens foram propostas visando uma utilização mais eficiente de

memória e processamento, além de representação mais realística dos movimentos dos personagens.

As abordagens hierárquicas tornaram-se uma unanimidade. Mas para resolver o problema de fissuras nos pontos de juntas dos modelos uma solução baseada na interpolação de posição dos vértices foi encontrada. As partes do corpo (conjuntos inteiros de vértices) não mais possuem informações de hierarquia, mas todos os vértices contêm suas próprias informações de associação hierárquica a uma ou mais juntas com atribuições de pesos a essas associações. Esse método costuma ser chamado de *mesh skinning* ou somente *skinning*.

O que se segue é uma descrição da evolução da técnica de *keyframing* aliada a outras técnicas de otimização para que o processo de animação em jogos 3D chegasse ao que se tem hoje na área. Para cada nova técnica é também apresentado um estudo de caso baseado em um modelo que exemplifica a utilização dessa técnica.

3.2 Animações baseadas em interpolação de posições de vértices

Animações baseadas na interpolação de posições de vértices ou *vertex animation* é uma técnica onde os *keyframes* são criados e armazenados como uma malha completa do modelo do personagem. O que diferencia um *keyframe* de outro são as posições que um determinado vértice possui em um e no outro. Dados os *keyframes*, outras posições intermediárias são derivadas por um processo de interpolação [29][30]. Um exemplo é mostrado na figura 3.6 onde as malhas 1, 2 e 3 se alternam para criar a ilusão de movimento.

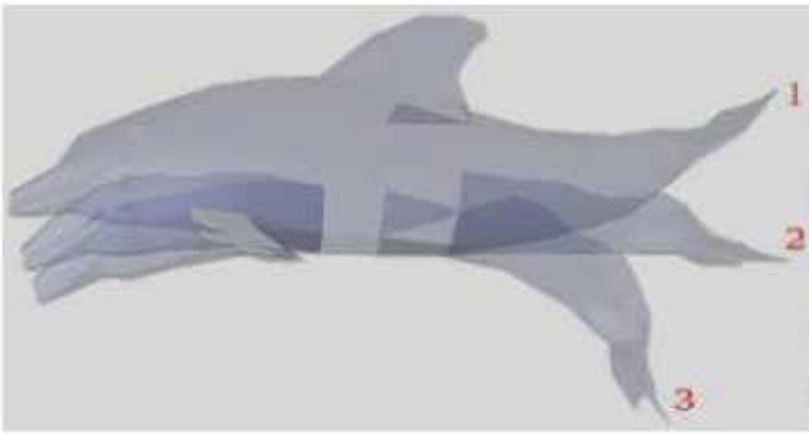

Figura 3.6. Interpolação de posição de vértices [30]

O problema da interpolação entre duas posições é resolvido facilmente por abordagens de interpolação cujo parâmetro é o tempo. A forma mais simples de interpolação é a linear, que produz uma posição intermediária à duas posições de interpolação dadas, como função linear do tempo. O problema dessa interpolação é que ela produz resultados não suaves entre os pontos de interpolação causando mudanças bruscas entre estes pontos [11].

Uma abordagem mais elaborada pode ser utilizada utilizando curvas paramétricas para realizar a interpolação. Um exemplo é uma interpolação baseada em *splines*, como a *Spline* de Catmull-Rom [14] [23] [27] [31]. As duas abordagens podem ser vistas na figura 3.7. Mais informações sobre curvas paramétricas utilizadas para interpolação podem ser vistas no Apêndice B.

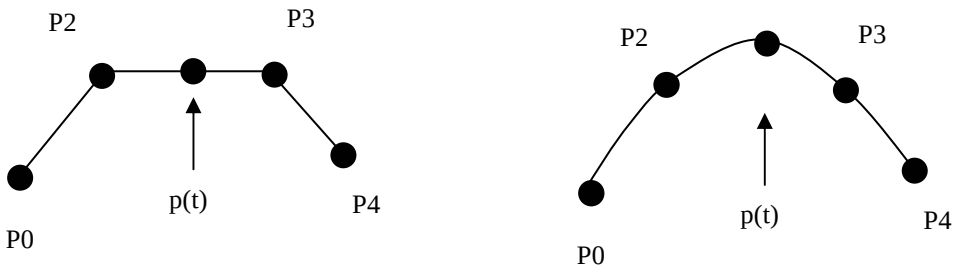

Figura 3.7. Interpolação linear e a *Spline* de Catmull-Rom

Algumas API's para criação de jogos em 3D permitem a automatização da interpolação entre pontos. O DirectX (versões 7 em diante) por exemplo, possui funcionalidades que permitem a utilização de uma função chamada *vertex shader* para a interpolação linear entre vértices (também chamada "*Tweening*") e também permitem a definição de outros métodos de interpolação [30] [32].

3.2.1 O formato Model Quake 2

O formato Model Quake Doom 2 (popularmente conhecido como md2) é o modelo criado pela IdSoftware [10] e utilizado no jogo Quake2. Ele é completamente baseado na idéia de interpolação de posições de vértices. Boas fontes de informação de como interpretar esse arquivo e animá-lo são [33], [34] e [35]. Aqui se pretende mostrar características gerais desse formato, apresentando-o como um modelo de animação por interpolação de posições de vértices.

A figura 3.8 mostra a estrutura de um arquivo md2 de forma esquemática.

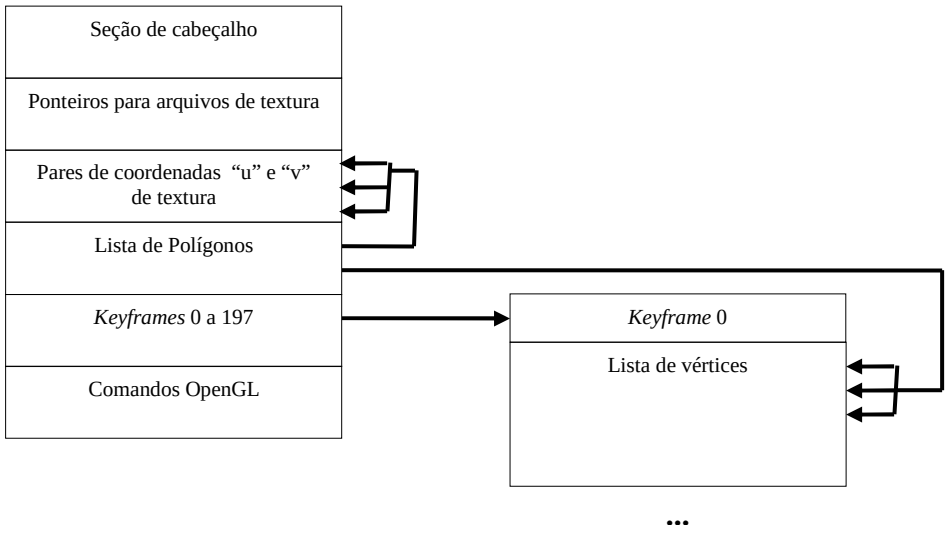

Figura 3.8. Modelo md2

Um modelo md2 é constituído de 198 *keyframes* de animação. Esses *keyframes* estão divididos em 20 diferentes movimentos. Cada seqüência de

animação que representa um movimento tem uma lista bem definida de *keyframes*. Por exemplo, a seqüência "*standing idle"* pode ser definida com os *keyframes* de 0 a 39 [34].

A tabela 3.1 mostra os *keyframes* utilizados para cada seqüência de animação. Pode-se observar como o modelo é limitado, uma vez que possibilita apenas as seqüências listadas na tabela.

Seqüência de animação	*Índices de* keyframes
STANDING_IDLE	*0 a 39*
RUN	*40 a 45*
ATTACK	*46 a 53*
PAIN 1	*54 a 57*
PAIN 2	*58 a 61*
PAIN 3	*62 a 65*
JUMP	*66 a 71*
FLIP	*72 a 83*
SALUTE	*84 a 94*
TAUNT	*95 a 111*
WAVE	*112 a 122*
POINT	*123 a 134*
CROUCH STAND	*135 a 153*
CROUCH WALK	*154 a 159*
CROUCH ATTACK	*160 a 168*
CROUCH PAIN	*169 a 172*
CROUCH DEATH	*173 a 177*
DEATH BACK	*178 a 183*
DEATH FORWARD	*184 a 189*
DEATH SLOW	*190 a 197*

Tabela 3.1. Seqüências de animação utilizadas pelo modelo md2

Cada *keyframe* é na verdade uma malha completa do modelo. O modelo é definido como uma lista de triângulos e uma lista de vértices. Cada triângulo é definido por 3 vértices da lista. Todos os *keyframes* devem conter o mesmo número de vértices já que definem o mesmo modelo em uma pose diferente.

O triângulo está associado a uma lista de 3 coordenadas de textura que permitem o mapeamento de textura. Um único arquivo, uma imagem 2D em formato PCX (formato de arquivo bitmap criado pela ZSoft [6]), é utilizado para a texturização de todas os *keyframes* contidos no modelo, embora o arquivo de modelo possa conter indicações para vários arquivos de texturas (são "*skins*" diferentes em cores da mesma textura que podem ser utilizadas por diferentes personagens,). Assim cada triângulo indexa um par de informações (u, v) de mapeamento de texturas respectivas a cada um dos vértices associados a ele.

Cada *keyframe* contém uma lista de vértices onde a única diferença entre tais listas são as posições de cada vértice em 3 dimensões e o vetor normal neste vértice. As posições definem a forma que o modelo terá naquele *keyframe*. Além disso, cada *keyframe* contém informações sobre escalonamento e translação que são importantes para as transições entre *keyframes*.

A figura 3.9 mostra um exemplo de arquivo de textura (*skin*) para um arquivo md2 e a figura 3.10 mostra um modelo md2 sendo animado.

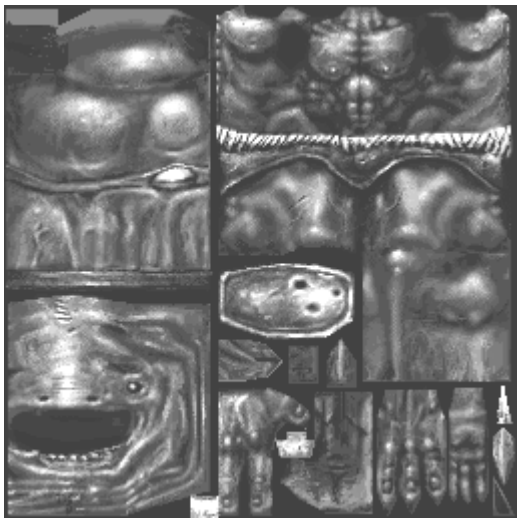

Figura 3.9. Textura de modelo md2 [33]

Entre outras coisas, um arquivo de modelo md2 pode conter também indicações para arquivos de som no formato WAV (WAVEform audio format da Microsoft e IBM), ícones e algumas informações que permitem a utilização otimizada da biblioteca OpenGL (definições de *fans* e *strips* de triângulos).

Figura 3.10. Um modelo md2 animado [33]

Para proceder a uma animação do modelo md2 um sistema deve ler as informações do arquivo do modelo corretamente e armazená-lo em uma estrutura que permita a definição de escolha de movimentos, o que em geral é feito com uma máquina de estados.

Como cada seqüência de animação contém um número de *keyframes* para a representação do movimento, procede-se a uma interpolação, para cada vértice, a fim de encontrar a posição intermediária entre duas posições definidas por dois *keyframes* consecutivos. O parâmetro do processo de interpolação costuma ser baseado no tempo em que se deseja que a animação se processe e taxa de *frames* por segundo esperada.

3.2.2 O formato Model Quake 3

O formato Model Quake 3 (popularmente conhecido como md3) foi criado pela idSoftware para o jogo Quake3 Arena [10]. Este formato é um interessante exemplo de abordagem híbrida entre a animação baseada em interpolação de posições de vértices e animação baseada em animação hierárquica. Esse é inclusive o motivo da controvérsia entre classificar este formato como um exemplo de uma ou outra técnica.

Ele será mostrado como um exemplo de interpolação de posições de vértices pelo motivo de não ser um formato completamente baseado em hierarquia. Em [35], [36] e [37] podem ser encontradas boas descrições de como interpretar e animar modelos criados neste formato.

O formato md3 é baseado em modelos armazenados em arquivos diferentes, cada um deles responsável por uma seção da hierarquia do personagem. O modelo md3 possui uma hierarquia fixa, definida pelas seguintes partes: pernas (ou *lower torso*), tronco (ou *upper torso*), cabeça e arma. A parte tronco é subordinada à parte pernas e as partes cabeça e arma são subordinadas à parte tronco. A hierarquia é mostrada na figura 3.11.

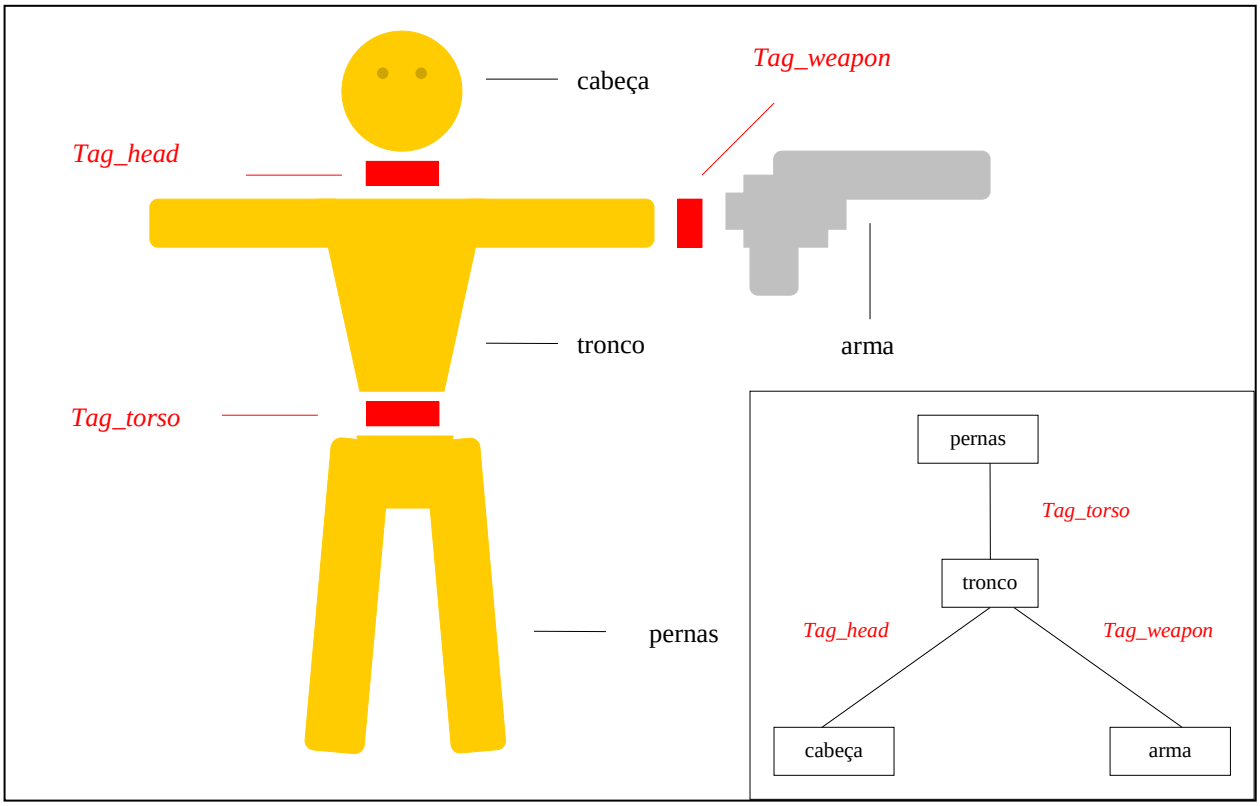

Figura 3.11. Hierarquia do modelo md3

Os arquivos relacionados às pernas e ao tronco contêm malhas de vértices e polígonos para cada *keyframe* de animação (semelhante ao formato md2). Os movimentos são controlados por um arquivo de configuração onde alguns movimentos são definidos como sendo independentes entre pernas e tronco, mas, há também movimentos que ambos devem realizar conjuntamente.

Na prática isso significa que, para muitos movimentos, há certa independência entre essas duas partes do corpo e é possível combinar movimentos das pernas com movimentos dos braços. Por exemplo: as pernas podem executar o movimento de dançar enquanto o tronco (que contém os braços) pode executar o movimento de socar. Contudo, para certos movimentos como, por exemplo, o personagem cair no chão morto, as pernas e o tronco devem executar as respectivas animações deste mesmo movimento para que haja coerência na animação.

A cabeça também é representada por um arquivo próprio. Mas ela não contém malhas extras definindo a animação da cabeça, ou seja, apenas uma malha é definida, como se a cabeça apresentasse sempre a mesma pose. Por esse motivo alguns modeladores de personagens preferem “esconder” a cabeça do personagem, em geral a colocando dentro do tronco, e criar na malha do tronco uma outra cabeça, essa sim que será vista [36]. Assim, os movimentos da cabeça com relação ao tronco são definidos pelo modelador no conjunto de malhas de animação do tronco. Repare que esse atalho faz com que o modelo fique ainda menos hierárquico do que ele já era.

O arquivo de arma, diferentemente de todos os outros, contém informações de animação, mas na forma de um *script* próprio. Na maioria dos casos, essas armas são representadas de forma estática tornando desnecessária a utilização desse subterfúgio.

Cada parte do corpo referencia pelo menos um arquivo de textura. Assim devem existir pelo menos quatro arquivos de textura associados a um modelo.

As informações de hierarquia que são passadas de partes “pai” para partes “filhos” são chamadas de *tags*. Cada *keyframe* da parte “pernas” do modelo contém informações, chamadas *tag_torso*, que devem ser passadas para a parte “tronco” relacionadas às rotações e translações herdadas. Cada *keyframe* da parte “tronco” do modelo contém informações, chamadas *tag_head* e chamada *tag_weapon*, que devem ser passadas respectivamente para as partes “cabeça” e “arma” relacionadas às rotações e translações herdadas. Além disso, rotações e translações herdadas por “tronco” de “pernas” também são passadas para as partes “cabeça” e “arma”.

A fim de animar um modelo md3, para cada parte do modelo, deve-se proceder a interpolação das translações entre *keyframes* e as translações herdadas. Além disso, as rotações herdadas devem ser interpoladas. A interpolação de translações pode ser feita por interpolação linear como já foi citado antes. Contudo, para interpolação de rotações outro processo deve ser

utilizado porque a interpolação linear causa resultados errados para rotações, conforme será visto na seção a seguir.

Um exemplo de modelo md3 pode ser visto na figura 3.12.

Figura 3.12. Modelo md3 animado [36]

3.3 Animação baseada em hierarquia

Pode-se dizer que a animação baseada em hierarquia é mais um passo dado no sentido de associar realismo à representação de animações em 3 dimensões.

A animação baseada em hierarquia (chamada, em certos textos, de *skeletal animation* ou também *bone animation* [14]) utiliza a idéia de que uma parte do corpo a ser animada tem seu movimento dependente do movimento de uma parte superior na hierarquia. Na prática as transformações de rotação e translação sofridas por uma parte do corpo do personagem são passadas recursivamente para todas as partes direta ou indiretamente subordinadas a ela.

A ligação entre duas partes, que define a hierarquia entre essas duas partes é chamada de junta. Assim, uma parte superior transmite transformações às partes inferiores diretamente ligadas a ela através de juntas.

Quando se trata de seres vivos costuma-se chamar a estrutura hierárquica completa de esqueleto, como o da figura 3.13, e cada ligação virtual entre duas juntas de osso. Por outro lado quando se trata de um objeto não vivo, costuma-se chamar sua estrutura hierárquica de corrente e as ligações entre as juntas são então chamadas de elos [28].

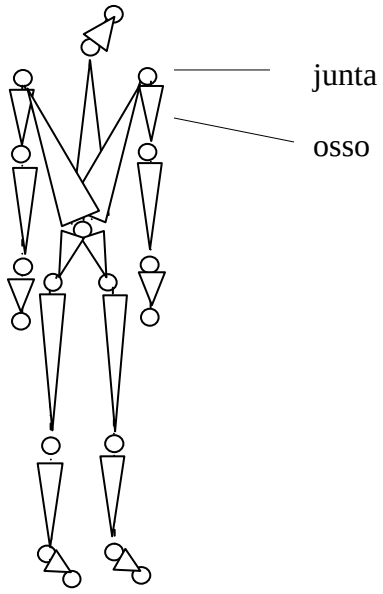

Figura 3.13. Esqueleto (estrutura hierárquica de juntas)

A animação hierárquica permite que somente as informações de rotação e translação sejam guardadas na transição entre duas poses de um personagem. Dessa forma podem-se criar animações baseadas em *keyframes* sem que seja necessário armazenar uma malha do modelo para cada *keyframe* na memória. A hierarquia permite compactar as informações de transição entre um *keyframe* e outro. Uma vez que todos os vértices de uma parte do corpo transladam e giram conjuntamente, pois essas partes são rígidas, não é necessário guardar informações para cada vértice, basta que para uma determinada parte se saiba quanto transladar e girar.

Um ponto interessante do qual jogos em 3D podem tirar proveito quando se trata de animação baseada em hierarquia é o fato de que, como as animações são em geral baseadas em uma única malha e adicionais informações de rotação e translação, é possível que tais informações possam ser utilizadas por vários tipos de personagens que possuam a mesma estrutura hierárquica, bastando que para isso seja modificado apenas a malha do *frame* de base. Dessa forma, é possível que personagens que possuam o mesmo esqueleto possam também utilizar as mesmas seqüências de animação diferenciando-se apenas nos modelos (malha).

Uma outra abordagem para animações utilizando estruturas hierárquicas é a representação realística de movimentos em tempo real [38] [39] [40]. Essa abordagem permite que as partes do corpo do personagem respondam a estímulos do ambiente ou a comandos do “cérebro” (alto nível) do personagem e realizem movimentos baseados em estudos e modelagem de movimentos humanos considerando centros de massa, inércia e balanço [35].

Neste caso, é necessário que as juntas sejam definidas na forma como os movimentos dos objetos ligados a elas serão permitidos, pois respostas originais como posições não previstas em tempo de *design* podem ocorrer. O tipo da junta define os graus de liberdade (se e quanto um objeto pode se mover em determinada direção) dos objetos ligados a ela [28].

As abordagens de implementação de hierarquia são duas: *foward kinematics* e *inverse kinematics* (figura 3.14) [35]. *Foward Kinematics* é a maneira como em geral os jogos programam as animações em hierárquias baseadas em *keyframes*, como já foi visto. A idéia é determinar a posição de cada parte do modelo baseado nas informações de rotações já conhecidas de cada junta [14].

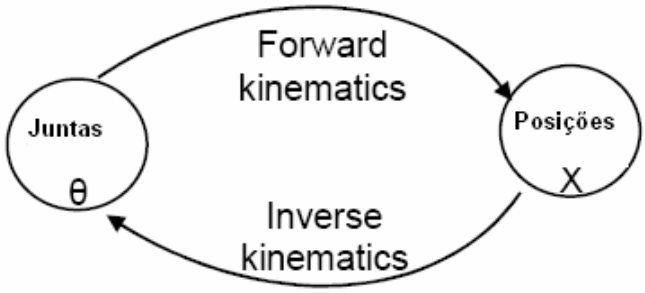

Figura 3.14. *Forward Kinematics* e *Inverse Kinematics*

Inverse Kinematics é um processo onde, dada a posição de uma determinada parte da estrutura, tenta-se inferir quais os ângulos que cada junta ascendente, a partir da junta desta parte, deve assumir para que essa posição seja encontrada. Este é um processo mais complexo que o anterior e pode gerar mais do que uma única solução.

A abordagem inversa é muito utilizada em aplicativos de criação de modelos em 3 dimensões, pois permite uma definição mais intuitiva de movimentos [41]. Contudo, em jogos, ainda não é uma unanimidade por sua programação ser mais complexa, sobretudo com a grande quantidade de juntas que um modelo de personagem humano costuma ter.

Existem, porém, algumas abordagens alternativas de implementação que tentam otimizar o processo de inferência de informações de rotações das juntas no sentido de propiciar a utilização de *inverse kinematics* em jogos em 3D como em [42].

Atualmente, a utilização de hierarquia em algum grau nos modelos de personagens para jogos 3D, tornou-se uma regra. Um exemplo de formato baseado em hierarquia é o Carcará Character Animation [43]. Neste caso, um modelo é criado e animado no 3D Studio Max, gerando *keyframes* da animação. Esse arquivo é exportado para um formato que então é processado de forma que os *keyframes* são transformados em informações de rotação.

Apenas a malha da posição inicial do personagem é guardada juntamente com as informações de rotação e translação de cada junta.

Como será mostrado, o formato MilkShape 3D que é bastante popular entre a comunidade de modeladores, também funciona baseado nesta abordagem.

3.3.1 Interpolações de translações e rotações

Diferentemente das animações baseadas em interpolação de posições de vértices, animações baseadas em hierarquia possuem não somente translações (como as anteriores), mas também rotações armazenadas como informações de um *keyframe*.

Por uma questão de economia na utilização de memória, é comum fazer com que o estágio de processamento calcule posições interpoladas entre *keyframes*. Assim o modelador pode representar poses realmente importantes, durante a animação com os *keyframes*, deixando para o estágio de processamento a tarefa de preencher as lacunas entre *keyframes* com *intermediate frames* fazendo a animação mais longa e suave.

Como foi mostrado na seção 3.2, a interpolação de transições pode ser feita linearmente (interpolação linear) ou com a utilização de *splines* para obtenção de resultados mais suaves entre os pontos de interpolação. Da mesma forma, existe a intuição natural de se representar informações de rotação como ângulos de Euler [11][44] e proceder à interpolação linear dessas rotações. Contudo, como pode ser visto no apêndice B, a representação de ângulos de Euler é inadequada pois pode causar o efeito chamado *gimbal lock* além do que a interpolação linear dos ângulos não ser uma forma natural de representar movimentos.

A solução para o problema de representação de rotações foi encontrada na matemática de quatérnios [11] [35] [44] [45] [46].

Os quatérnios são um grupo numérico cujas propriedades permitem a representação de rotações de forma adequada.

Tendo representado rotações como quatérnios (na verdade utiliza-se um subgrupo de quatérnios unitários), é possível fazer uso da operação conhecida como *Spherical Linear Interpolation* (*Slerp*) para interpolar essas rotações [11] [22][46].

Os quatérnios podem ser convertidos para a representação matricial de forma eficiente facilitando assim o processo de concatenação entre interpolações de rotações e translações para o cálculo da posição final de cada vértice do modelo do personagem. No apêndice B encontra-se uma explicação mais aprofundada de quatérnios e *Slerp*.

Embora, os quatérnios sejam menos intuitivos do que a representação de ângulos de Euler [46] e a S*lerp* seja mais difícil de implementar e mais dispendiosa em termos de processamento do que a interpolação linear (alguns estudos mais aprofundados sobre o impacto da utilização da S*lerp* para a performance de uma aplicação em tempo real podem ser encontrados em [47] e [48].), ainda assim suas vantagens valem os esforços e por isso a representação de rotações por quatérnios tem se tornado mais e mais popular entre jogos 3D.

3.3.1 O formato MilkShape 3D

O MilkShape é um *software* de modelagem muito famoso entre os modeladores. O formato MilkShape 3D é o formato nativo deste *software* mas ele pode ser exportado para vários formatos de jogos em 3D famosos. Como

exemplos podem-se citar: Quake III (Arena) [49], Half-Life [50], Max Payne [41], Unreal [51].

Diferentemente do formato md3, o MilkShape 3D é completamente baseado em hierarquia. Vários *keyframes* para animações são armazenados sob a forma de informações de rotação e translação de juntas às quais estão associadas objetos ou partes de um corpo.

Uma descrição bastante abrangente do modelo MilkShape 3D pode ser encontrada em [52] e [53], e em [54] pode ser encontrada uma implementação de animação com esse formato.

O arquivo do MilkShape 3D é um arquivo binário (como também o md2 e o md3). O modelo é baseado em triângulos. Os triângulos formam conjuntos de malhas (grupos) que são as partes do corpo do modelo. Um grupo também está associado a um único material que contém informações de iluminação e um identificador do arquivo de textura.

Os triângulos estão associados a três vértices que os definem espacialmente. Além disso, os triângulos contêm as informações de normais nos vértices e mapeamentos de textura. Os vértices contêm exclusivamente suas posições (x, y e z).

Cada vértice está associado a uma junta. Além disso, as juntas também mantêm uma hierarquia entre si. E é através dessas associações que a hierarquia do modelo se faz presente. É interessante notar, contudo que pode haver vértices que não estão ligados a nenhuma junta.

Obrigatoriamente todo modelo contém um nó raiz o qual serve para referenciar todas as outras juntas no espaço do modelo. As juntas estão associadas a *keyframes* (quadros-chave) que guardam informações de rotação e translação dessa junta para determinada pose relativa à posição inicial do modelo.

Vértices e as normais nestes vértices são transformados pelas informações contidas nas juntas. Essas informações são transformações herdadas de transformações interpoladas entre a pose atual e a nova pose a ser atingida.

Repare que ao contrário do formato md2, o formato MilkShape 3D pode possuir várias texturas associadas a um modelo ao mesmo tempo. Uma diferença importante entre o md3 e o MilkShape 3D é que a hierarquia neste último não é fixa e pode ser definida à vontade pelo modelador.

Diferentemente de Carcará Character Animation [43], não há uma associação direta entre partes do corpo (grupos) e juntas. A associação é feita diretamente entre vértices e juntas o que possibilita maior realismo na animação com a resolução de problemas de fissuras nas regiões das juntas dos modelos, conforme será explicado na seção seguinte.

A figura 3.15 mostra um exemplo de modelo animado baseado no formato MilkShape3D [54].

Figura 3.15. Modelo baseado no formato MilkShape 3D

3.4 Técnicas de Skinning

Abordagens tradicionais baseadas em animação por hierarquia apresentam uma desvantagem que a animação baseada em interpolação de posições de vértices não compartilha. Como as partes do corpo são modeladas como objetos rígidos, determinados movimentos fazem com que na região das juntas apareçam falhas (fissuras), pois não há uma malha que ligue esses dois objetos [42] [55], como pode ser visto na figura 3.16.

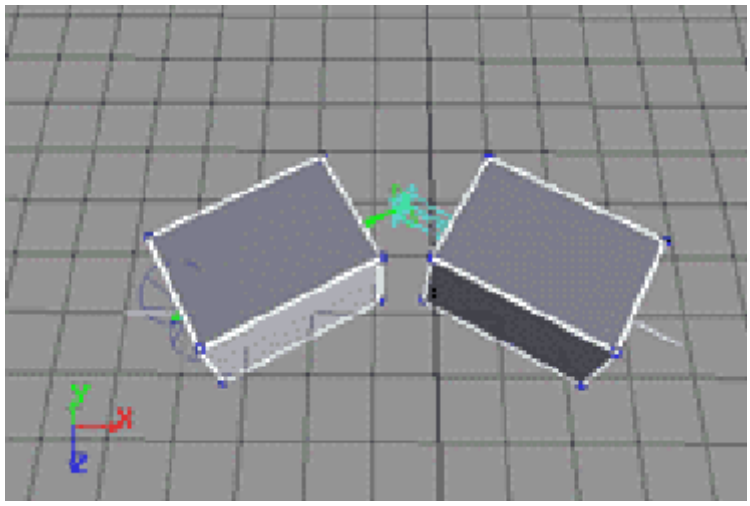

Figura 3.16. Fissura entre partes do corpo na região da junta [55]

Por esse motivo foi muito comum em alguns jogos em 3D os personagens terem partes do corpo com vestimentas que encobriam a região das juntas. Observando mais atentamente, pode-se ver que quando se associa uma parte do corpo a uma junta, está associando-se todos os vértices de todos os polígonos formadores dessa parte do corpo àquela junta. E por isso, todos os vértices são solidários a apenas uma junta e nenhum polígono é capaz de funcionar como uma “cola” entre as duas juntas [56].

Essa observação fez nascer uma primeira solução para esse problema, chamada *single weight vértices* [40], *rigid skinned characters* [57] ou *stitching* [58]. Essa técnica se baseia na idéia de que vértices de uma mesma parte do

corpo podem estar associados a juntas diferentes. Assim, os polígonos que estão associados às duas juntas, posicionados entre elas, podem ser “esticados” ou “encolhidos” funcionando como ligação elástica entre elas, como pode ser visto na figura 3.17. Um exemplo dessa abordagem é o formato MilkShape 3D, já citado.

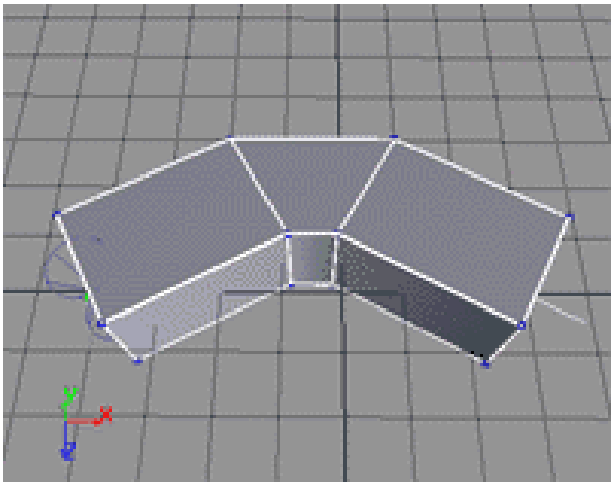

Figura 3.17. Polígonos servindo como ligação elástica entre as partes do corpo que se encontram em uma junta [55]

Embora essa solução resolva o problema das fissuras, a parte do corpo que serve como liga na região da junta pode se distorcer de forma visualmente não factível dependendo do movimento realizado. Um exemplo pode se visto na figura 3.18.

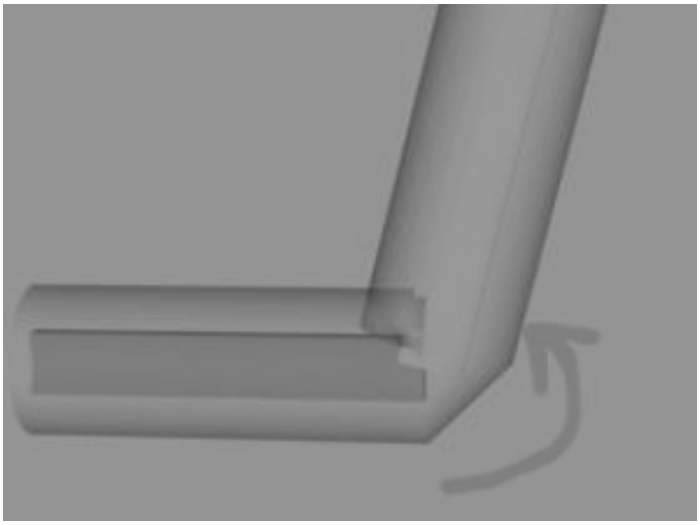

Figura 3.18. Distorção inconveniente da “liga” na região da junta [40]

A solução encontrada foi associar cada vértice não a duas, mas a várias juntas. Cada junta contribuindo com um peso para o movimento desse vértice. Os pesos somam o valor total de 1 (100%). Essa técnica é conhecida como *soft skinned characters* [42], *multiple weighted vértices for smoth skinning* [40] ou simplesmente *skinning* [58].

Na figura 3.19, pode-se observar como através dessa técnica é possível ter ligas na região das juntas, que são mais flexíveis. Na figura 3.23 pode ser visto como essa técnica resolve o problema de distorções indesejáveis aproximando-se muito do efeito visual causado pela flexão de musculatura do corpo humano. Um exemplo de formato que utiliza essa solução é o md5, que será visto mais adiante.

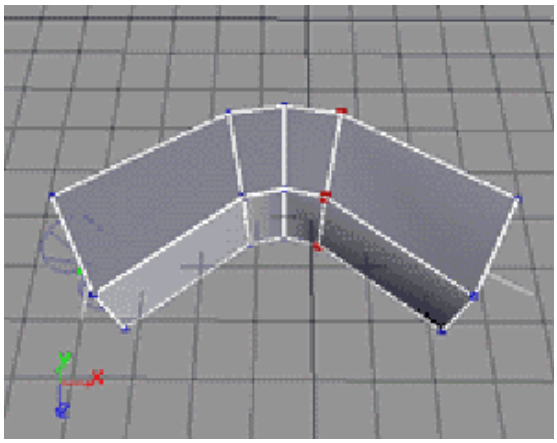

Figura 3.19. Vértices influenciados por mais de uma junta por pesos [55]

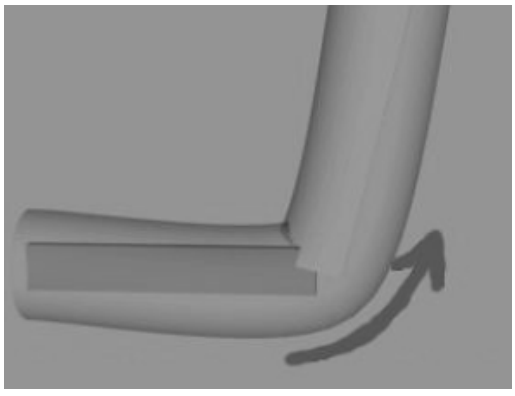

Figura 3.20. Resolução do problema de distorções indesejáveis [40]

Essa técnica torna o consumo de memória um pouco maior, pois as estruturas de dados de vértices devem conter, para n juntas, n -1 pesos (o último é igual a um menos o somatório de todos os outros). Também com relação ao desempenho ela acrescenta um razoável *overhead* no processamento da animação.

Algumas APIs de programação, voltadas para computação gráfica e jogos, disponibilizam funções que permitem acessar funcionalidades de *hardware* capazes de otimizar o processo de cálculo da posição de vértices com pesos. Um exemplo é o DirectX (versões 7 em diante) [32].

Em [22] é mostrado que as técnicas de *skinning* podem ser entendidas como aplicações específicas de abordagens genéricas para o problema de animação de objetos flexíveis. De fato, o problema de ligação entre as partes do modelo é na verdade um problema de criação de uma animação para uma superfície flexível que se distorce conforme as partes do corpo que a dividem se movem (se afastam ou se aproximam).

3.4.1 O formato Model Doom 3

O formato Model Doom 3 (conhecido popularmente como md5) foi criado pela IdSoftware [59] para modelos animados do jogo Doom3. Ele possui uma estrutura sofisticada de animação hierárquica com *skinning* baseado em pesos, modelando as rotações por quatérnios.

O formato do arquivo principal é em ASCII (texto puro) e define o esqueleto e um conjunto de malhas que definem as partes do corpo do personagem. As animações propriamente ditas estão separadas em vários arquivos também em ASCII. Isso possibilita compartilhar animações com diferentes modelos quando estes compartilham a mesma estrutura hierárquica [59]. A figura 3.21 mostra como está subdivido esse modelo. As figuras 3.22 e

3.23 mostram respectivamente como é formado o arquivo de malhas e como são formados os arquivos de seqüências de animação.

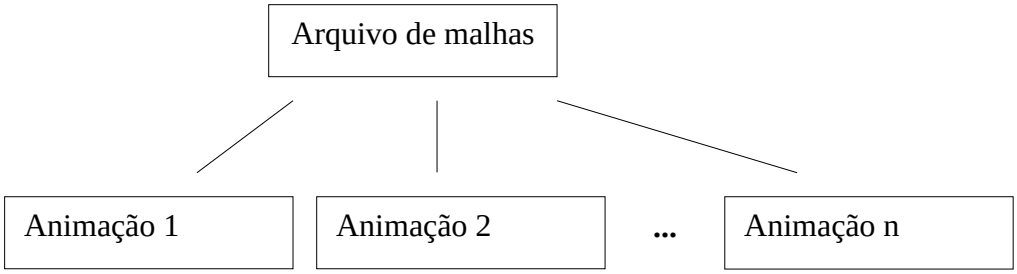

Figura 3.21. O modelo md5 dividido em um arquivo de malhas e os arquivos de seqüências de animação

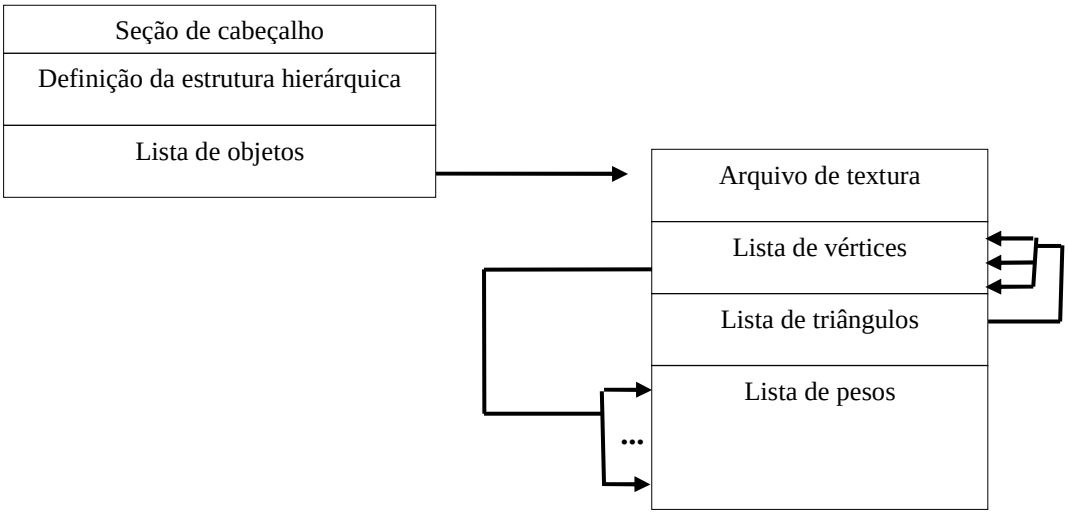

Figura 3.22. Estrutura do arquivo de malhas

A lista de juntas define o esqueleto do modelo. Cada junta tem um rótulo, como “Hips”, uma posição (x, y, z), uma orientação e uma ligação com uma junta superior (junta “pai”). A única junta que não possui uma junta “pai” é a junta raiz obrigatoriamente existente em cada modelo, geralmente chamada “*origin*”. A orientação fica codificada em três valores em ponto flutuante que são os três componentes de um quatérnio unitário (o quarto componente pode ser encontrado através dos outros três, sendo oculto por otimização).

Seção de cabeçalho
Definição da estrutura hierárquica com indicações de transformações
Lista de caixas envoltórias
Baseframe
Keyframes

Figura 3.23. Estrutura do arquivo de seqüência de animação

Para cada objeto que forma o corpo do personagem há uma malha. Cada malha é definida por uma lista de faces (triângulos), uma lista de pesos (para realização do *skinning*) e uma indicação para o arquivo de textura. Cada vértice contém informações sobre o mapeamento de textura e uma indicação para a lista de pesos (o peso inicial e a quantidade de pesos subseqüentes). Cada triângulo é descrito por índices indicando seus vértices. Um peso é definido por uma indicação de junta, o peso propriamente dito (sua contribuição para definição final da posição do vértice) e também sua posição (x, y e z).

Pela definição acima se pode perceber que no modelo md5, os vértices estão relacionados a várias juntas através dos pesos. Utilizando o esqueleto e o conjunto de malhas do arquivo principal é possível renderizar o modelo em uma posição *default*.

Cada arquivo de animação contém a hierarquia de juntas, caixas envoltórias de colisão, um *baseframe*, que é o ponto de partida da animação, e os *keyframes*.

A hierarquia de juntas que aparece em cada arquivo de animação não é apenas uma redundância. Ela contém novas informações (inexistentes na hierarquia do arquivo de modelo) e que é diferente para cada animação. Essa informações são um conjunto de 6 flags que podem, cada um, aparecer ou não em cada junta. Quando um flag não existe isso indica que o seu valor é zero.

Os flags são indicações de transformações de translações (x, y e z) e quatérnios de rotação (x, y e z) a serem lidos a cada *keyframe*.

Uma lista de caixas envoltórias de colisão é dada definindo os limites de uma caixa alinhada circundante (AABB) para o *baseframe* e cada *keyframe*. Embora o jogo Doom 3 utilize colisão por polígono, tais caixas são utilizadas para otimizar o processo de descarte de colisão [59]).

Finalmente o *baseframe* contém informações de posicionamento e transformações (translações e rotações) que permitem renderizar o modelo do personagem em uma posição inicial. Os demais *keyframes* são definidos como uma lista de valores de transformações (translações e rotações) que são lidas baseadas nos flags já citados e então são aplicados sobre o modelo na posição inicial para determinar novas posições, definindo a seqüência de animação.

Um exemplo de modelo no formato md5 pode ser visto na figura 3.24.

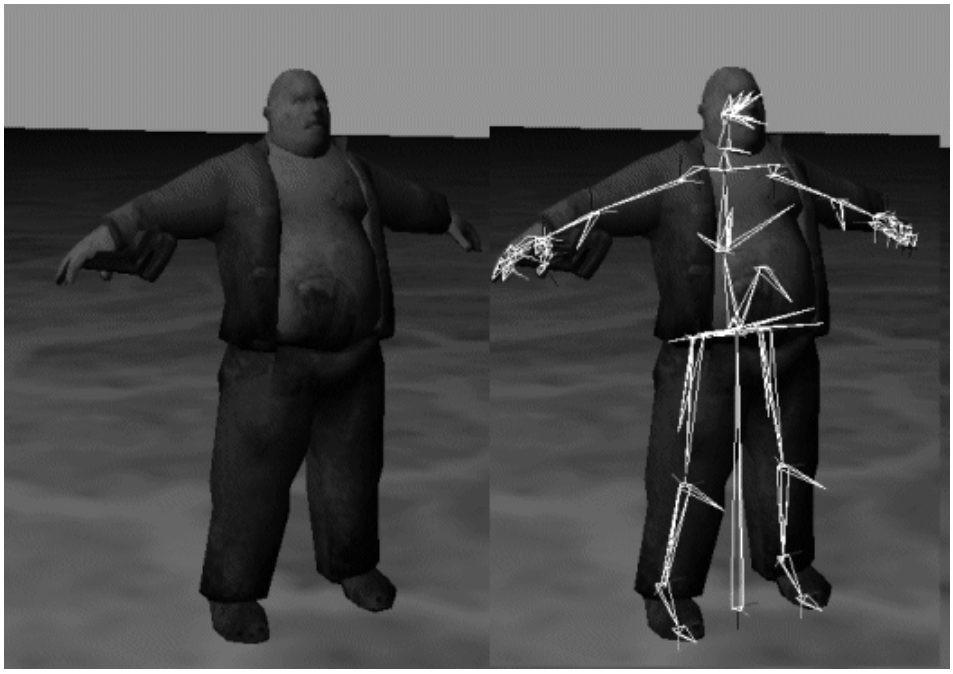

Figura 3.24. Modelo criado no formato md5

3.3 Conclusão do Capítulo

Sistemas de controle de animações de personagens podem ser divididos em duas camadas de abstração: alto nível e baixo nível. O alto nível é responsável pela inteligência do personagem enquanto que o baixo nível obtém e reproduz as seqüências de animação do personagem.

Dentre as técnicas de criação de animações, a técnica de *keyframing* ainda é mais popular entre os jogos 3D para representação de personagens.

A técnica de *keyframing* evoluiu para acompanhar a demanda por memória e aproveitando a capacidade processamento cada vez maior dos computadores. Assim novas técnicas foram incorporadas ao processo de criação e reprodução de seqüências de animações como definição de estruturas hierárquicas, utilização de quatérnios para representar rotações e *skinning*

4. O sistema de controle de animações de personagens do *framework* Guff

Neste capítulo inicia-se a apresentação do sistema de controle de animações de personagens do *framework* Guff. Primeiramente o *framework* Guff é apresentado para então em seguida ser mostrado como o módulo do sistema de controle de animações de personagens foi desenvolvido.

Algumas comparações foram feitas com bibliotecas e *game engines* para que se possa visualizar mais claramente como o sistema de controle de animações de personagens do *framework* Guff atende aos requisitos esperados.

4.1 O framework Guff

O framerwork Guff é formado basicamente por uma camada de aplicação e um *toolkit* [1]. A camada de aplicação determina a arquitetura das aplicações que serão criadas usando o *framework*. Essa camada é modelada como uma máquina de estados que permite criar aplicações decompostas em um conjunto de estados. Estados podem ser simples ou compostos (agrupamento de estados) e cada estado possui um estado pai que por *default* é um estado padrão chamado MasterState, como pode ser visto na figura 4.1. Estados possuem métodos e eventos definidos na interface abstractState. Os eventos podem ser de três tipos: eventos do sistema, eventos do ciclo principal de execução e eventos de estado.

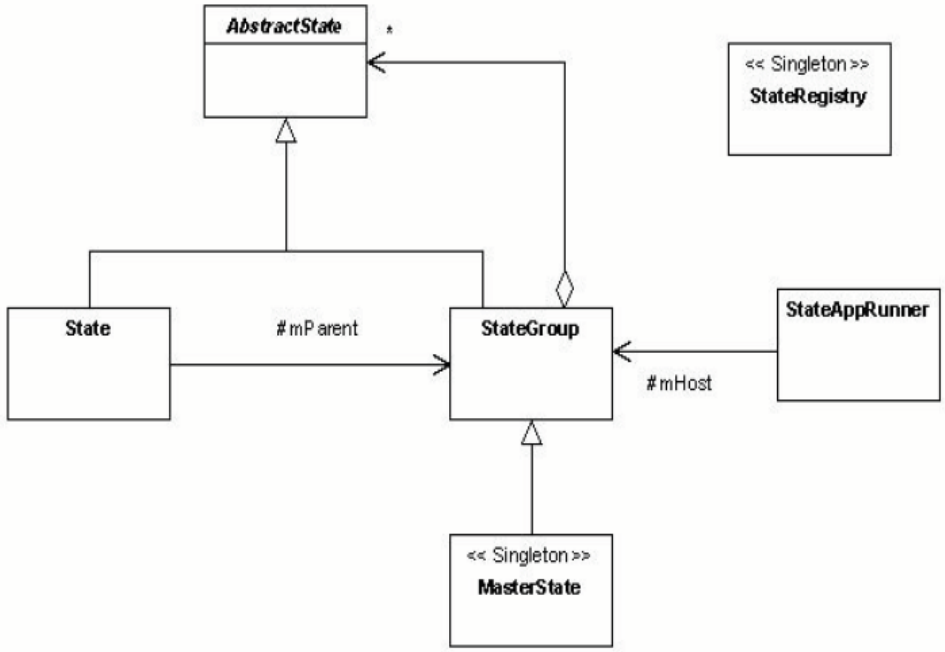

Figura 4.1. Estados do framework Guff [1]

O ciclo principal do jogo é baseado no modelo *Fixed Frequency Uncoupled Algorithm* [60], como pode ser visto na figura 4.2. Esse modelo é apropriado para jogos em computadores pessoais uma vez que permite a execução do jogo de forma determinística, ao mesmo tempo em que permite que o *loop* de renderização ocorra de forma mais veloz possível.

O *toolkit* disponibiliza uma série de funcionalidades que facilitam a criação de jogos. O *toolkit* está subdividido em *namespaces* que agrupam funcionalidades relacionadas permitindo o acesso e a utilização de cada funcionalidade de maneira mais eficiente. As classes do *toolkit* estão baseadas em bibliotecas de utilização livre, a saber: OpenGL, GLEW, boost, lib3ds, audiere, FTGL, Lua, DevIL e SDL [1]. Os *namespaces* podem ser visto na figura 4.3.

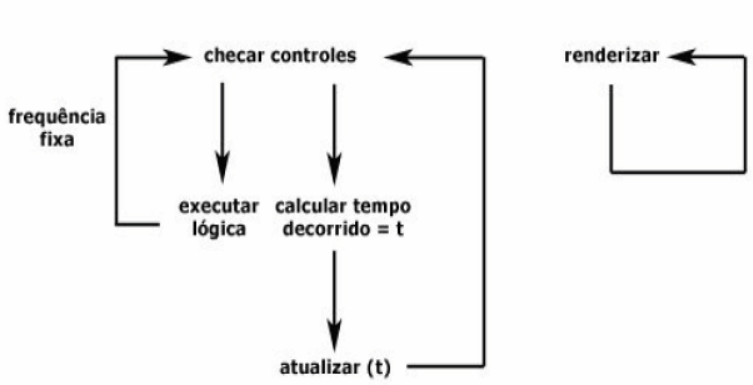

Figura 4.2. Ciclo principal de aplicação baseada no framework Guff [60]

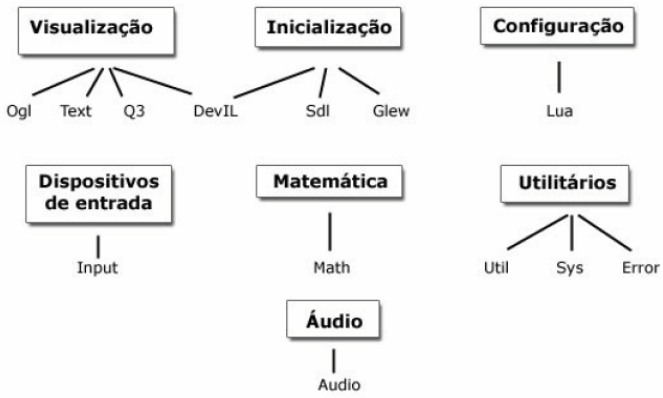

Figura 4.3. NameSpaces do toolkit do framework Guff [1]

4.2 O sistema de controle de animações de personagens

O sistema de controle de animações de personagens do *framework* Guff foi implementado como uma biblioteca pertencente ao *namespace* “Characters” como será mostrado no Apêndice A. Ele é divido em duas camadas: alto nível e baixo nível [61] [62].

O baixo nível utiliza a técnica de *keyframing* para a criação e reprodução de seqüências de animação. Os modelos e animações são baseados no formato md5 do jogo Doom3.

O formato md5 (que nada tem a ver com a função criptográfica de *hashing* MD5, utilizada para checagem da integridade de arquivos) foi escolhido por ser

considerado o estado da arte em termos de animação de personagens para jogos em 3D e por ser um formato cujas informações são descritas em código ASCII.

Pode-se questionar se o formato md5 é um formato livre. Para que um formato seja livre segundo a *Free Software Foundation* [63], é necessário que ele possa ser usado, copiado, estudado, modificado e redistribuído sem restrições. O fato de ser livre não significa ser gratuito.

Por outro lado, um "*software open-source*" é um *software* que inclui uma licença baseada na *Open Source Initiative* [63]. Ter acesso à descrição de um formato não faz dele um "código aberto" já que a definição da *Open Source Initiative* inclui itens como livre redistribuição, permissão de trabalhos derivados, não discriminação, distribuição e licença, entre outros.

Dessa forma, ter acesso à descrição de um formato ou código fonte é condição necessária, mas não suficiente para que ele seja livre e "*open-source*". Até onde foi possível apurar, as *game-engines* Quake e Quake II podem ser adquiridas através da GPL, que é uma licença de *software* livre [10]. A *game-engine* Quake III não está disponível com uma licença de *software* livre. Não foi possível obter informações sobre aquisição da *game-engine* Doom3.

O que se pode concluir a princípio é que o formato md5 é um formato proprietário (não aberto e não livre). Contudo, esse fato não é considerado um problema para a sua utilização, já que a ideologia do *framework* Guff é de o desenvolvimento de conhecimento na área de criação de jogos [1]. Sendo o md5 um formato cujo acesso à sua descrição é total e sendo esse formato o mais sofisticado em termos de modelagem e animação para personagens de jogos em 3D, ele foi considerado ideal. Cabe ainda lembrar que outros formatos também proprietários são utilizados no *framework* Guff como, por exemplo, o formato bsp (também conhecido como Q3) de cenários do jogo Quake III e o formato 3ds da empresa AutoDesk.

Além dos estágios de *design* e processamento, que um baixo nível baseado em *keyframing* costuma apresentar, há também um estágio intermediário de configuração que é responsável por disponibilizar ao projetista de jogo controles (através de uma ferramenta de configuração) complementando o estágio de *design* e suprindo o estágio de processamento de informações para a reprodução das seqüências de animação. O estágio intermediário pode ser visto na figura 4.4.

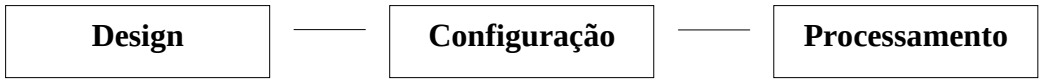

Figura 4.4. O estágio intermediário de configuração no baixo nível do sistema de personagens baseado em *keyframing*

A primeira motivação para a utilização de um estágio intermediário de configuração é permitir ao projetista de jogo realizar um “ajuste fino” através da escolha do número de *intermediate frames*. Como será visto mais adiante, a configuração de um número de *intermediate frames* variável (para cada par de *keyframes*) dentro de uma seqüência de animação, permite controlar a uniformidade entre *keyframes* durante a seqüência de animação, controle de efeitos de não-uniformidade desejados pelo projetista de jogo, aqui chamado de evolução da seqüência de animação, e finalmente o controle do fator de *timing*.

Uma segunda motivação para se ter um estágio intermediário de configuração é disponibilizar ao projetista de jogo uma forma de “conectar” o baixo nível com o alto nível no sistema de controle de animações de personagens. Como os comportamentos do personagem, gerados no alto nível, serão representados com a reprodução de seqüências de animação pelo baixo nível, a correta associação entre comportamentos e seqüências de animação pode ser realizada pelo projetista de jogo no estágio intermediário de configuração. Geralmente essa tarefa é realizada utilizando uma ferramenta de configuração que gera informações de script [19], sendo utilizadas no estágio

de processamento para reprodução das seqüências de animação corretamente.

Finalmente, o estágio intermediário de configuração possibilita também ao projetista de jogo a configuração de atributos que serão utilizados no processo de transição entre duas seqüências de animação. Esses atributos permitem que a transição, como será visto mais à frente, ocorra de forma mais suave e visualmente mais agradável.

A utilização do estágio intermediário de configuração permite uma diferenciação mais bem definida entre os papéis do modelador e do projetista de jogo. O modelador cria modelos e as seqüências de animação. Enquanto que o projetista de jogo utiliza esses modelos e seqüências para criar personagens com comportamentos definidos durante o jogo.

O estágio intermediário de configuração, com suas funções, pode ser visto na figura 4.5.

Figura 4.5. O estágio intermediário de configuração e suas funções

Para que essas funções possam ser realizadas de forma simples e prática, uma ferramenta de configuração foi criada. Esta ferramenta de configuração é apresentada no Apêndice A.

O alto nível foi desenvolvido mais como uma forma de testar a capacidade do baixo nível em prover seqüências de animações, durante o estágio de processamento, corretamente, do que propriamente para suprir os personagens com um sistema de inteligência. Por isso, uma máquina de estados finitos bastante simplificada foi definida de forma que o personagem pudesse apresentar os comportamentos mais comuns (como ataque, defesa, normalidade e morte) e então estes pudessem ser representados por seqüências de animações convenientemente reproduzidas pelo baixo nível.

4.3 Comparações com outros frameworks e game engines

Segue-se uma comparação entre o sistema de controle de animações de personagens do *framework* Guff e de *bibliotecas* e *game engines* bastante conhecidas e utilizadas. As *bibliotecas* e *game engines* analisados aqui são: OGRE *software*, CAL3D e a *game engine* do jogo Doom3.

OGRE [64] *software* é uma *engine* de renderização 3D em tempo real. Seus códigos são *open source* e ela pode ser utilizada na criação de jogos, visualização arquitetônica e simulações.

O sistema de animação da OGRE permite a obtenção de modelos exportados dos principais *softwares* de modelagem (3D Studio Max, Maya, Blender). Permite a animação e modelos baseados em *skeletal animation* com *skinning*. Disponibiliza também a mesclagem em transições de animações.

Até onde se pôde observar, não contém um estágio de configuração para animações e não permite a configuração de uma camada de alto nível para os personagens. O desenvolvedor de jogo deve implementá-las via código.

Cal3D [65] é biblioteca escrita com uma API gráfica própria para animação de personagens em 3D. Cal3D pode ser utilizada para a criação de jogos e aplicativos de visualização em 3 dimensões.

Cal3D permite a obtenção de arquivos exportados do 3D Studio Max, Blender e MilkShape3D. Além disso, é baseada em *skeletal animation* e permite a mesclagem de animações de personagens. Também permite a criação de movimentos dinamicamente durante a execução da aplicação.

Cal3D não disponibiliza um estágio intermediário de configuração e não possui uma camada de alto nível do sistema de controle de animações de personagens. O desenvolvedor de jogo deve implementá-los, como acontece com o sistema OGRE.

A *engine* do jogo Doom3 serve para a criação de *mods*. Um *mod* é uma modificação de um jogo [10]. Um exemplo bastante famoso de *mod* é o jogo Counter Strike criado utilizando-se a *engine* do jogo Half Life.

Como já foi mostrado antes o jogo Doom3 utiliza o formato md5 para animações de personagens. Dessa forma, os personagens são baseados em *skeletal animation* utilizando *skinning*. Os modelos podem ser exportados dos *softwares* de modelagem: 3D Studio Max, Blender e Maya [26].

A *engine* do jogo Doom3 possui *softwares* de configuração que permitem a associação de seqüências de animação à comportamentos da camada de alto nível do sistema de controle de animações de personagens. Essas configurações são armazenadas em *scripts* que provêm tais informações de configuração ao estágio de processamento durante a execução do jogo [10].

Não foi possível descobrir com clareza se a *engine* Doom3 utiliza mesclagem nas transições entre as seqüências de animação. As informações obtidas em [10], [26] e da equipe de desenvolvimento da *engine* Doom3 (contactada via email) são inconclusivas sobre este aspecto.

Na tabela 4.1 as características de cada sistema citado e também do *framework* Guff são sumarizados e comparados.

Biblioteca, *framework* ou *game engine*	*Skeletal animation* e *skinning*	Estágio de configuração	Tratamento de transições	*Camada de alto nível*
OGRE	Sim	?	Sim	*Não*
Cal3D	Sim	Não	Sim	*Não*
Doom3 *engine*	Sim	Sim	?	*Sim*
Guff	*Sim*	*Sim*	*Sim*	*Sim*

Tabela 4.1. Comparação entre os sistemas de controle de animações de personagens de bibliotecas, *game engines* e o *framework* Guff

4.4 Conclusão do Capítulo

O *framework* Guff é baseado em uma camada de aplicação e um *toolkit*. A camada de aplicação é baseada em uma máquina de estados que possibilita a criação de aplicações baseadas em estados. O *toolkit* é um conjunto de ferramentas que disponibiliza classes úteis ao desenvolvedor de jogo.

O sistema de controle de animações de personagens do *framework* Guff apresenta em seu baixo nível três estágios: *design*, configuração e processamento. O estágio de *design* realiza a obtenção de animações baseadas no jogo Doom3. O estágio intermediário de configuração possibilita ao projetista de jogo realizar ajustes nas quantidades de *intermediate frames*, associar seqüências de animações para responder aos comportamentos resultantes do alto nível e configurar transições entre seqüências de animação. No estágio de processamento as seqüências de animação são reproduzidas.

O alto nível do sistema de controle de animações do *framework* Guff foi desenvolvido de forma bastante simplificada para, em primeira instância, possibilitar o desenvolvimento de testes do baixo nível.

O sistema de controle de animações de personagens do *framework* Guff foi comparado com bibliotecas e *game engines*, mostrando que ele apresenta as principais características esperadas.

5. Configuração da seqüência de animação pelo número de *intermediate frames*

Ao criar uma seqüência de animação em uma ferramenta de modelagem 3D, no estágio de *design*, o modelador deve salvá-la e então convertê-la para o formato md5 [26]. Todos os *frames* (*keyframes* ou *intermediate frames*) configurados durante a modelagem, são convertidos para *keyframes* quando a animação é salva e convertida para o formato md5. O problema existente nesse processo é que os *intermediate frames* não são tão importantes quanto os *keyframes* e não seria necessário gravá-los sob a forma de *keyframes*. Por isso, essa abordagem implica um desperdício de memória para armazenamento de transformações (translações e rotações), como é feito com *keyframes*. Para jogos em tempo real a economia de utilização de memória é uma questão importante uma vez que muitos aspectos do jogo (cenários, personagens, trilha sonora, entre outros) devem ser armazenados por cada nível do jogo.

Uma solução utilizada por jogos 3D é apenas criar *keyframes* durante o estágio de *design* e no estágio de processamento utilizar um número inteiro global para definir quantos *intermediate frames* devem ser criados por interpolação. Isso significa que todos os intervalos entre *keyframes* (também chamados segmentos) de uma mesma seqüência de animação serão preenchidos com o mesmo número de *intermediate frames.* O sistema de controle de animações de personagens do jogo Doom3 é baseado nesse conceito.

Essa solução apresenta duas deficiências: primeiramente não há como evitar a geração de *intermediate frames* (que ao final serão armazenadas como *keyframes*) ao criar uma animação em uma ferramenta de modelagem 3D, quando o modelador utiliza algum tipo de controle de evolução, o que é muito comum [66]. Além disso, uma quantidade única de *intermediate frames* para cada seqüência de animação é uma configuração bastante restritiva, pois como

será visto a seguir, não permite que novos controles sejam associados à seqüência de animação.

O sistema de controle de animações de personagens do *framework* Guff apresenta uma nova abordagem. O modelador pode apenas criar *keyframes* (sem utilizar algum tipo de controle de evolução). Depois, no estágio intermediário de configuração, o projetista de jogo pode configurar a quantidade de *intermediate frames* variável para cada intervalo entre dois *keyframes* dentro da seqüência de animação. Os valores de *intermediate frames* são gravados em um arquivo de configuração como números inteiros. Isso significa uma grande economia de memória no estágio de processamento, quando as seqüências de animação ficam armazenadas em memória para reprodução. A figura 5.1 mostra as três abordagens de tratamento dos *intermediate frames*.

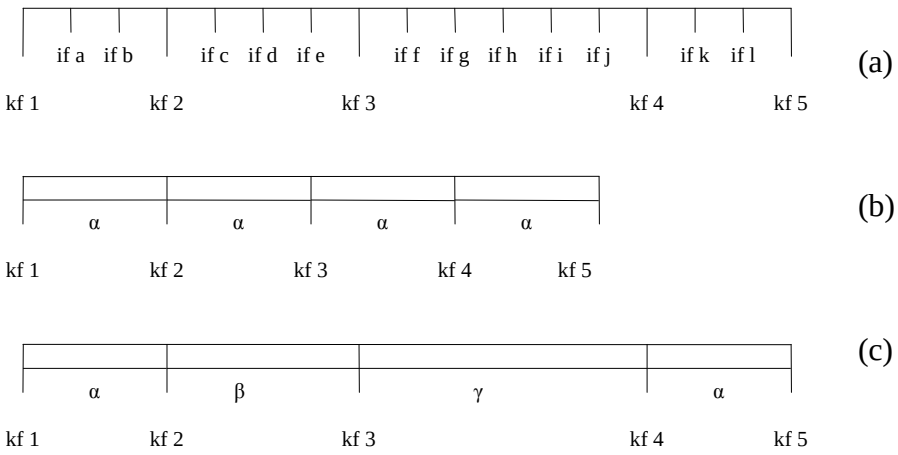

Figura 5.1. Comparação entre três abordagens de tratamento de *intermediate frames*, em (a) eles são gravados como *keyframes*, em (b) são representados por um valor constante (α) para cada par de *keyframes*, e em (c) são representados por valores variáveis (α, β, γ) para cada par de *keyframes*

Pode-se imaginar, por exemplo, uma seqüência de animação com 3 *keyframes*. Para realizar uma animação um modelador pode configurar dentro da ferramenta de modelagem 3D que entre o primeiro e o segundo *keyframe*

haja dois *intermediate frames* e que entre o segundo e o terceiro *keyframe* haja três *intermediate frames*. Ao salvar a seqüência de animação o resultado serão oito *keyframes* e não somente três. Uma segunda alternativa é salvar apenas três *keyframes* e depois no estágio de processamento inserir um número constante de *intermediate frames*, por exemplo, dois. O problema dessa abordagem é que se perde a possibilidade de modular efeitos de evolução à seqüência de animação. A alternativa apresentada pelo *framework* Guff é através do estágio intermediário de configuração inserir dois *intermediate frames* entre o primeiro e o segundo *keyframes* e três *intermediate frames* entre o segundo e o terceiro *keyframes*. Essa solução tem o mesmo resultado da primeira sem, contudo armazenar os *intermediate frames* como *keyframes*.

Em termos de desempenho, o estágio de processamento não apresenta maior *overhead* uma vez que um número variável ou constante de *intermediate frames* não significa nenhuma diferença de complexidade sobre a função de interpolação e no processamento das seqüências de animação como um todo.

Além disso, é possível configurar um número diferente de *intermediate frames* para cada par de *keyframes* dentro de uma seqüência de animação. Essa variação de quantidades de *intermediate frames* permite criar ajustes pós-*design* automáticos, aqui chamados de controles da seqüência de animação. O estágio de configuração intermediário permite a configuração, atualmente, de três controles sobre uma seqüência de animação. Esses controles são agregados para formar um número de *intermediate frames* para pares de *keyframes*.

Primeiramente pode ser útil para o projetista de jogo, minimizar alguma não-uniformidade não desejada, causada pela distância variável mal dimensionada de *keyframes*, na seqüência de animação, criada no estágio de *design*. O controle de uniformidade é capaz de compensar essas diferenças através de uma variação no número de *intermediate frames*.

Uma questão que se pode colocar é se não seria mais fácil retornar ao estágio de *design* para consertar esta não-uniformidade. Primeiramente não se pode garantir que os modelos e seqüências de animação tenham sido criados por um modelador que faça parte do projeto, já que muitas vezes, se obtém modelos e seqüências remotamente (pela Internet, por exemplo). Além disso, a configuração pelo estágio intermediário é uma forma mais econômica que pode figurar como uma homologação ou prototipação rápida para então se decidir se vale à pena voltar ao estágio de *design*. Finalmente, o estágio intermediário modifica o resultado final da seqüência através de *scripts*, como pode ser visto no apêndice A, sem modificar a seqüência original o que significa que uma mesma seqüência de animação pode ter vários ajustes diferenciados de acordo com o interesse do projetista de jogo.

O segundo controle que pode ser utilizado pelo projetista de jogo é o controle de evolução. O termo evolução de seqüência de animação, neste texto, é utilizado como sinônimo de qualquer não-uniformidade desejada pelo projetista de jogo. O efeito de evolução mais conhecido é o *slow in* e *slow out* [25].

O terceiro controle é utilizado pelo projetista de jogo para configurar um fator de *timing* a ser aplicado a todos os números de *intermediate frames*. Esse fator pode ser modificado durante o estágio de processamento de acordo com o desempenho da aplicação fazendo a animação ser reproduzida mais rapidamente quando necessário.

Além dos controles automáticos, o estágio intermediário de configuração permite também que o projetista de jogo ajuste manualmente os números de *intermediate frames* se ele achar necessário, como proposto por [67]. Segue-se uma descrição mais aprofundada de cada um dos controles.

5.1 Controle baseado na uniformidade entre keyframes

Uma questão relevante na manipulação do número de *intermediate frames* é a uniformidade entre *frames*. A idéia é que o estágio intermediário seja capaz de equiparar *keyframes* muito distantes através de um número maior ou menor de *intermediate frames* entre eles.

Dessa forma, se pode equilibrar a distância entre *keyframes* aumentando ou diminuindo o número de *intermediate frames* e minimizar os efeitos bruscos causados por modificações muito acentuadas de um *keyframe* para outro. Essas mudanças bruscas podem ou não ser pretendidas pelo modelador no estágio de *design*. Caso elas sejam indesejáveis, é possível atenuá-las através desse controle. Dessa forma, o controle de uniformidade pode ser entendido como filtro que retira os ruídos nas seqüências de animação.

A maior dificuldade para a realização desse controle é definir o que é a distância entre dois *keyframes*. Para efeitos práticos e como uma primeira abordagem, no projeto de desenvolvimento do sistema de controle de animações de personagens do *framework* Guff, foi utilizada uma definição bastante simples de distância entre os *keyrames,* baseada na posição final entre os vértices que compõem o corpo (a malha) do personagem.

Distância, neste caso, pode ser entendida como diferença. Assim, quanto maior a diferença entre dois *keyframes* contíguos, ou seja, a diferença das posições dos vértices nos dois *keyframes*, maior a distância entre eles.

Uma determinada função real, e um conjunto domínio formam um espaço métrico se quatro condições são satisfeitas. São elas [68]:

Positividade: se a distância de um ponto a outro dentro do conjunto domínio é sempre um valor positivo e de um ponto para ele mesmo é zero.

Positividade restrita: se a distância entre dois pontos do conjunto domínio for igual a zero então esses pontos são na realidade o mesmo ponto.

Simetria: se a distância de um ponto x a um ponto y é igual à distância do ponto y ao ponto x. Sendo ambos os pontos pertencentes ao conjunto domínio.

Desigualdade do triângulo: se a distância de um ponto x a um ponto y é menor ou igual à distância do ponto x a um ponto z e do ponto z ao ponto y. Sendo os pontos x, y e z pertencentes ao conjunto domínio.

A função de distância proposta é utilizada para a determinação da diferença entre os *keyframes* é baseada no somatório da distância euclidiana [68], que obedece às quatro condições citadas.

Em um determinado *keyframe* pode-se dizer que a posição de um vértice v é definida pelo trio de valores v_{x1}, v_{y1}, $v_{z1,}$ valores estes associados aos eixos canônicos x, y e z respectivamente. No *keyframe* seguinte a posição do vértice v é dada pelo trio v_{x2}, v_{y2}, $v_{z2,}$ nas mesmas condições citadas com relação aos eixos canônicos. É interessante notar que essas são posições finais do vértice v para cada *keyframe*, já considerando a influência da junta (ou juntas) a qual ele está associado.

A distância euclidiana entre os valores em cada um dos eixos é dada por:

$$d_{v12} = ((v_{x1} - v_{x2})^2 + (v_{y1} - v_{y2})^2 + (v_{z1} - v_{z2})^2)^{1/2}$$

Onde d_{v12} é a distância entre o vértice v no *keyframe* 1 e o mesmo vértice v no *keyframe* 2. Ao somar as distâncias de cada vértice entre o *keyframe* 1 e o *keyframe* 2 obtemos a distância entre os dois *keyframes*. A figura 5.2 mostra a distância entre dois *keyframes*: os pontos azuis são os vértices do personagem, onde d_{12} é a distância total entre os dois *keyframes*.

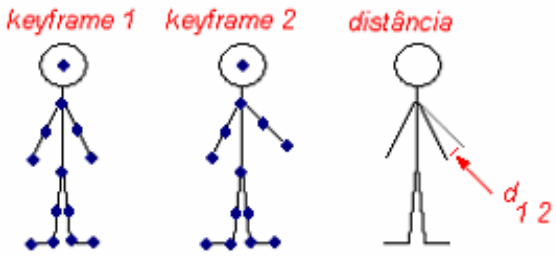

Figura 5.2. Distância entre dois *keyframes*

A escolha da utilização do somatório das distâncias de todos os vértices foi feita para evitar distorções quando poucos vértices estiverem muito distantes, mas os *keyframes* não forem muito diferentes entre si.

A figura 5.3 mostra que um vértice não é suficiente para definir a distância entre dois *keyframes*. Pode-se observar que embora o vértice da mão esquerda tenha se movido mais no *keyframe* 2, o *keyframe* 2' é mais distante do *keyframe* 1 pois outros vértices contribuíram para o cálculo da distância.

Depois que as distâncias são calculadas para cada segmento (entre os pares de *keyframes* contíguos) pode-se associar um valor normalizado aos maiores e menores valores para que todos os valores estejam dentro de uma faixa pré-definida que fará uma correspondência de maiores valores de distância com maiores valores de número de *intermediate frames*.

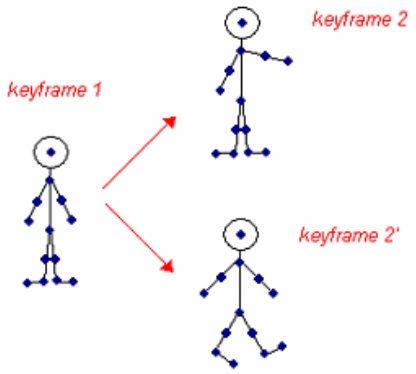

Figura 5.3. Contribuição de todos os vértices para o cálculo da distância

5.2 Controle baseado em evolução

Na criação de animações de personagens em 3D, existem dois princípios [25]: *timing* e evolução, que podem ser dissociados do estágio de *design* e que podem ser tratados no estágio intermediário de configuração para posterior utilização no estágio de processamento.

Eles estão diretamente relacionados com o número de *intermediate frames* em cada segmento de uma seqüência de animação e por isso, podem ser implementados através da definição correta do número de *intermediate frames* em cada segmento (intervalo entre um par de *keyframes* contíguos). Assim é possível que o modelador crie uma animação e depois o projetista de jogo faça um ajuste fino através da manipulação dos números de *intermediate frames*, como é feito em pacotes de animação dotados de ferramentas de edição de *time-editing* [69][70].

Evolução é a característica de não uniformidade de um movimento causada pela: aceleração ou desaceleração do corpo; influência de alguma força externa; ou algum tipo de comportamento que o projetista de jogo deseja enfatizar. O exemplo mais comum de evolução é o chamado *slow in* e *slow out* (acelerar e desacelerar) que é a característica de um movimento de iniciar lento, acelerar em seu ápice e depois desacelerar novamente [25][69][71].

Um exemplo pode ser visto na figura 5.4. Pode-se observar que a animação foi concebida com 5 *keyframes* e depois com a inserção de *intermediate frames* o efeito de *slow in* e *slow out* foi conseguido, considerando tempo constante para a representação de cada *frame*. A percepção final é de que o começo e o fim da seqüência de animação demoram mais do que os movimentos intermediários, isto é, parece que leva mais tempo ir do *keyframe* 1 (kf 1) para o *keyframe* 2 (kf 2) do que ir deste para o *keyframe* 3 (kf 3). O mesmo ocorre do meio para o fim da seqüência de animação.

É importante observar que a idéia da evolução não é, como pode parecer a princípio, contrária a idéia da uniformidade apresentada na seção anterior. A uniformidade corrige erros de continuidade não desejados ao passo que a evolução insere diferenças de continuidade desejáveis para a representação do movimento. As duas idéias são na verdade complementares.

Outros exemplos de evolução poderiam ser: um comportamento atenuado mostrando cansaço, ou algum tipo de comportamento errático como um tique nervoso ou uma aceleração mostrando pressa, medo e fuga.

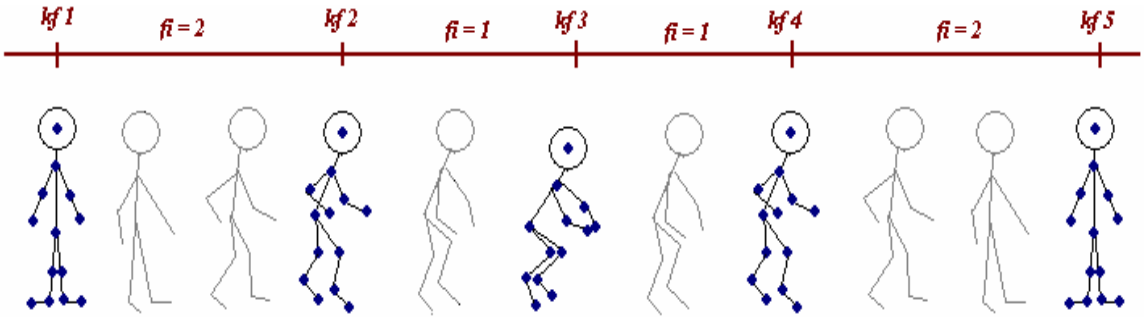

Figura 5.4. *Slow in* e *Slow out*

Segundo [27], que chama o *slow in* e *slow out* de *easy in* e *ease out*, existem funções de interpolação apropriadas para simular esse tipo de variação no número de *intermediate frames*. Algumas curvas podem ser convenientemente modeladas para proporcionar o efeito desejado como curvas senoidais e parabólicas [27]. Também curvas cúbicas (*splines*) definidas como em [72] podem ser utilizadas para a criação do efeito de *slow in* e *slow out* [25].

Com as curvas senoidais, parabólicas e *splines*, podem ser encontrados valores nos quais se podem basear os números de *intermediate frames* conforme a função definida. Dessa forma, o controle de evolução pode ser entendido como um modulador que faz os valores de *intermediate frames* dos segmentos da seqüência de animação obedeceram ao comportamento da função escolhida.

No estágio intermediário de configuração, o projetista de jogo pode escolher uma dessas três funções para criar o efeito de *slow in* e *slow out*. A

abordagem para a configuração utilizada no projeto foi permitir ao usuário configurar de forma analítica (através dos coeficientes dos polinômios) as funções senoidais, parabólicas e *splines*.

Genericamente a função senoidal α + βsen(γπ x) pode ter seus parâmetros configurados como α = 1, β = -1 e γ = 1 resultando na função: F(x) = 1 - seno (π x), onde x varia de [0 , 1], e cujo comportamento é mostrado na figura 5.5.

Funções parabólicas podem ser representadas como: $ax^2 + bx + c$. Como exemplo, pode ser fazer a = 4, b = -4 e c = 1, resultando na função: $F(x) = 4x^2 – 4x + 1$, onde x varia de [0 , 1], e cujo comportamento é mostrado na figura 5.6, para uma seqüência de animação de 35 *keyframes*.

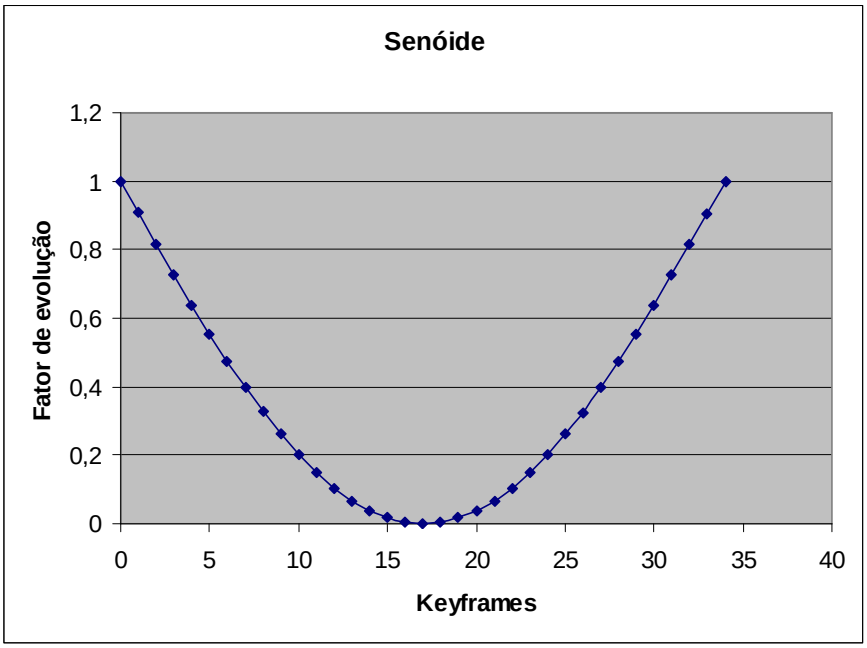

Figura 5.5. Curva senoidal

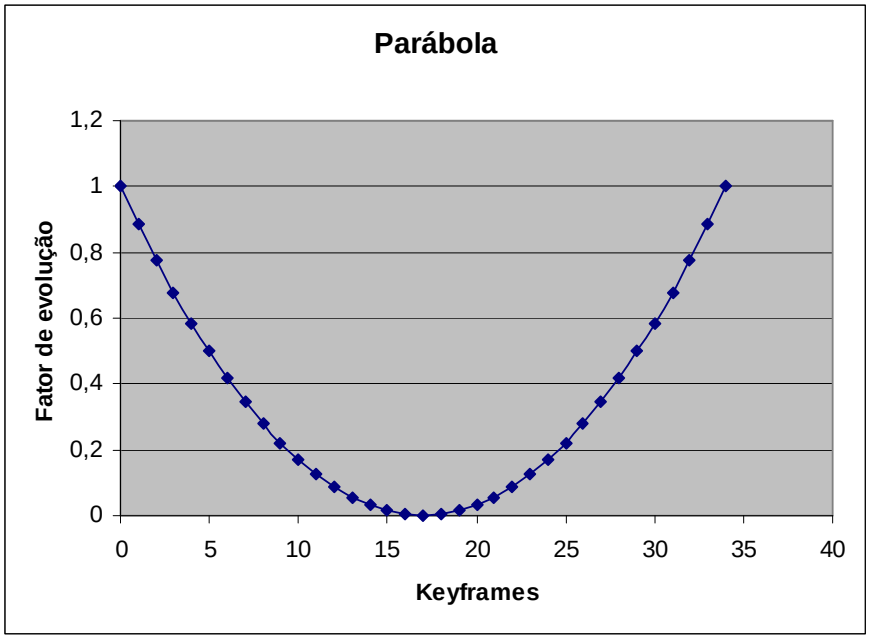

Figura 5.6. Curva parabólica

A função B-*spline* uniforme cúbica [14] [25] [72], cuja forma analítica pode ser vista no apêndice B, com os pontos de controle arbitrados com os valores 0, 0.5, 0.5 e 0, apresenta o comportamento mostrado na figura 5.7.

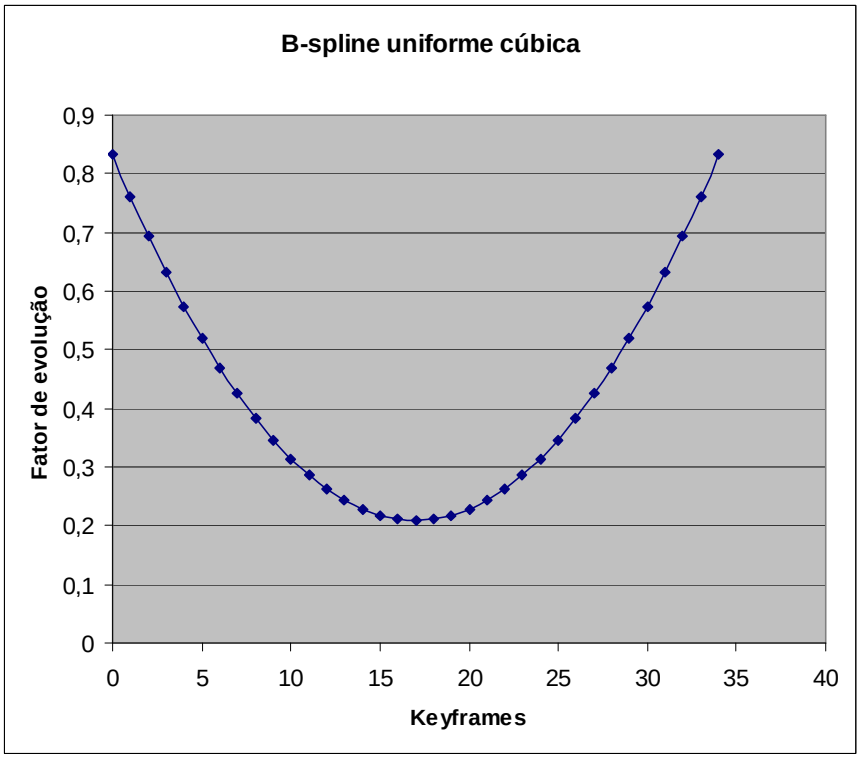

Figura 5.7. Curva B-*spline* uniforme cúbica

Na implementação do estágio intermediário de configuração, os segmentos de uma determinada seqüência de animação foram normalizados para serem associados a valores dentro do intervalo [0 , 1]. Assim, se a animação possui 35 segmentos o primeiro deles está associado ao valor 0 e o trigésimo quinto ao valor 1.Dessa forma um determinado segmento está associado ao valor : (a - min) / (max - min), onde a é o valor n-ésimo do segmento e min é o valor mínimo do intervalo (zero) e max é o valor máximo do intervalo.

5.3 Controle de timing adaptativo baseado na taxa de frames por segundo da aplicação

Timing é a característica de velocidade de um movimento apresentada em uma seqüência de animação [25][69][70][73]. Algumas das informações mais importantes para o expectador de uma animação são dadas através da escolha correta do *timing*. Ele pode influenciar a visualização do movimento uma vez que o tempo deve ser suficiente para que a seqüência seja percebida pelo expectador.

Também pode influenciar na noção de peso do objeto uma vez que objetos mais pesados têm maior inércia e costumam executar movimentos mais lentos do que objetos mais leves. Além disso, o *timing* também é capaz de proporcionar a noção de tamanho e escala de um objeto ou personagem.

O estado emocional de um personagem também pode ser bem definido com o ajuste da velocidade de suas ações. Um exemplo da influência do *timing* no estado emocional de um personagem foi dado por [25] e é transcrito a seguir:

Apenas duas poses de uma cabeça, a primeira pose mostrando a cabeça inclinada na direção do ombro direito e a segunda pose com a cabeça sobre o ombro esquerdo e seu queixo levemente levantado, podem ser utilizadas para comunicar múltiplas idéias, dependendo inteiramente do *timing* utilizado. Cada

intermediate frame adicionado entre estes dois “extremos” dá um novo significado ao movimento, como mostrada na tabela 5.1.

Número de *intermediate frames*	**Significado do movimento**
0	O personagem foi atingido com tremenda força. Sua cabeça foi quase que arrancada.
1	O personagem foi atingido por um tijolo, um rolo de macarrão ou por uma frigideira.
2	O personagem tem um tique nervoso, um espasmo muscular ou uma contração involuntária.
3	O personagem está se esquivando de um tijolo, um rolo de macarrão ou de uma frigideira.
4	O personagem está dando uma ordem seca: “Vamos, Mexa-se”.
5	O personagem é mais amigável: “Hei! Venha aqui!”, “Venha para cá!”.
6	O personagem viu uma menina bonita ou o carro esporte que ele sempre quis.
7	O personagem tenta ver melhor alguma coisa.
8	O personagem está procurando algo (como a manteiga de amendoim sobre uma prateleira na cozinha).
9	O personagem move a cabeça pensativamente.
10	O personagem estica um músculo dolorido.

Tabela 5.1. Significados de movimento informados através do *timing* da seqüência de animação

Como pode ser observado pelo exemplo acima o *timing* é um controle que deve, ao contrário dos demais, ser definido globalmente (o mesmo para todos os segmentos) da seqüência de animação. Por isso o fator de *timing* define um mesmo número de *intermediate frames*, para cada segmento da seqüência de animação.

Os controles de uniformidade, evolução e *timing* são complementares e seus valores, para cada segmento da seqüência de animação são agregados, no estágio de processamento para determinar o real número de *intermediate frames* de cada segmento. Assim:

número de *intermediate frames* = fator de evolução · fator de uniformidade · fator de *timing* (1)

fator de evolução $\in$ [0.001 , 1]

fator de uniformidade $\in$ [1 , 10]

fator de *timing* $\in$ [1, 100]

Como o desempenho computacional (aqui considerado em função do número de *frames* por segundo) de um jogo 3D pode variar muito, se a quantidade de *frames* de uma seqüência de animação for fixa, o resultado será que o tempo de duração da animação variará com a taxa de *frames* por segundo. Isso não é o desejável (a taxa de *frames* por segundo para jogos de computador não deve ser inferior a 30 fps [6]).

Pode-se criar um controle sobre a configuração do projetista de jogo para que seja possível ao estágio de processamento interferir aumentando ou diminuindo a duração da seqüência de animação. Sendo o fator de *timing* o único que é constante, então ele se mostra ideal para que esse ajuste seja feito em função da taxa de *frames* por segundo, utilizada como *feedback*.

Assim o número de *intermediate frames* em cada segmento passa a ser variável com o número de *frames* por segundo, em seu fator de *timing*. Os fatores responsáveis por controle de evolução e controle de uniformidade, definidos, se manterão inalterados, uma vez que somente após esse ajuste do fator de *timing* se procede à mesclagem de todos os fatores de controle conforme mostrado na equação (1).

A figura 5.8 mostra como a taxa de *frames* por segundo serve como um controle dinâmico para velocidade de reprodução da seqüência de animação.

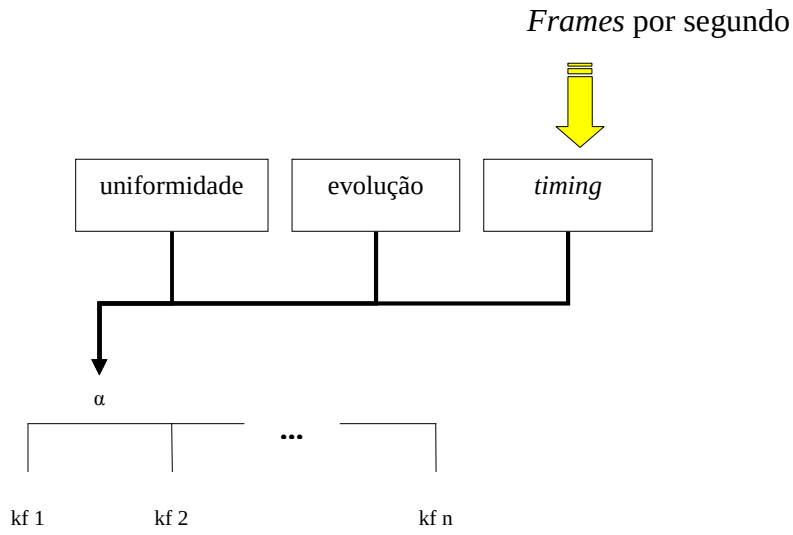

Figura 5.8. A taxa de *frames* por segundo interfere no fator de *timing* variando a velocidade.

5.4 Conclusão do Capítulo

O estágio intermediário de configuração permite que sejam configuradas as quantidades de *intermediate frames* para cada segmento das seqüências de animação de um personagem. Essa configuração pode ser feita manualmente pelo projetista de jogo ou baseada em métodos de controle: uniformidade, evolução e *timing*.

O controle de uniformidade permite atenuar efeitos de não-uniformidades indesejáveis entre *keyframes* produzidas no estágio de *design*, enquanto o controle de evolução permite criar o efeito de não-uniformidade conhecido como *slow in* e *slow out*. Finalmente o controle de *timing* permite não só a configuração de um fator global para a velocidade, como também permite que esse fator seja manipulado dinamicamente durante o estágio de processamento com um ajuste baseado na taxa de *frames* por segundo.

6. Módulo de transição de seqüências de animação

Durante a vida de um personagem em um jogo ele irá mudar seu estado de acordo com os eventos que acontecem no meio ambiente em que está inserido e também de acordo com a sua inteligência. Mudanças de estado significam na maioria das vezes mudanças de seqüências de animações, por isso, o tratamento dado às transições é um ponto importante para qualquer sistema de controle de animações de personagens.

No estudo que se segue será utilizada a seguinte terminologia: seqüência atual é seqüência de animação que está sendo reproduzida no momento corrente, seqüência seguinte é a seqüência de animação escolhida para se suceder à seqüência atual devido a uma mudança de estado do personagem. Chama-se fase de mesclagem o intervalo de tempo em que os *frames* finais reproduzidos são na verdade uma combinação entre os *frames* da seqüência atual e os *frames* da seqüência seguinte. Chamam-se *frames* alinhados os *frames* de seqüências diferentes que são interpolados. A figura 6.1 mostra esses conceitos graficamente. Neste diagrama, o fato dos *frames* serem representados na mesma cor significa que eles fazem parte da mesma seqüência e não que eles são idênticos.

Na implementação do sistema de controle de animações de personagens do *framework* Guff [62], duas questões relacionadas a transições foram analisadas: o posicionamento e a mesclagem. A primeira questão existe porque durante o estágio de *design*, o modelador irá posicionar o modelo, no início de uma seqüência, provavelmente em um lugar completamente diferente de onde o mesmo modelo está posicionado ao término de uma outra. Isso acontece, por exemplo, com os modelos obtidos do jogo Doom3.

As coordenadas de cada vértice do modelo, no primeiro *keyframe* da seqüência seguinte, têm que ser remapeadas para o ponto onde o modelo se encontra, na seqüência que está sendo reproduzida atualmente.

A questão da mesclagem se refere à suavidade com que a seqüência atual será substituída pela seqüência seguinte. Quando não existe mesclagem, a transição entre duas seqüências de animação com *keyframes* limites (final da atual e início da seguinte) muito diferentes, será muito brusca.

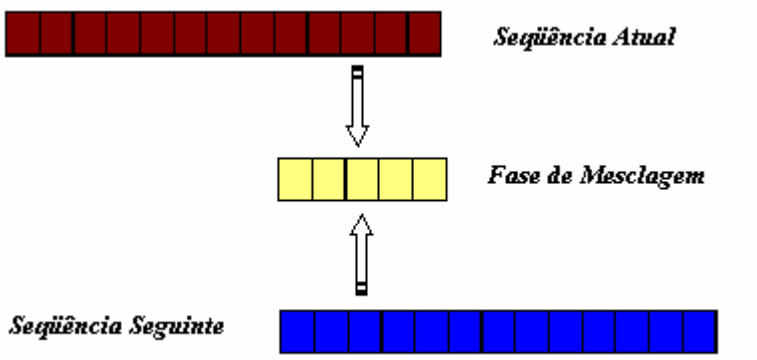

Figura 6.1. Conceitos de seqüência atual, seqüência seguinte e fase de mesclagem

6.1 Tipos de transições

No curso deste trabalho foram identificados dois tipos de mudanças de estado que um personagem pode apresentar: transições determinísticas e transições não-determinísticas. Estas mudanças referem-se ao momento em que uma seqüência de animação será substituída por outra seqüência.

Transições determinísticas são aquelas em que se sabe exatamente em que *frame* (sendo um *keyframe* ou um *intermediate frame*) a seqüência de animação atual começará a ser substituída pela próxima. Em geral, esse tipo de transição ocorre quando o personagem muda o seu estado sem ter sido influenciado pelo meio ambiente em que está inserido. Por exemplo, se o personagem está correndo (a seqüência de animação atual é “correndo”) e gradativamente ele pára e fica na posição em repouso (a seqüência de animação seguinte é “repouso”). Pode-se definir previamente qual o exato

momento (*frame* da seqüência atual) começa a acontecer uma transição para a seqüência seguinte.

Em transições não-determinísticas, ao contrário, não é possível definir em qual *frame* começará a transição. Elas ocorrem quando um personagem, de repente, muda seu estado devido a algum acontecimento no meio ambiente. Por exemplo, quando um personagem recebe um disparo e então, deixa seu comportamento de repouso (seqüência de animação atual é “repouso”) e começa o comportamento de fuga (seqüência de animação seguinte é “correndo”).

Deixar toda responsabilidade das transições sobre estágio de *design* não é uma boa solução, porque fica difícil para o modelador prever todas as possíveis transições que podem ocorrer entre todas as seqüências de animações, e criar condições de posicionamento e mesclagem, sobretudo quando as transições são não-determinísticas.

Os *frameworks* e *game engines* estudados no capítulo 4, parecem utilizar, até onde se pode concluir, a abordagem de que ou o estágio de *design* resolve as questões de transição previamente ou o estágio de processamento atua diretamente nas juntas modificando suas posições para proceder às transições caso necessário [64][65][10].

As transições são tratadas durante o estágio intermediário de configuração. Neste estágio o projetista de jogo configurará as seqüências que serão reproduzidas a cada mudança de estados, como será visto mais adiante. Também configurará atributos que farão com que as transições ocorram de forma mais suaves e eficientes.

6.2 Posicionamento

Para solucionar a questão de posicionamento uma abordagem simples e eficiente foi adotada. O sistema de coordenadas foi remapeado para o ponto

central do modelo no *frame* onde a transição se inicia, na seqüência de animação atual. O ponto central é o ponto referencial da junta raiz do esqueleto do modelo.

Dessa forma, as coordenadas de todos os vértices dos *frames* (*keyframes* ou *intermediate frames*) da seqüência de animação seguinte são modificadas fazendo o ponto central do modelo da seqüência seguinte coincidir com o ponto central do modelo no *frame* atual da seqüência atual.

A operação para um vértice genérico chamado v_f, da sequencia seguinte ,remapeado para o ponto v_r, sendo c_a o ponto central do modelo na seqüência atual, e c_f o ponto central da seqüência seguinte, é:

$$v_r = v_f + (c_a - c_f) \qquad (2)$$

A figura 6.2 mostra o problema do posicionamento graficamente e a figura 6.3 mostra a solução proposta.

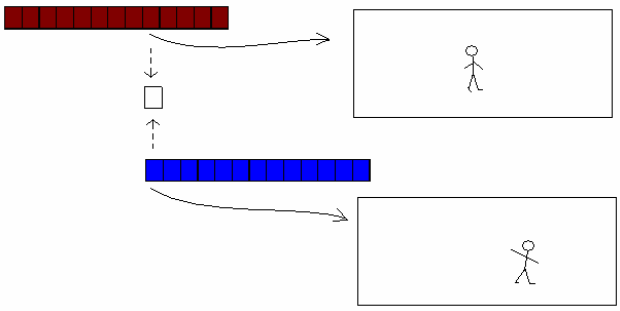

Figura 6.2. A questão do posicionamento distante em *frames* alinhados

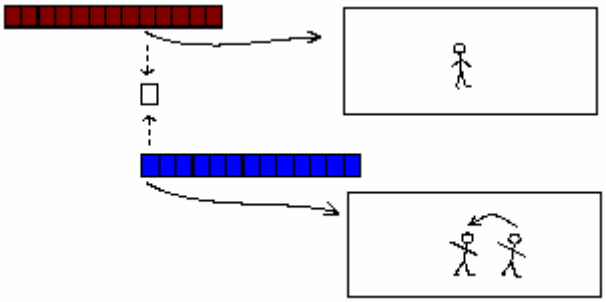

Figura 6.3. Remapeamento do sistema de coordenadas

O posicionamento também é muito útil na situação em que as animações são reproduzidas continuamente (repetidas vezes) e o último *frame* embora seja idêntico ao primeiro *frame* está muito afastado dele. Um exemplo típico desse que pode ser citado é de uma animação de caminhada. O primeiro *frame* apresenta o personagem parado, ele então começa a se mover e dá alguns passos voltando a posição de parado. Os *frames* inicial e final são idênticos mas estão em posições bem diferentes com relação às coordenadas do ambiente.

Essa situação pode ser entendida como uma transição entre duas animações idênticas sem mesclagem. O *frame* final, da seqüência atual, é seguido imediatamente (sem fase de mesclagem) pelo *frame* inicial da seqüência seguinte. Com o posicionamento, a posição do *frame* final da seqüência atual coincide com a posição do *frame* inicial da seqüência seguinte.

No exemplo da seqüência de animação de caminhada o posicionamento faz com que o pesonagem “continue andando”, e não “ande e volte para trás” como acontece sem a utilização do posicionamento.

6.3 Mesclagem

A solução adotada para o problema de mesclagem também é muito simples, isso porque, dessa forma as transições podem ser realizadas de forma visualmente convincentes e ao mesmo tempo não trazem um significativo *overhead* ao estágio de processamento.

A idéia principal, baseada em [22], é a interpolação linear das duas seqüências de animação fazendo um desaparecimento gradual da seqüência atual ao mesmo tempo em que é feito um aparecimento gradual da seqüência seguinte.

Sendo a seqüência atual A_a e a seqüência seguinte A_f, e A o resultado final no tempo *t*, a operação de mesclagem é mostrada a seguir:

$$A = (1 - \alpha t).\ A_a + (\alpha t).\ A_f \qquad \alpha \in [0, 1] \quad (3)$$

É importante ressaltar que o produto αt varia dentro do intervalo [0, 1] conforme o tempo passa, sendo α o fator de interpolação. Além disso, a seqüência atual e seqüência seguinte estão mudando também. Então a cada instante os *frames* (*keyframes* ou *intermediate frames*) são diferentes tanto para a seqüência atual como para a seqüência seguinte.

A figura 6.4 mostra a idéia de interpolação utilizando cores para representar o efeito de interpolação entre as duas seqüências de animação. Observa-se que durante processo de mesclagem não há mais diferenciação entre *keyframes* e *interpolation frames*. Eles são considerados simplesmente *frames* de uma seqüência de animação a serem interpolados com o correspondente *frame* alinhado da outra seqüência de animação.

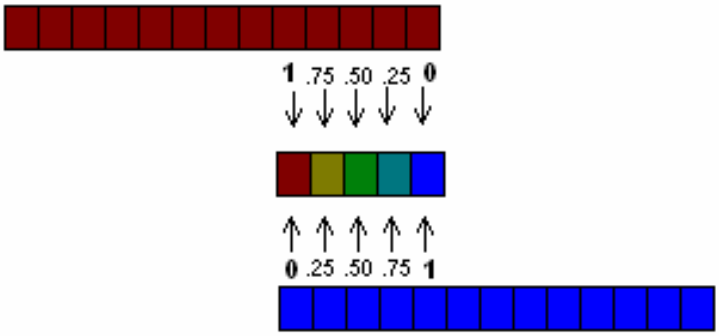

Figura 6.4. Mesclagem com interpolação linear

Como mencionado em [22], não se pode garantir que essa abordagem para mesclagem de seqüências de animação funcione como transições do mundo real. Contudo, ela apresenta bons resultados desde que as seqüências de animação estejam alinhadas e sejam similares.

Alguns problemas apresentados por essa técnica, foram constatados durante o desenvolvimento do projeto do sistema de controle de animações de

personagens do *framework* Guff. Esses problemas foram analisados e são apresentadas soluções que minimizam ou resolvem esses problemas. Tais soluções tornaram-se configurações no estágio intermediário de configuração permitindo ao projetista de jogo tratar os problemas do processo de mesclagem convenientemente.

Foram identificadas quatro situações, em que a mesclagem baseada em interpolação linear (às vezes também influenciada pelo posicionamento, como será mostrado a seguir), não funciona convenientemente. São elas: efeito “borracha”; efeito “vai suave e de repente salta”, efeito “vai sauve então reposiciona e então volta” e o efeito “bate e volta”.

No efeito “borracha”, a animação, durante a transição, parece estar sendo reproduzida com a malha do modelo do personagem sendo distorcida dando a impressão de que o personagem é feito de borracha. Intuitivamente pode-se inferir que esse efeito ocorre pela grande diferença entre os *frames* alinhados (cada um pertencente a uma das seqüências) que estão sendo interpolados. Contudo, após terem sido realizados experimentos com várias seqüências de animação (como pode ser visto no capítulo 8) pode-se constatar que a questão é mais complexa.

Mesmo seqüências que tem *frames* alinhados com poucas diferenças podem apresentar distorções. Quando as diferenças são analisadas em cada uma das 3 dimensões individualmente pôde-se constatar que algumas transições apresentavam maiores variações que outras. Além disso, é possível que a interpolação linear cause posições corretas em termos matemáticos mas indesejáveis no que se refere à percepção humana dos movimentos para um determinado personagem.

A solução proposta, para este efeito, é escolher animações que ao serem mescladas apresentem uma transição mais suave. No estágio intermediário de configuração, o projetista de jogo pode visualizar a demonstração de uma

transição previamente configurada podendo assim, observar se esta transição apresenta boa fluidez visual.

O efeito “vai suave e de repente salta” como o nome diz, ocorre quando uma transição vai suave e então de repente ela salta para um *frame* sem suavidade. Esse efeito ocorre porque às vezes a seqüência de animação atual termina antes que a fase de mesclagem (conjunto de *frames* alinhados que estão sendo interpolados) termine. Isso acontece quando o projetista de jogo configura a animação atual para não recomeçar durante a fase de interpolação. Após o término da seqüência atual, no restante da fase de mesclagem, a seqüência seguinte segue sozinha, dando a impressão de não suavidade. A Figura 6.5 mostra como esse efeito ocorre. No quarto *frame* interpolado da fase de mesclagem acontece um salto porque a seqüência atual já terminou.

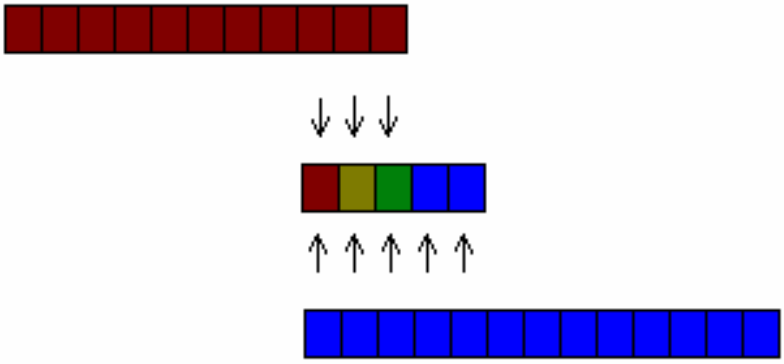

Figura 6.5. Efeito “vai suave e de repente salta”

A solução proposta neste caso é a escolha de animações atuais maiores possíveis. Mesmo fazendo isso, na ocorrência de transições não determinísticas, o efeito ainda pode ocorrer. Contudo, quanto maior a seqüência atual menor a incidência desse efeito. Em se tratando de transições determinísticas, esse problema é completamente controlado pelo projetista de jogo durante o estágio intermediário de configuração, uma vez que ele pode escolher exatamente em qual *frame* da seqüência atual a fase de mesclagem será iniciada, e fazendo a fase de mesclagem terminar antes do fim ou exatamente no último *frame* da seqüência atual.

O estágio intermediário de configuração também apresenta a possibilidade de visualização de demonstrações prévias das transições para que o projetista de jogo tenha uma boa noção de como essas transições ocorrerão no estágio de processamento do jogo.

O efeito “vai suave reposiciona e então volta”, como o nome diz, ocorre quando a transição vai suave e de repente o modelo aparece em outro ponto do ambiente e então ele volta (deslizando) para a localização correta onde ele deveria estar. Esse efeito ocorre pelo mesmo motivo do caso anterior, mas o efeito é diferente porque, o projetista de jogo permitiu que a animação atual reiniciasse durante a fase de mesclagem. Quando a seqüência atual reinicia o modelo sofre erroneamente o efeito da operação de reposicionamento e é por isso que o modelo aparece em um local diferente sendo através do restante da fase de mesclagem trazido de volta ao seu local original correto. A figura 6.6 mostra esse efeito.

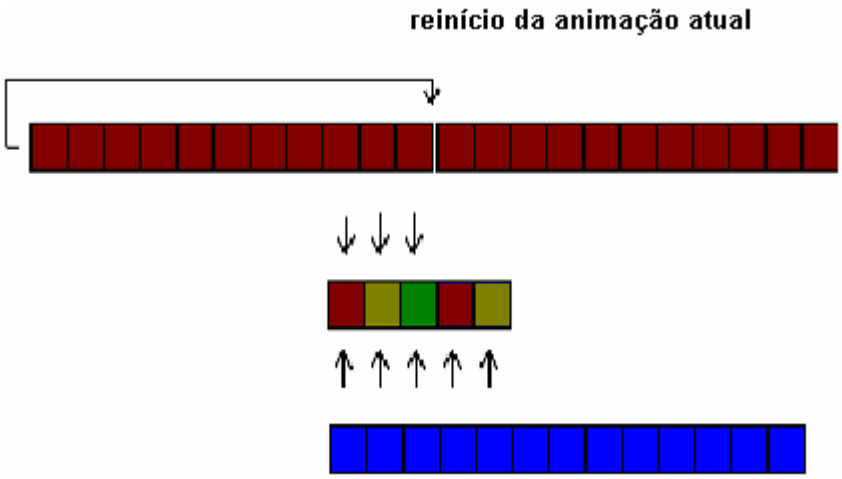

Figura 6.6. Efeito “vai sauve então reposiciona e então volta”

A solução, neste caso, é não permitir que a seqüência reinicie durante o processo de mesclagem. De fato, esse efeito também pode ocorrer com a seqüência seguinte, por isso, também é recomendável que ela não seja reiniciada durante a fase de mesclagem. Além disso, a seqüência seguinte deveria ser escolhida de forma que ela nunca termine antes da fase de

mesclagem. Todas essas configurações podem ser realizadas no estágio intermediário de configuração.

O efeito “bate e volta”, ocorre quando a transição dá a impressão de que houve uma ida e volta na seqüência de animação. Para entender esse efeito é necessário se aprofundar no relacionamento entre as mudanças de estados, as seqüências de animação e as transições.

Quando uma transição não-determinísticas ocorre, a seqüência de animação atual começa a ser substituída pela seqüência de animação seguinte. De uma maneira geral, a seqüência seguinte será substituída por uma nova seqüência quando o personagem voltar ao estado de repouso. Por exemplo, quando o personagem recebe um disparo, ele corre assustado, até que ele sinta que está a salvo. Então ele pára de correr e volta ao seu estado de repouso.

Como foi dito antes, para transições determinísticas, o projetista de jogo deve escolher o *frame* em que a seqüência atual começará a ser substituída pela seqüência seguinte, a qual representa o estado de repouso. Se ele escolher um *frame* que está muito próximo do começo da seqüência atual pode ocorrer que esse *frame* já esteja participando da fase de mesclagem da transição anterior (transição não-determinística).

Como a fase de mesclagem não permite o início de uma nova fase de mesclagem antes que a atual finalize, a segunda transição (determinística) não pode ser realizada. Por isso, a seqüência que está entrando tem que terminar e reiniciar para então iniciar a transição determinística. O segundo ciclo causa o efeito de “bate e volta” na transição das animações. A figura 6.7 mostra como esse efeito ocorre.

Para evitar esse efeito o projetista de jogo deve configurar as seqüências de animação para transições determinísticas, grandes o suficiente para suportar uma transição “de entrada”, seguida de uma parte da seqüência de

animação sem transições e finalmente uma parte que tomará parte na transição “de saída”.

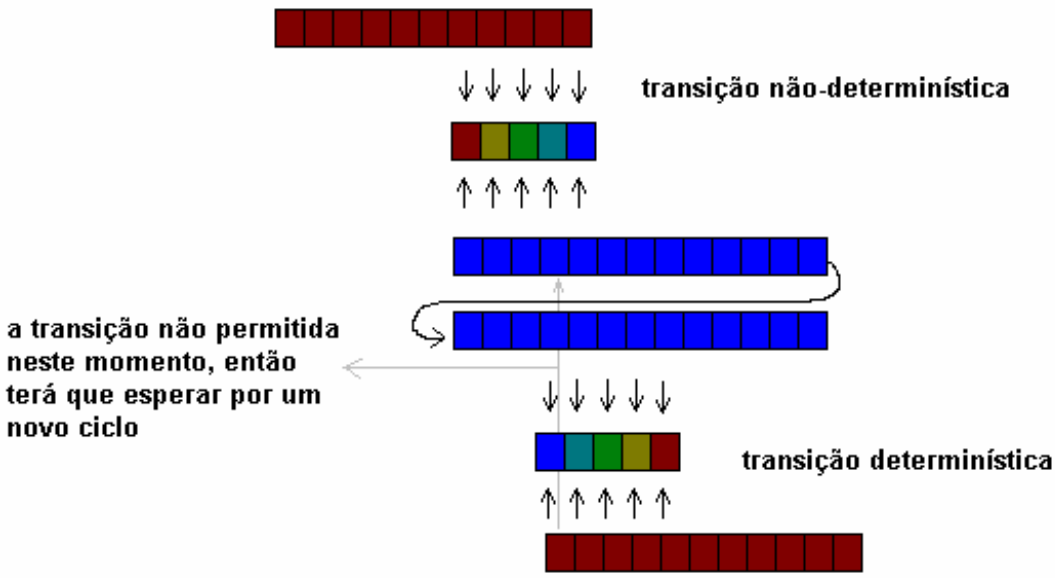

Figura 6.7. Efeito “bate e volta”

6.4 Conclusão do Capítulo

O sistema de controle de animações de personagens do *framework* Guff trata duas questões fundamentais para a transição entre seqüências de animação: posicionamento e mesclagem.

O posicionamento é realizado através do remapeamento de coordenadas dos vértices de *frame* inicial de uma seqüência a ser iniciada, baseado na posição atual (do *frame* corrente da seqüência atual) do personagem. A mesclagem é realizada através da interpolação linear das coordenadas de cada vértice em dois *frames* alinhados das duas seqüências de animação que estão sendo mescladas.

As soluções para efeitos colaterais indesejáveis das abordagens utilizadas para o posicionamento e a mesclagem tornaram-se configurações no estágio

intermediário de configuração, permitindo ao projetista de jogo criar transições eficientes e ao mesmo tempo suaves.

7. Alto nível do sistema de controle de animações de personagens

Assim como o baixo nível lida com as seqüências de animação, o alto nível é responsável pelo relacionamento do personagem com os componentes do ambiente em que o personagem está inserido como: outros personagens, obstáculos e relevos (paredes, escadas, rampas, pontes), objetos (armas, kits de reabilitação de saúde).

A maneira como o personagem percebe esses componentes é a primeira preocupação do alto nível. O personagem deve ser capaz de detectar outros personagens (e distinguir personagens parceiros e oponentes), visualizar objetos e se locomover pelo meio ambiente. A detecção em geral utiliza um sistema de detecção e resposta de colisão. A locomoção costuma se basear em algum tipo de sistema de *path finding*. Nesse primeiro estágio que é chamado de detecção e locomoção (ou *kinematic* [22]), o personagem não possui capacidade de reagir a eventos ou escolher ações baseadas no contexto.

Em um estágio mais sofisticado as percepções que o personagem recebe podem gerar ações imediatas. Por exemplo, o personagem pode atacar se sentir que o seu “território” foi invadido. Ele também pode se sentir “acuado” e fugir se for atingido por um disparo. Esse estágio é conhecido como reativo (*reactive* [22]). De uma forma geral esse estágio pode ser bem definido com uma máquina de estados finitos [22].

Um estágio superior pode ser considerado, onde o personagem, baseado no contexto do seu meio ambiente, e em objetivos bem definidos, pode tomar decisões que envolvem ações complexas (formadas por vários passos de ações simples). Além disso, pode acontecer de um personagem possuir objetivos conflitantes em um mesmo momento, como por exemplo, se refugiar e, ao mesmo, tempo alvejar o inimigo [74], sendo necessária a escolha da

decisão a ser tomada. Esse estágio pode ser chamado de estratégico (ou *goal directed* [22]). Este estágio pode ser desenvolvido baseado em tecnologias de inteligência artificial mais sofisticadas como redes neurais [75] ou lógica fuzzy [21].

Essa subdivisão do alto nível pode ser vista na figura 7.1. O conjunto de funcionalidades do alto nível é às vezes chamado de agente de inteligência do personagem [22].

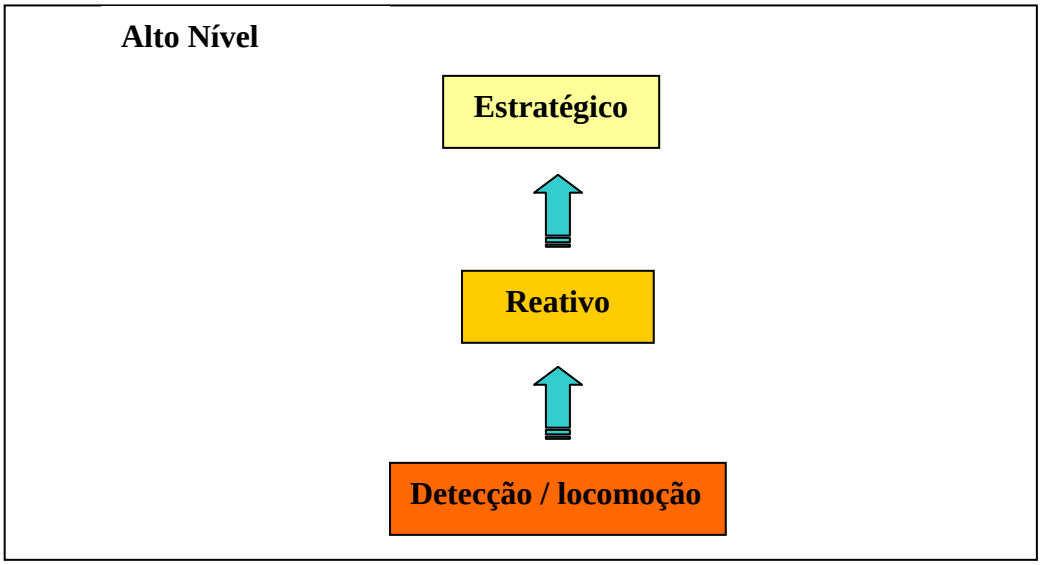

Figura 7.1. Alto nível de um sistema de controle de animações de personagens

Alguns textos apresentam variações na definição desses estágios como em [76], onde o estágio estratégico é dividido em *long-term decision* e *short-term decision* e o estágio de detecção e locomoção é dividido em *perception, localization* e *motion*.

7.1 Alto nível do sistema de controle de animações de personagens do framework Guff

O sistema de controle de animações de personagens do *framework* Guff foi desenvolvido em duas camadas de abstração (baixo nível e alto nível)

porque dessa forma, é possível que se possa desenvolver cada um dos dois níveis sem que seja necessária interferência direta no outro.

No projeto desta dissertação, optou-se por uma implementação simplificada do alto nível. Isso porque, neste primeiro estágio de desenvolvimento, uma ênfase maior foi dada ao baixo nível e o alto nível figurou mais como uma ferramenta de teste para o baixo nível. Ou seja, as requisições do alto nível resultaram em seqüências de animações convenientemente providas pelo baixo nível. A máquina de estados foi construída via código por isso sua estrutura é em princípio fixa.

Dessa forma, foi desenvolvido um sistema de percepção preliminar onde o personagem percebe se a câmera ou outro personagem entrou em seu campo de visão e também se ele foi atingido por um disparo.

A percepção de um intruso no campo de visão do personagem é baseada em uma esfera circundante em torno do personagem e no algoritmo de colisão esfera-ponto citado em [11]. A percepção de um disparo que tenha atingido o personagem é baseada em uma caixa alinhada circundante (AABB) em torno do personagem e no algoritmo de colisão caixa-raio [11].

Um estágio reativo foi desenvolvido baseado em uma máquina de estados finitos como pode ser visto na figura 7.2. Uma máquina de estados finitos é uma representação de um sistema baseado em regras bem definidas com eventos sofridos pelo personagem e respostas associadas a estes eventos. Cada estado define uma resposta que, por sua vez resulta em uma requisição de seqüência de animação ao baixo nível.

A máquina de estados finitos possui a capacidade de “estimar” o novo estado de um personagem baseado em seu estado atual e no evento sofrido por esse personagem. Caso não sofra eventos, o personagem permanece em um determinado estado continuamente. Podem existir estados dos quais o

personagem não pode mais sair, como é caso do estado de morte, por exemplo.

A cada estado está associada uma seqüência de animação que representa, durante o estágio de processamento do jogo, este estado. Quando o personagem muda de estado inicia-se uma transição entre a seqüência atual e a seqüência seguinte representando a mudança de estado do personagem.

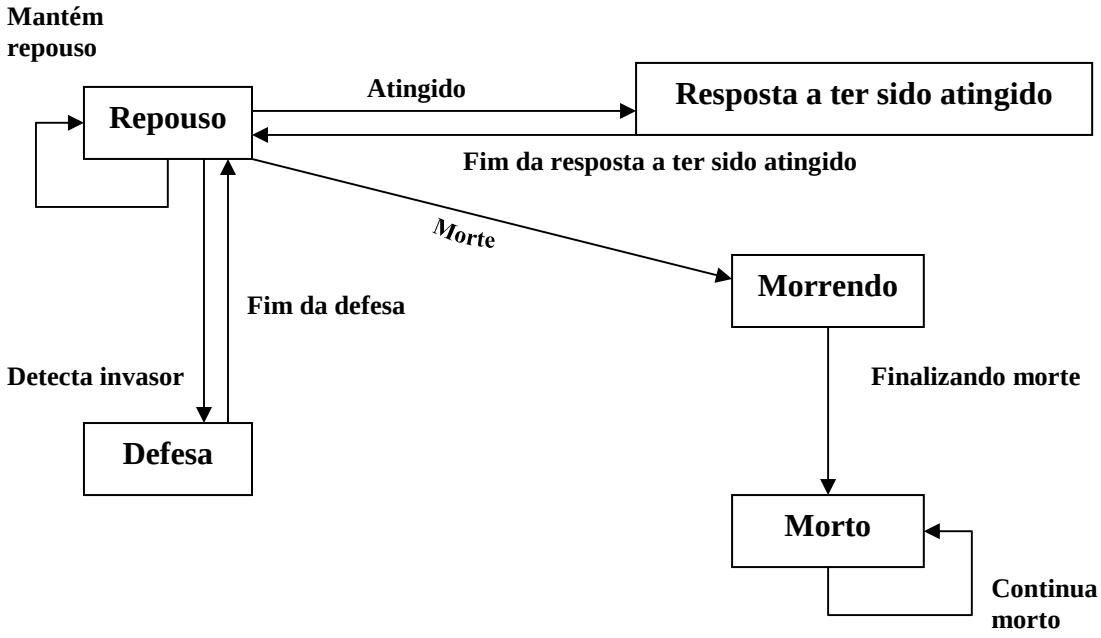

Figura 7.2. Máquina de estados finitos do estágio reativo

Os eventos: “detecta invasor”, “atingido” e “morte” ocasionam transições não-determinísticas. Os demais eventos ocasionam transições determinísticas, sendo permitido que o projetista de jogo escolha exatamente o momento em que o personagem mudará de estado, ou em se tratando do baixo nível, em qual *frame* da seqüência atual começará a transição para a seqüência seguinte.

O evento “detecta invasor” ocorre quando a câmera ou outro personagem invade o território do personagem (interior da esfera circundante). O evento “atingido” ocorre quando um tiro disparado atinge a AABB do personagem.

O personagem possui uma “resistência”, isto é, ele pode receber um número limite de disparos sem morrer. Tendo recebido um número de disparos além do limite, o evento “morte” é acionado e o personagem começa a morrer. Quando a seqüência de animação de morte vai chegando ao fim o personagem fica em um estado de “morto” do qual não pode mais sair.

Para todos os eventos, é permitido ao projetista de jogo configurar seqüências de animações para as transições de mudanças de estado. No caso de transições determinísticas o projetista de jogo deve também definir em qual *frame* da seqüência de animação atual ela começará a ser substituída pela próxima seqüência de animação. Essas configurações são feitas através do estágio intermediário de configuração.

7.2 Conclusão do Capítulo

O alto nível de um sistema controle de animações de personagens é responsável por detectar o meio ambiente e possibilitar a locomoção do personagem. Também é responsável por gerar respostas imediatas a estímulos recebidos. Pode ainda prover estratégias baseadas em metas e objetivos que serão fragmentadas em tarefas menores e executadas pelo personagem.

O alto nível do sistema de controle de animações de personagem do *framework* Guff foi criado com o intuito inicial de testar a capacidade do baixo nível em prover seqüências de animações corretamente. Ele possui um sistema de detecção e resposta a estímulos baseados em algoritmos de colisão e em uma máquina de estados finitos.

Essa implementação permite que os personagens tenham um comportamento reativo ao ambiente em que estão inseridos. Ela também permite que o alto nível sirva como módulo de testes e homologação do baixo nível do sistema de controle de animações de personagens do *framework* Guff.

8. Resultados Experimentais

Para determinar a eficácia do sistema de controle de animações de personagens e da utilização de um estágio intermediário de configuração entre os estágios de *design*, alguns testes foram executados [77] [62], cujos resultados são mostrados neste capitulo. Além disso, algumas conclusões são inferidas a partir desses resultados.

8.1 Testes e Resultados

Os resultados apresentados aqui são de testes efetuados com animações de personagens do jogo DOOM3 (Archvile, Cherub, Guardian). O personagem Archvile na seqüência de animação “attack2” é mostrado na figura 8.1. Essa seqüência contém 35 *keyframes*.

Os gráficos de resultados mostrados nessa seção são referentes a essa seqüência de animação. Embora resultados semelhantes tenham sido obtidos em outras seqüências e com outros personagens, essa seqüência foi escolhida para sumarizar estes resultados. As características da máquina onde os testes foram executados são: Pentium IV com 3GHz e 1GBytes de memória RAM e placa de vídeo ATI-RADEON 9250 e sistema operacional Windows XP Professional.

O primeiro teste foi feito com uma comparação entre as três abordagens de evolução para o efeito *slow in* e *slow out*: senoidal, parabólica e *spline*. Os resultados foram satisfatórios com relação ao efeito desejado para as três abordagens. Contudo, a abordagem de utilização de *splines* se mostra mais propícia para impressão de efeitos de evolução uma vez que ela pode modular funções de formas variadas permitindo ao usuário uma gama maior de efeitos de evolução sobre as seqüências de animação [77] [62].

A seguir foi feito um teste comparativo das abordagens com e sem a uniformidade de número de *intermediate frames* baseada em distâncias dos

keyframes. Na figura 8.2, a seguir pode-se observar as distâncias com valores normalizados para cada segmento entre pares de *keyframes* contíguos da seqüência de animação estudada.

Figura 8.1. Personagem Archvile do jogo DOOM3

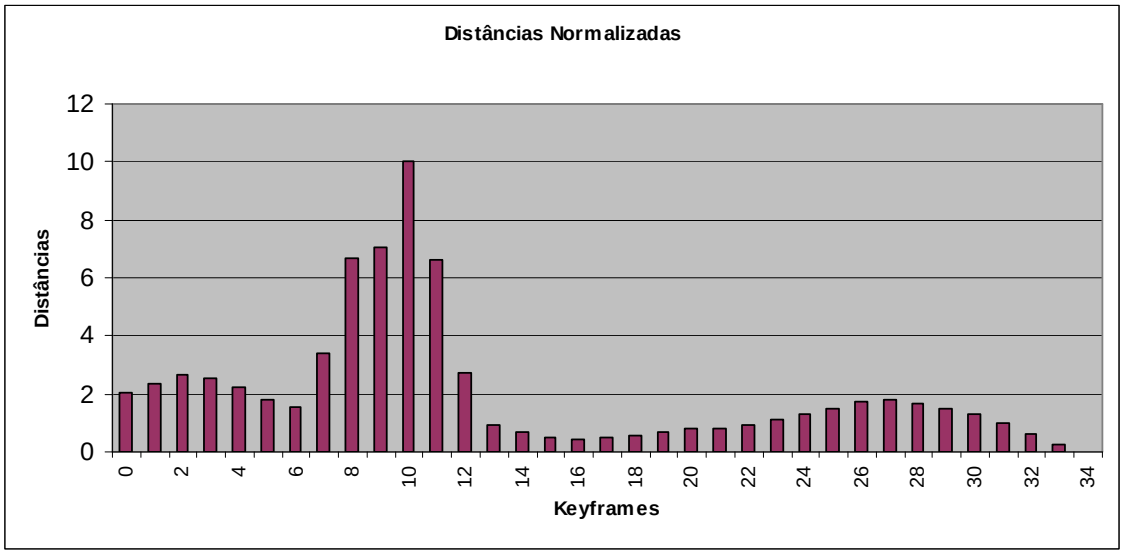

Figura 8.2. Distâncias de segmentos da seqüência de animação attack2 do personagem Archvile do jogo DOOM3 [77]

A análise visual da animação com e sem o controle de uniformidade mostrou que o controle de uniformidade funcionou bem mostrando a seqüência de animação de forma contínua do início ao fim, ao contrário da seqüência sem o controle de uniformidade.

É interessante observar que talvez, a não equalização original da seqüência possa ter sido intencional. É possível que o modelador estivesse tentando criar algum efeito de evolução deixando a seqüência desequalizada e esse efeito foi desmantelado pelo controle de uniformidade.

Uma conclusão importante a que se pode chegar a partir desse fato é que não é interessante ter evolução e uniformidade em estágios diferentes. Isso porque o estágio de *design* transforma o efeito de evolução em *keyframes* com distâncias variáveis e não em número de *intermediate frames* variáveis. Por essa razão esses efeitos são anulados pelo controle de uniformização no estágio intermediário de controle [77].

Por outro lado, é possível ter uniformização e evolução trabalhando conjuntamente em um mesmo estágio. O número de *intermediate frames* pode ser submetido aos dois controles e guardar as duas influências se for corretamente configurado para esse fim. De fato, esses dois controles devem ser realizados de forma complementar.

Na figura 8.3, a seguir são mostradas as combinações do controle de uniformidade com o controle de evolução (efeito *slow in* e *slow out*) e também um fator de *timing* para a seqüência de animação estudada. Comparando-se essa figura com a figura 8.2 pode-se perceber como o controle de evolução agiu sobre os valores obtidos no controle de uniformidade para chegar ao efeito *slow in* e *slow out*. A figura 8.3 também permite visualizar como a abordagem de utilização de *splines* é mais bem sucedida que as outras duas (senoidal e parabólica) em modular o efeito de evolução sem causar bruscas mudanças no número de *intermediate frames*, mostrando como ela é uma abordagem mais adequada ao controle de evolução por prover o projetista de jogo com mais recursos para encontrar o efeito desejado. O fator de *timing* agiu simplesmente multiplicando todos os valores obtidos pela combinação de uniformidade e evolução por um fator de 10 unidades.

Um terceiro teste foi realizado a fim de observar a atuação do estágio de processamento sobre o fator de *timing* conforme a variação da taxa de *frames* por segundo da aplicação.

Utilizou-se um fator de corte de 60 *frames* por segundo, o que significa que acima desse valor o fator de *timing* não foi afetado. Abaixo desse valor, o fator de *timing* foi incrementado ou diminuído seguindo as variações da taxa de *frames* por segundo da aplicação.

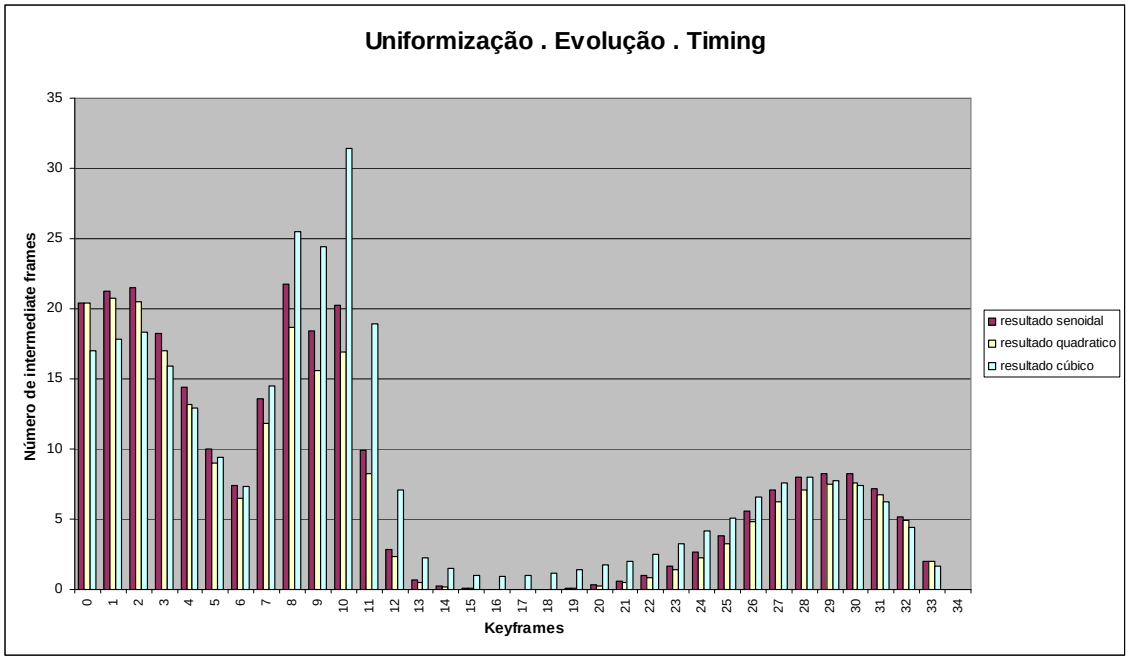

Figura 8.3. Combinação dos controles de uniformização, evolução e *timing* [77]

Os gráficos das figuras 8.4 e 8.5, a seguir, mostram as variações para 7 e 10 cópias, respectivamente, do personagem se movendo sendo renderizadas simultaneamente (para valores menores como 1, 2 e 5 cópias a taxa de *frames* por segundo se manteve superior a 60 *frames* por segundo).

Seguindo as variações de quadros por segundo o fator de *timing* fez com que o tempo de execução se mantivesse mais estável. Não se pode, contudo garantir que esse tempo será constante uma vez que os fatores de evolução e de equalização não variam com a taxa de quadros por segundo. Assim o número de quadros intermediários varia com a taxa de quadros por segundo da

aplicação, mas mantém os comportamentos de equalização e evolução durante o processo [77].

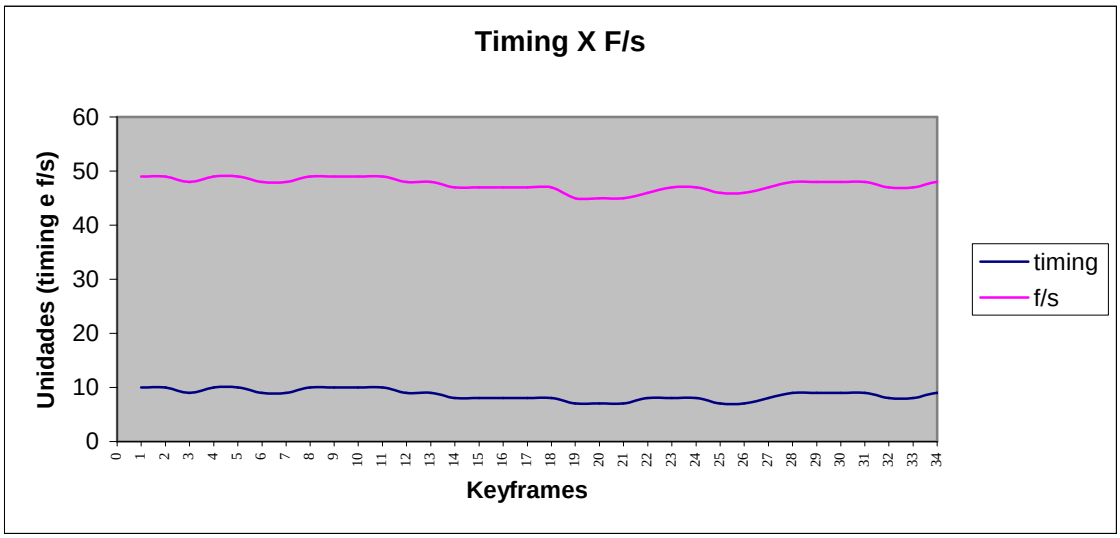

Figura 8.4. Variação de *frames* por segundo (f/s) e Fator de *timing* (*timing*) com 7 cópias do personagem [77]

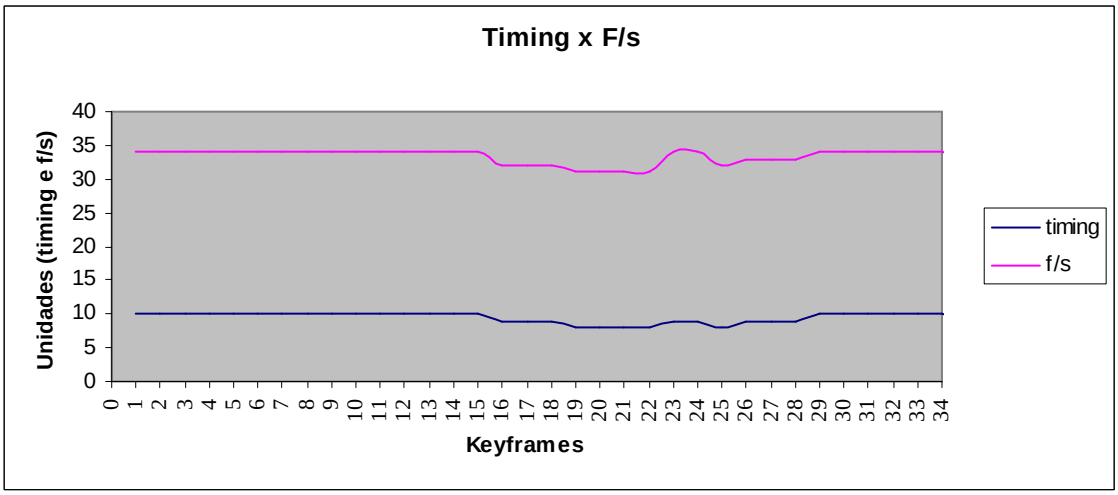

Figura 8.5. Variação de *frames* por segundo (f/s) e Fator de *timing* (*timing*) com 10 cópias do personagem [77]

Um último teste foi feito para definir as distâncias entre *frames* alinhados durante a fase de mesclagem de uma transição entre duas seqüências de animação. O teste mostrado a seguir foi feito iniciando-se da animação “walk” do personagem Archvile seguindo-se as animações “evade_left”, “pain_head1”,

“sight2” e “attack1”. A figura 8.6 mostra as distâncias médias entre os *frames* alinhados em cada uma das transições (a, b, c e d respectivamente).

Pode-se reparar como todas tem valores de distâncias não muito diferenciados. Contudo, na prática as transições da animação “walk” para as animações “pain_head1” e “attack1” apresentam leves distorções do modelo (efeito “borracha”). Por outro lado, as transições da animação “walk” para as animações “evade_left” e “sight2” são bastante convincentes.

Esse resultado sugere que não é apenas a distância entre *frames* alinhados na fase de mesclagem de uma transição que define se o efeito “borracha” ocorrerá ou não ocorrerá.

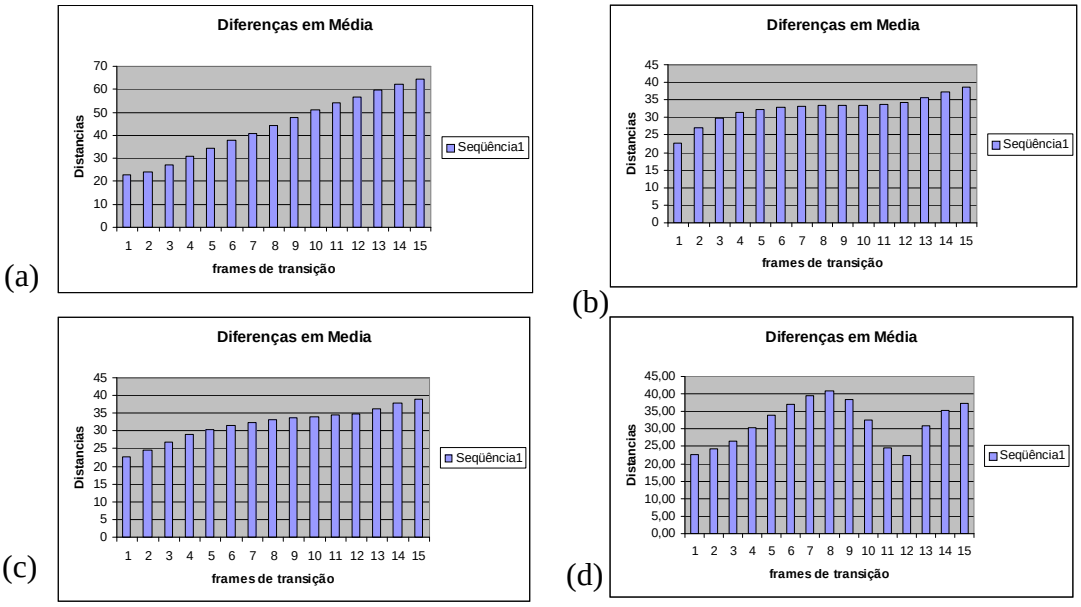

Figura 8.6. Resultados obtidos de transições da animação “walk” para as animações “evade_left”, “pain_head1”, “sight2” e “attack1” ((a), (b), (c) e (d) respectivamente)

8.2 Conclusão do Capítulo

O baixo nível respondeu bem às requisições de seqüências de animação do alto nível. A máquina de estados finitos serviu bem ao propósito de teste do baixo nível.

O estágio intermediário de configuração permitiu criar seqüências com uniformidade, evolução e *timing*. Também permitiu associar corretamente as seqüências de animação à máquina de estados finitos na qual está baseado o alto nível.

Os controles de uniformidade, evolução e *timing* responderam bem às expectativas. Constatou-se que a função B-*spline* uniforme cúbica é mais adequada ao controle de evolução. Também se pode observar que os estágios de uniformidade e evolução devem ser utilizados de forma complementar e não de forma concorrente.

O ajuste do fator *timing* durante o estágio de processamento funcionou corretamente corrigindo esse fator baseado na taxa de *frames* por segundo do jogo. Esse ajuste não comprometeu os controles de uniformidade e evolução, como era esperado.

As transições entre seqüências de animações podem apresentar distorções (efeito “borracha”) mesmo quando a distância média entre os *frames* alinhados é pequena.

9. Conclusão

Jogos de computador precisam adquirir, gerenciar e reproduzir animações de personagens de forma a otimizar a utilização de recursos (sobretudo memória e processamento) sem comprometer o desempenho geral da aplicação.

Esse trabalho apresentou princípios para a utilização de animações de personagens em jogos, partindo desde os jogos 2D até chegar aos jogos 3D de última geração. Um enfoque especial foi dado a técnicas com as quais o usuário tivesse uma percepção real em 3 dimensões.

Frameworks para a criação de jogos de computador, em geral, possuem um sistema de controle de animações de personagens que permite obter, gerenciar e reproduzir animações de personagens. Esses sistemas são em geral divididos em duas camadas: alto nível, que trata da inteligência do personagem e baixo nível, que resolve as requisições por seqüências de animação.

Este trabalho apresentou também os princípios e técnicas no qual se baseou o projeto de implementação do sistema de controle de animações de personagens para o *framework* Guff. Esse projeto foi responsável pela implementação das camadas de baixo nível e de alto nível do sistema de controle de animações de personagens.

O baixo nível utiliza técnicas de animação *keyframing*. Uma inovação deste projeto é utilização de um estágio intermediário de configuração entre os estágios de *design* e processamento, que possibilita a manipulação da quantidade de *intermediate frames* para cada segmento de uma seqüência de animação. Com essa abordagem é possível definir estas quantidades de *intermediate frames* com controles de uniformidade, evolução e *timing*.

O estágio intermediário de configuração também permite que o projetista de jogo associe seqüências de animação aos estados de comportamento resultantes do alto nível. Além disso, disponibiliza a configuração de transições entre seqüências de animações. A ferramenta do estágio intermediário de configuração pode ser vista no apêndice A.

O alto nível foi criado, em primeira instância para prover um ambiente de teste para o baixo nível. Ele foi desenvolvido com um sub-estágio, de detecção, que permite ao personagem se relacionar com o meio ambiente e com um sub-estágio de resposta a eventos, baseado em uma máquina de estados finitos.

9.1 Dificuldades

A maior dificuldade que se encontra ainda hoje no desenvolvimento de ferramentas para jogos é o hermetismo com que empresas de jogos e desenvolvedores de jogos autônomos disponibilizam informações.

Além disso, há a falta de padronização de conceitos, definições e termos na área de desenvolvimento de jogos. Talvez essa deficiência se deva ao fato de que jogos de computador ainda sejam considerados “aplicações de menor importância” por se tratarem de *softwares* de entretenimento.

9.2 Trabalhos Futuros

Trabalhos futuros apontam para o desenvolvimento de novas funcionalidades no alto nível, como *path finding* e também um estágio de controle estratégico baseado em metas e objetivos. Além disso, pode-se pensar na possibilidade de criação de movimentos de um personagem no estágio de processamento baseado em eventos detectados pelo agente do personagem. Assim seria possível a manipulação direta de do personagem através de *scripts* ou de um sistema de *inverse kinematics* que possibilitaria seqüências de animação criadas em tempo real.

Outra questão é a criação de um sistema de detecção e resposta de colisão entre personagem-personagem e personagem-cenário que pode estender a máquina de estados em que está baseado atualmente o estágio reativo do alto nível. Além disso, as funcionalidades de transição já existentes podem ser adaptadas para a representação de animações faciais e de animações de superfícies não rígidas como peças de vestuário.

Um estudo mais analítico das questões transição pode revelar possibilidades de soluções dessas questões no estágio de processamento através da manipulação direta de juntas, extrapolação de *frames* de uma seqüência de animação e abordagens alternativas à interpolação linear, sendo configuradas pelo projetista de jogo no estágio intermediário de configuração.

Finalmente é interessante pensar na possibilidade de estender o estágio intermediário de configuração de forma que ele possibilite configurar outras funcionalidades existentes no *framework* Guff através da geração de *scripts*. Atualmente todo esse trabalho de configuração é feito via código. O *software* de configuração pode ser estendido a fim de propiciar uma interface amigável para que o projetista de jogo configure opções do jogo.

Referências Bibliográficas

[1] Valente, L. GUFF: Um framework para desenvolvimento de jogos. Niterói. UFF. 2005. at (02/2006): http://www.ic.uff.br/~lvalente/en/projects.html

[2] Crawford, C. The Art of Computer Game Design. Vancouver. Washington State University. 1997. Disponível em (07/2005): http://members.xoom.com/kalid/art/art.html

[3] Foley, J., van Dam, A., Feiner, S. K., Hughes, J. F. Computer Graphics: Principle and Practice, 2nd Edition. Reading. Addison-Wesley. 1990.

[4] Feldman, A. Designing Arcade Computer Game Graphics. Plano. WordWare Publishing, Inc. 2001.

[5] Hall, J.; Loki Software Inc. Programming Linux Games. San Francisco. No Starch Press. 2001.

[6] Lamothe, A. Black Art of The 3D Game Programming. Corte Madera. Waite Group Press. 1995.

[7] Herman, L. The History of Vídeo Games. Disponível em (07/2005): http://www.gamespot.com/gamespot/features/video/hov/

[8] WikiPedia. Multiplane Camera. Disponível em (09/2006): http://en.wikipedia.org/wiki/Multiplane_camera

[9] Trinta, F.; Miranda, O.; Ramalho, G. Jogos isométricos em dispositivos móveis. Recife. UFPE. 2000.

[10] IdSoftware. Disponível em (07/2005): http://www.idsoftware.com/

[11] Möller, T.; Haines, E. Real Time Rendering. Natick. A. K. Peters. 1999.

[12] Abrash, M. Michael Abrash´s Graphics Programming Black Book, Special Edition. Scottsdale.The Coriolis Group. 1997.

[13] All Game´s Genre and Style List. One Stop Group. 2000. Diponível em (21/06/06) : http://www.allgame.com/genres.html

[14] Azevedo, E.; Conci, A. Computação Gráfica. Teoria e Prática. Rio de Janeiro. Editora Campus. 2003.

[15] Wright Jr., R.; Lipchak, B. OpenGL SuperBible, Third Edition. Indianapolis. Sams. 2004.

[16] Neider, J.; Davis, T.; Woo, M. OpenGL Programming Guide. New York. Addison-Wesley Publishing Company. 1994.

[17] LaMothe, A. Tricks of the Windows Game Programming Gurus. Indianapolis. Sams.1999.

[18] Dunlop, R.; Shepherd, D.; Martin, M. Teach Yourself DirectX in 24 hours. Indianapolis. Sams. 2000.

[19] Watt, A.; Policarpo, F. 3D Games: Real-Time Rendering and Software Technology. Vol.1. New York. Addison-Wesley. 2001.

[20] Dybsand, E. AI Middleware: Getting Into Character. Game Developer. The H. W. Wilson Company. 2003.

[21] Malcher, F. Personabrum: A Fuzzy-System Based Architecture to Personality and Stimuli Modeling. São Paulo. Anais do Wjogos 2005 (pp. 189 a 199). SBGames. 2005.

[22] Watt, A.; Policarpo, F. 3D Games: Real-Time Rendering and Software Technology. Vol.2. New York. Addison-Wesley. 2001.

[23] Thalmann, M. N.; Thalmann, D. Computer Animation in Future Technologies. Miralab. 1998.

[24] Hodgins, J.; O'Brien, J. F.; Bodenheimer, R. E. Computer Animation. Atlanta. Georgia Institute of Technology. 1999.

[25] Lasseter, J. Principles of Traditional Animation Applied to 3D Computer Animation. San Rafael. Pixar. ACM. 1987.

[26] Making Doom3 Mods: Introduction. 2004. Disponível em (01/2006): http://www.iddevnet.com/doom3/

[27] Parent, R. Computer Animation Algorithms and Techniques. San Francisco. Morgan-Kaufmann. 2001.

[28] The Character Animation FAQ. 2002. Disponível em (07/2005): http://www.morrowland.com/apron/

[29] Oudshoorn, J. A. Ray Tracing as The Future of Computer Games. University of Utrecht. 1999.

[30] Taylor, P. Tweening 3-Way, or Using Vertex Shaders, Part 2. Microsoft Corporation. 2001.

[31] Marselas, H. Interpolated 3D Keyframe Animation. Game Programming Gems p 464-469. Hingham. Charles River Media. 2000.

[32] Cem, C. Eficcient Animation. Nvidia Corporation. Disponível em (07/2005): http://www.nvidia.com/developer

[33] Humphrey, B. MD2 loader and MD2 animation. Disponível em (07/2005): http://www.gametutorials.com

[34] Lamothe, A. Tricks of 3D Game Programming Gurus. Indianapolis. Sams. 2003.

[35] Hitchner, L. Real-Time 3D Computer Graphics Software System – Lecture 13, 14 and 15 –Importing External Model Files. University of Auckland. 2005. Disponível em (07/2005):
http:// http://www.cs.auckland.ac.nz/compsci707s2c/lectures/

[36] Humphrey, B. MD3 loader and MD3 animation. Disponível em (07/2005): http://www.gametutorials.com

[37] MD3 format. Disponível em (07/2005):
http://www.icculus.org/~phaeton/q3/formats/

[38] Ringuet, J. M. Three-Axis Animation: The HardShips of Animating Three Dimensional Characters in Real Time Games. Gamasutra. 2001.

[39] Mckenna, M.; Zeltzer, D. Dynamic Simulation of Autonomous Legged Locomotion. Computer Graphics, Volume 24, Número 4. 1990.

[40] Putz, M.; Hufnagl, K. Character Animation for Real-Time Application. Austria. Graz University of Technology.

[41] Hallier, J. Maddieman's Tutorial on how to animate MP characters. 2002. Disponível em (07/2005): http://hellskitchen.paynereactor.com/

[42] Anderson, E. F. Real-Time Character Animation for Computer Games. Bournemouth University. 2001.

[43] Camilo, M.; Martins, R; Hodge, B.; Sztajnberg, A. Considerações sobre Técnicas para Implementação de Skeletal Animation em Jogos 3D. Rio de Janeiro. Cadernos do IME - UERJ. 2003. Disponível em (02/2006): http://www.ic.uff.br/~mcamilo/gprogramming/skeletalanimcarcara.pdf

[44] Gattas, M.; Biasi, S. C. de. Utilização de quatérnios para representação de rotações em 3D. 2002. Disponível em (09/2006): http://www.tecgraf.puc-rio.br/~mgattass/Quaternios.pdf

[45] Hamilton, S. W. R. On a new species of imaginary quantities connected with a theory of quaternions. Proceedings of the Royal Irish Academy. Vol. 2. pg 424-434. 1843.

[46] Shoemake, K. Animating Rotation with Quaternion Curves. San Francisco. The Singer Company – Link Flight Simulation Division. 1985.

[47] Batiste, S. Squeezing the Animation. Game Developer. The H. W. Wilson Company. 2003.

[48] Blow, J. Understanding Slerp, Then Not Using It. Game Developer. The H. W. Wilson Company. 2004.

[49] MilkShape3D. Disponível em (07/2005): http://www.swissquake.ch/chumbalum-soft/ms3d/

[50] Cook, S. Adding a New MilkShape3D Model to Half-Life. 2000. Disponível em (07/2005): http://www.planethalflife.com/hlsdk2/sdk/MS3d_to_HL.htm

[51] UnrealWiki. Disponível em (07/2005): http://wiki.beyondunreal.com/wiki/

[52] Porter, B. NeHe Productions - Lesson 31. Disponível em (07/2005): http://nehe.gamedev.net/

[53] Polack, T. Trend Polack´s OpenGL Game Programming Tutorial: Model Mania. 2001. Disponível em (07/2005): http://nehe.gamedev.net/

[54] Monster, M. Sketal Animation. Disponível em (07/2005): http://home.planet.nl/~monstrous

[55] Lander, J. Slashing Through Real-Time Character Animation. Game Developer. 1998.

[56] Lander, J. On Creating Cool Real-Time 3D. Gamasutra. 1997.

[57] Hagland, T. Fast and Simple Skinning Technique. Game Programming Gems p. 471-475. Hingham. Charles River Media. 2000.

[58] Woodland, R. Filling the Gaps – Advanced Animation Using Stitching and Skinning. Game Programming Gems p. 476-483. Hingham. Charles River Media. 2000.

[59] Monster, M. Doom 3 Models. 2004. Disponível em (07/2005): http://home.planet.nl/~monstrous

[60] Valente, L; Conci, A.; Feijó, B. Real Time Game Loop Models for Single-Player Computer Games. São Paulo. Anais do Wjogos 2005 (pp. 89 a 99). SBGames. 2005.

[61] Camilo, M.; Conci, A. Guff´s Character System Animation for 3D Games. Manaus. Anais do Sibgrapi 2006. 2006.

[62] Camilo, M.; Conci, A. Guff´s Character System Animation for 3D Games and Applications. Recife. Anais do SBGames 2006. 2006.

[63] Campos, A. O que é software livre. Florianópolis. BR-Linux. 2006.

[64] Ogre 3D. Disponível em (07/2006):
http://www.ogre3d.org

[65] Cal3D. Disponível em (07/2006):
http://download.gna.org/cal3d/documentation/api/html/index.html

[66] Steed, P. Animating Real-Time Game Characters. Hingham. Charles River Media. 2003.

[67] Perez, A. Synthetica 2.0: Software for the Synthesis of Constrained Serial Chains. Utah. Procedings of Detc'04. 2004.

[68] Naylor, A. W.; Sell, G. R. Linear Operator Theory in Engineering and Science. New York. Holt, Einenart and Winston. 1998.

[69] GDC 2003: The 12 Principles of Animation Applied to 3D Animation. GamaSutra. Disponível em (01/2006): http://www.gamasutra.com/gdc2003/

[70] Cantor, J. 19 Common CG Animation Pitfalls. Sonic Pictures ImageWork. 2002.

[71] Lam, D. Animating for Convincing Human Motion. New School University. 2004.

[72] Watt, A.; Watt M. Advanced Animation and Rendering Techniques: Theory and Practice. New York. ACM Press. Addison Wesley. 1992.

[73] Kerlin, I. Applying the Twelve Principles to 3D Computer Animation. 2004. Disponível em (01/2006): http://www.artof3d.com/feature.htm

[74] Pinto, H.; Alvares, L. Redes de Comportamentos Aumentadas para Agentes em Jogos: Uma investigação de Qualidade de Seleção de Ação e Performance de Agente no Unreal Tournament. São Paulo. Anais do Wjogos 2005 (pp. 200 a 209). SBGames. 2005.

[75] Alegretti, F.; Alvares L. ChicuxBot – Genetic Algorithm Configured Behavior Network MultiAgent for Quake II. São Paulo. Anais do Wjogos 2005 (pp. 210 a 218). SBGames. 2005.

[76] Monteiro, I. Uma Arquitetura Modular para Desenvolvimento de Agentes Cognitivos em Jogos de Primeira e Terceira Pessoa. São Paulo. Anais do Wjogos 2005 (pp. 219 a 229). SBGames. 2005.

[77] Camilo, M; Conci, A. Um Estágio Intermediário de Configuração de Animações para jogos e aplicações de simulação 3D. Rio de Janeiro. Anais do SPOLM 2006. 2006.

Apêndice A

Neste Apêndice será mostrado como utilizar o sistema de controle de animações de personagens do *framework* Guff para criar aplicações 3D com personagens.

No estágio de *design* o modelador cria o modelo do personagem e também suas seqüências de animação. No estágio de configuração são configurados as seqüências de animação, as transições e o comportamento do personagem pelo projetista de jogo.

Finalmente o estágio de processamento é definido pelo desenvolvedor de jogo através das classes existentes no *framework* Guff para processamento de personagens.

A.1 Estágio de Design

No estágio de *design* o modelador deve utilizar uma ferramenta de modelagem para criar o modelo do personagem e as seqüências de animação. Depois ele deve utilizar um *software* exportador para salvar o arquivo de modelo (md5Mesh) e os arquivos de animação (md5Anim).

Atualmente existem exportadores e importadores (que permitem abrir arquivos do jogo Doom3) para as ferramentas 3D Studio Max, Maya e Blender. Em geral, os exportadores e importadores são *scripts* que devem ser instalados na ferramenta de modelagem e então podem ser ativados para abrirem (importadores) ou salvarem (exportadores) arquivos do jogo Doom3.

A figura A1 mostra um modelo de arquivo do Doom3 sendo importado para o 3D Studio e a figura A2 mostra um modelo criado no 3D Studio sendo exportado para um arquivo md5 (md5Mesh).

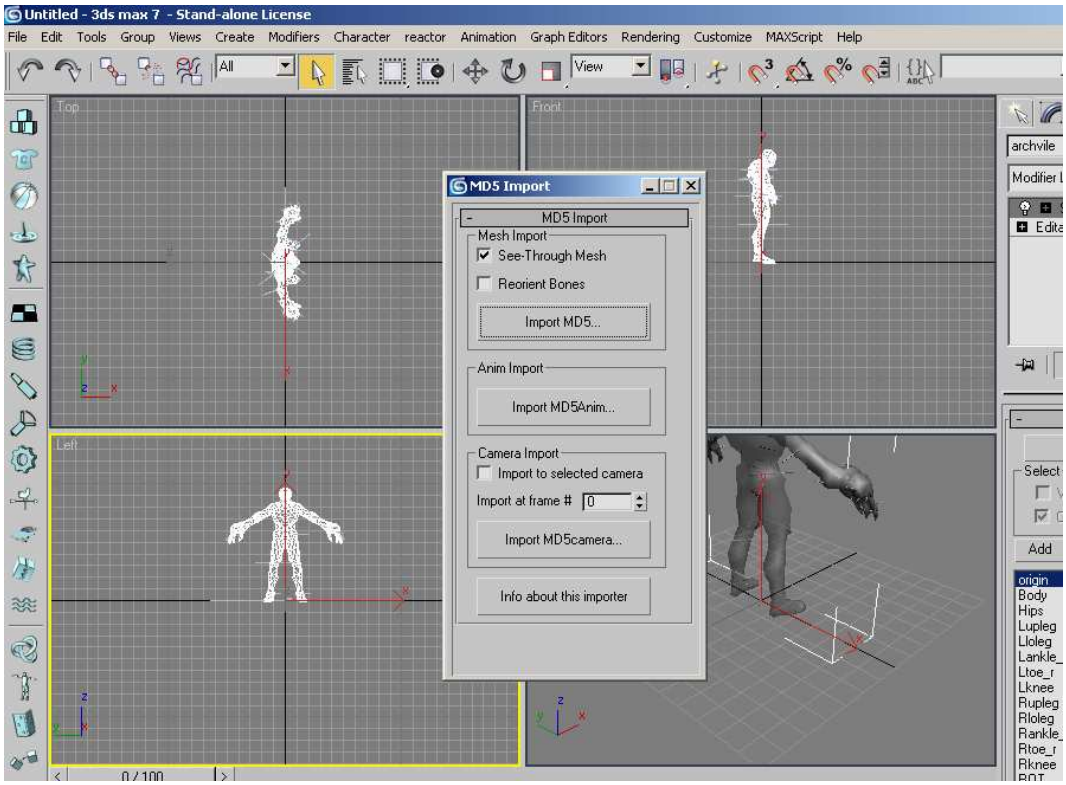

Figura A1. Modelo sendo importado para o 3D Studio

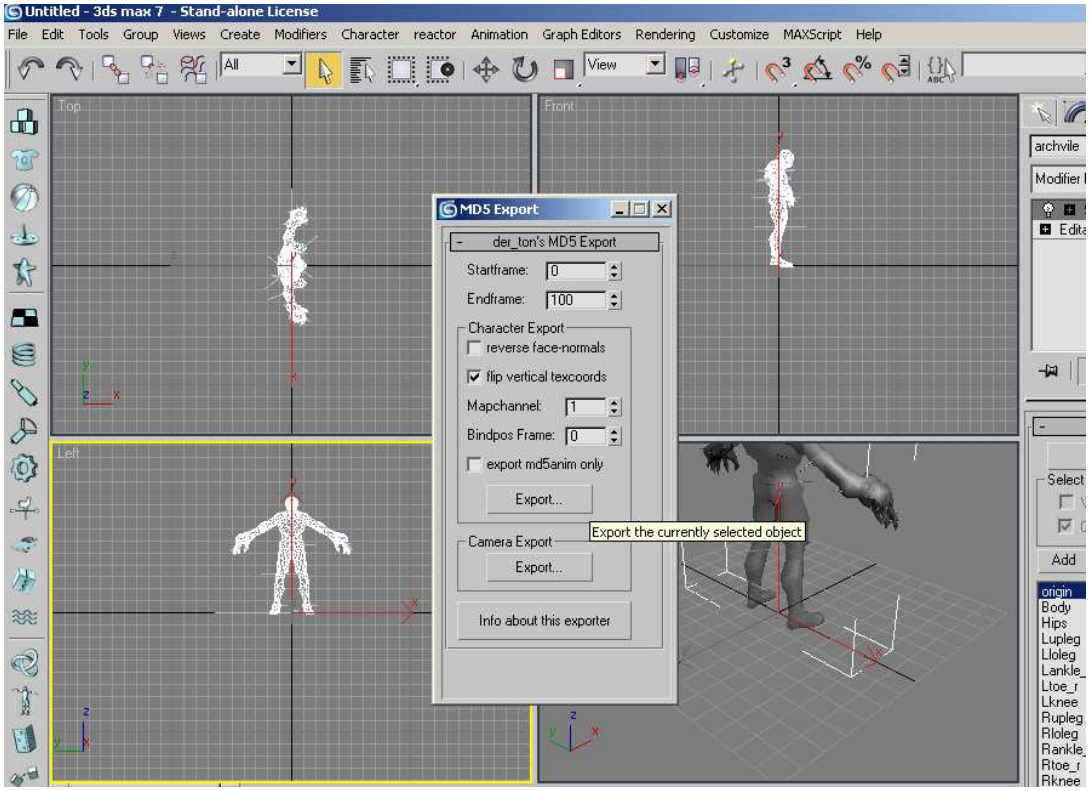

Figura A2. Modelo sendo exportado para o formato md5

A.2 Estágio de Configuração

O estágio intermediário de configuração é executado através de uma ferramenta criada no projeto no qual se baseia essa dissertação, chamada Guff Character Configurator.

Os módulos da ferramenta Guff Character Configurator permitem criar *scripts* de controles de personagens para jogos 3D. Para que essa ferramenta seja utilizada é recomendável que os modelos e animações dos personagens já tenha sido obtidos. De uma forma geral, esses modelos e animações são criados em um aplicativo de modelagem e depois convertidos para o formato md5 [26].

A figura A.3 mostra a tela inicial do Guff Character Configurator. Essa tela permite a configuração de novos personagens. Ao clicar com o botão direito do *mouse* em "New Character" uma nova janela é aberta ("Character Configuration") onde se pode configurar o nome, a posição inicial e o comportamento do personagem.

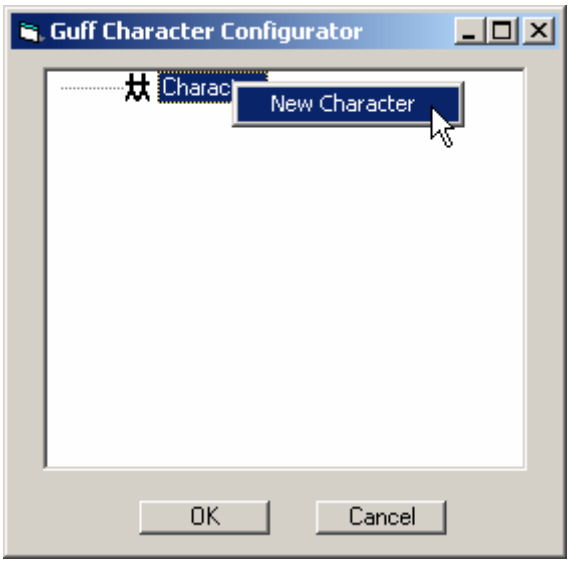

Figura A.3. Tela inicial do Guff Animation Configuration

A figura A.4. mostra a janela "Character Configuration". Ao clicar o botão "Browse Behavior ..." pode-se abrir um arquivo de *script* de comportamento de

personagem e ao clicar no botão “Create Behavior ...” a janela “Behavior Configuration”, mostrada na figura A.5, é aberta.

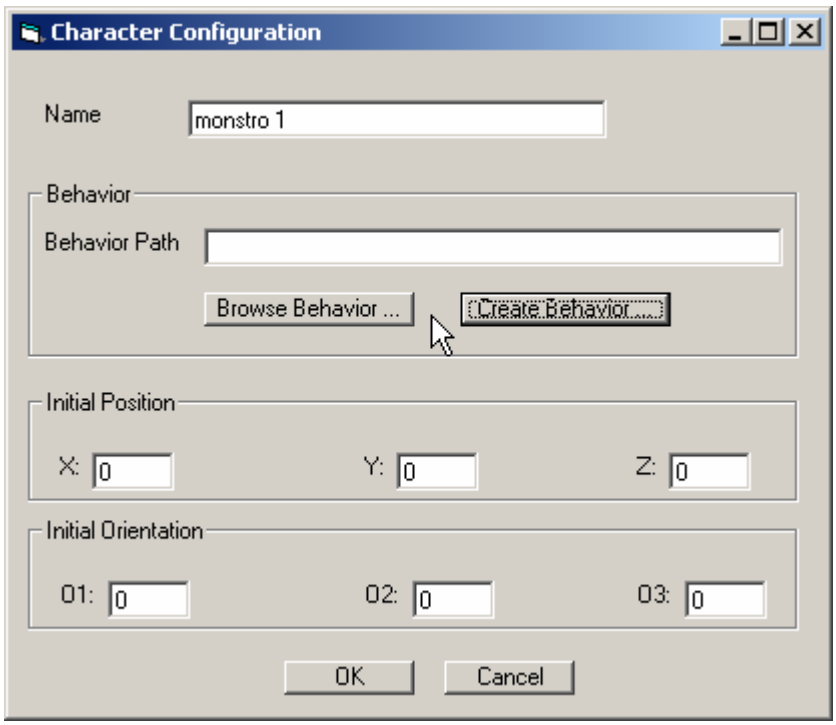

Figura A.4. Janela “Character Configuration”

Na janela “Behavior Configuration” são configurados o modelo e as animações do personagem. As animações são associadas a eventos e são também configuradas as transições entre as seqüências de animação. Essa janela também permite a configuração de “hit points” (pontos de vida do personagem), “points taken by damage” (pontos tirados a cada disparo sofrido pelo personagem).

Ao clicar no botão “Browse Mesh...” o usuário deve escolher um arquivo de modelo no formato md5. O usuário pode visualizar o modelo do personagem clicando em “Show Mesh” ou apenas o esqueleto do personagem clicando em “Show Skeleton”.

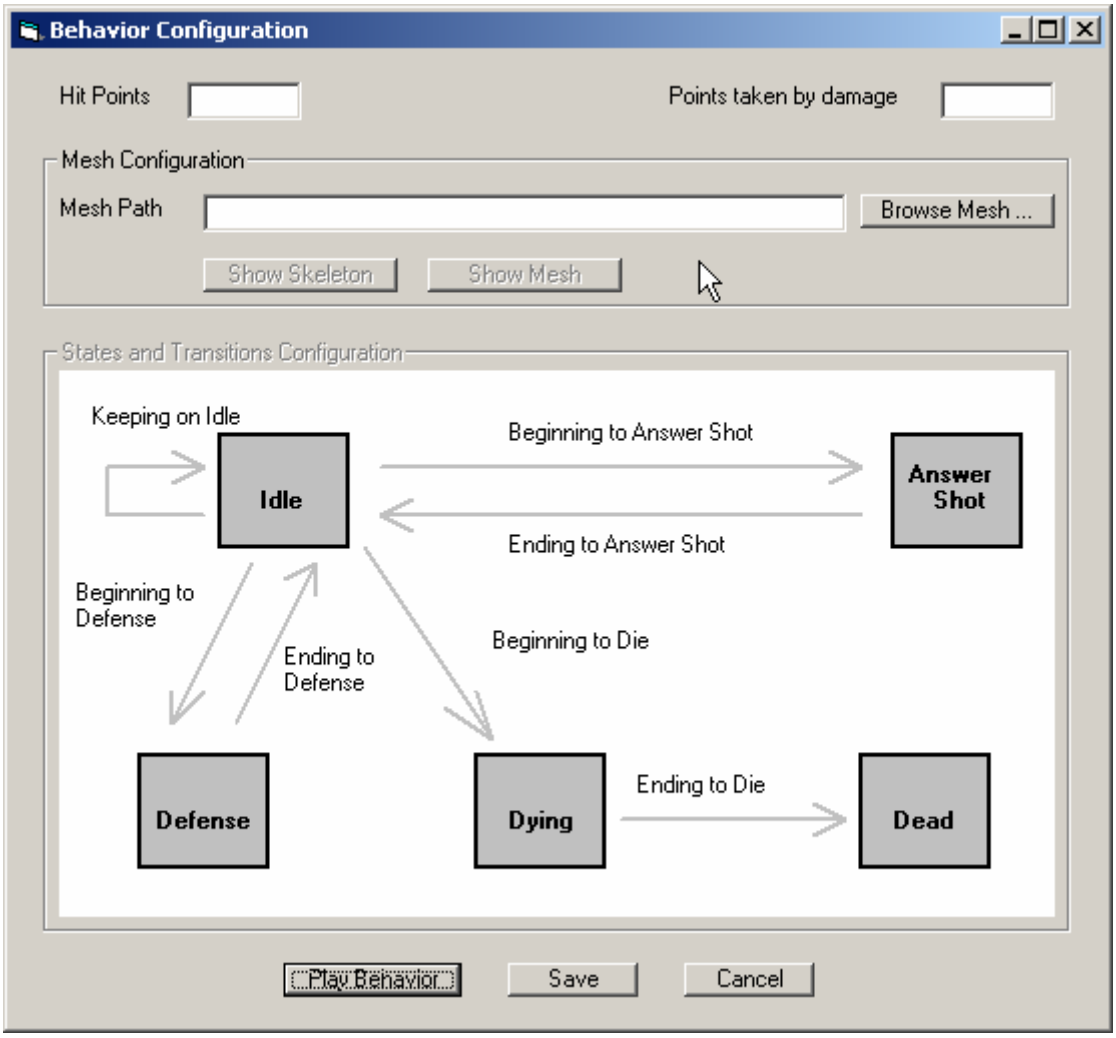

Figura A.5. Tela “Behavior Configuration”

Após escolher o modelo o usuário deve escolher as seqüências de animação para cada estado do personagem e configurar as transições para mudança de estado. Ao dar duplo-clique em um estado a janela “State Configuration”, mostrada na figura A.6, aparece.

A tela “State Configuration” é responsável por associar uma seqüência de animação ao estado escolhido. Ao clicar no botão “Browse Animation...” o usuário deverá escolher uma animação no formado md5. Após ter escolhido a animação, uma tabela será montada para preenchimento do número de *intermediate frames* para cada par de *keyframes* da seqüência de animação. O

valor default “1” é preenchido em cada célula. O usuário pode reproduzir a animação clicando no botão “Play Animation”.

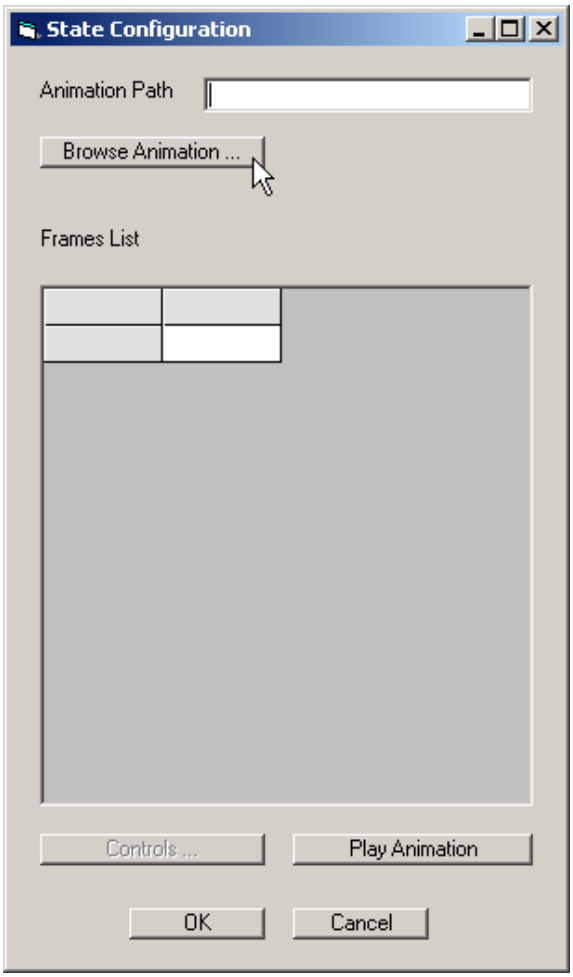

Figura A.6. A janela “State Configuration”

O usuário pode configurar manualmente os números de *intermediate frames* ou pode utilizar o botão “Controls...” para abrir a janela “Animation Controls Configuration”, mostrada na figura A.7. Essa janela permite que sejam configurados os controles de uniformidade, evolução e *timing* da seqüência de animação.

A figura A.7 mostra o controle de uniformidade. As abas permitem alternar entre os três controles. Depois de configurados os três controles (não é obrigatório que se configure qualquer um dos três), pode-se retornar à janela “State Configuration”.

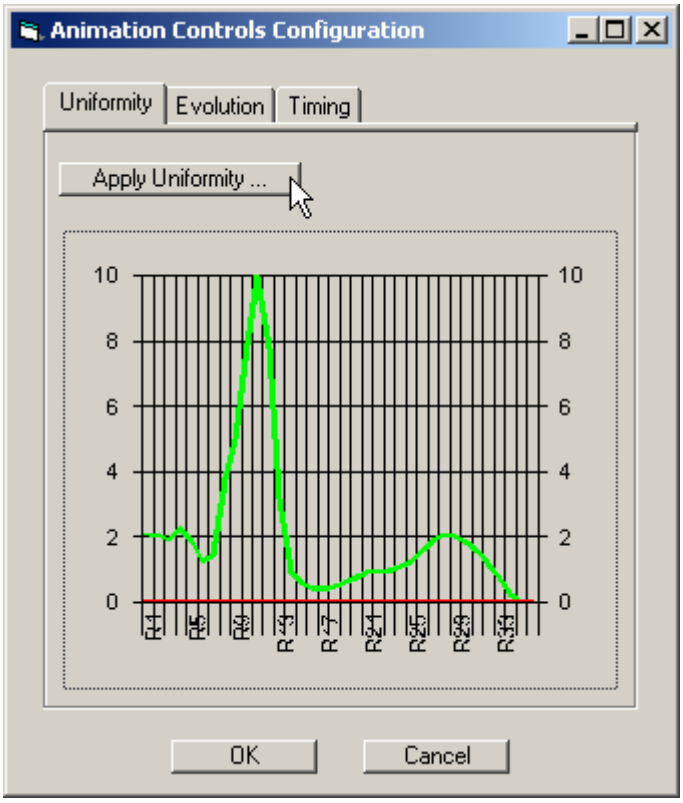

Figura A.7. A janela “Animation Controls Configuration”

Na tela “Behavior Configuration” após ter configurado os dois estados entre os quais está uma determinada transição, o usuário pode configurar essa transição clicando no nome dessa transição e abrindo assim a janela “Transition Configuration”, mostrada na figura A.8.

Na janela “Transition Configuration” o usuário pode configurar as características da transição como “Interpolation Rate” (velocidade da interpolação), “Limit Following Animation” (limitação de repetição da seqüência seguinte) e “Limit Actual Animation” (limitação de repetição da seqüência atual).

Ainda nesta janela, o usuário tem um *feedback* gráfico sobre a transição e pode também visualizar essa transição clicando no botão “Play Transition”. As linhas mostrada na figura A.8 são representações esquemáticas das seqüências atual (pela linha de cor vermelha) e próxima (pela linha de cor azul) e da fase de mesclagem (pela linha de cor branca).

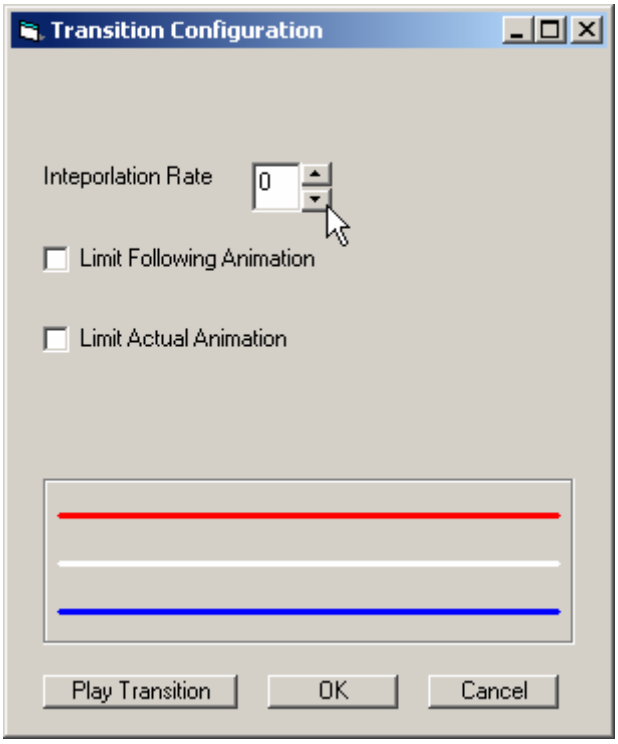

Figura A.8. A janela “Transition Configuration”

A.3 Estágio de Processamento

O processamento das animações pode ser definido pelo desenvolvedor de jogo através da classe “Character” pertencente ao *namespace* “Characters”, como é mostrado na figura A.9.

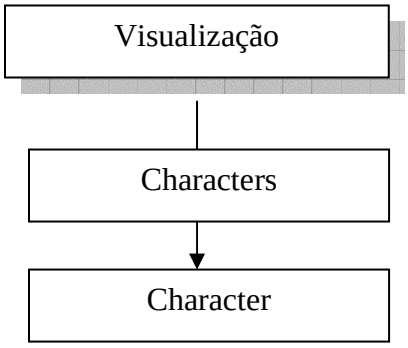

Figura A.9. Classe “Character” subordinada ao *namespace* Character

Essa classe possui os métodos necessários para a leitura dos arquivos de *scripts* e para o processamento do comportamento dos personagens e controle das seqüências de animação em tempo de execução.

O estágio de configuração gera um arquivo chamado “Characters.cnf” que informa quais são os personagens configurados para a aplicação e o caminho do arquivo de *script* para cada um destes personagens.

O arquivo de *script* define quais as seqüências de animação são reproduzidas em para cada estado do personagem. Indica também o número de intermediate frames para cada segmento de cada seqüência de animação. Finalmente indica os caminhos para arquivo de modelo (md5Mesh) e os arquivos de animação (md5Anim). A figura A.10 mostra a relação entre os arquivos de *scripts*.

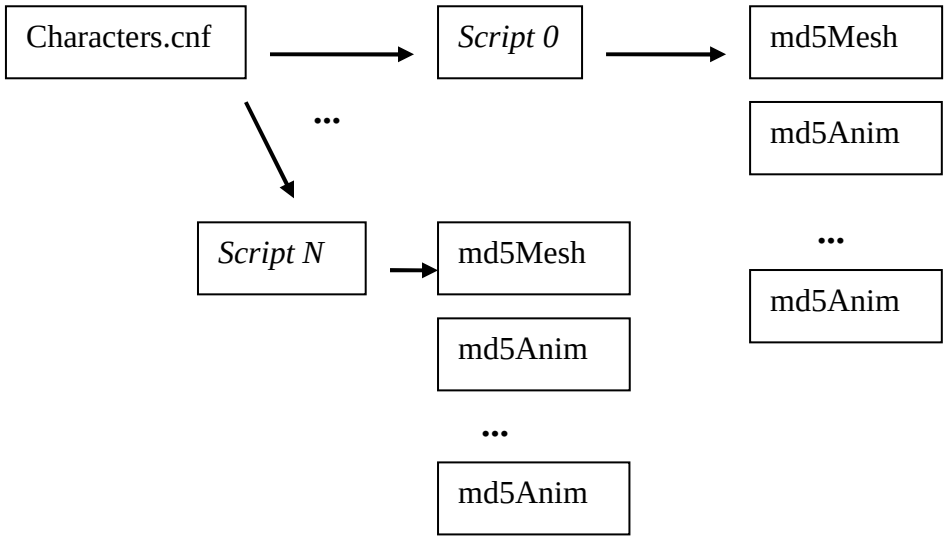

Figura A.10. Relação entre os arquivos de *script* do sistema de controle de animações de personagens do *framework* Guff

Dessa forma, basta que o desenvolvedor de jogo instancie um ou mais objetos a partir da classe “Character”, faça o carregamento do arquivo “Characters.cnf” na inicialização da aplicação e faça a atualização do estado do personagem

(método “UpdateCharacter” pertencente à classe “Character”) na função de atualização de renderização.

Apêndice B

Neste Apêndice serão mostrados de forma mais aprofundada conceitos e definições utilizadas ao longo do texto desta dissertação.

Na seção B.1 serão mostradas algumas curvas paramétricas utilizadas que são comumente utilizadas como funções de interpolação. Essas curvas apresentam visualmente mais suaves do que a interpolação linear.

Na seção B.2 serão mostradas as formas de representação de ângulos e rotações em 3 dimensões.

B.1 Curvas Paramétricas

A definição paramétrica de uma curva em 3 dimensões é dada por

$$Q(u) = (X(u), Y(u), Z(u)) \text{ onde } 0 \leq u \leq 1 \qquad \text{(B.1)}$$

A representação paramétrica é mais interessante para a computação gráfica do que a representação implícita, pois embora ambas sejam formas analíticas, a forma paramétrica permite maior facilidade na descrição de curvas complexas que são na verdade formada por “pedaços” de curvas mais simples.

Uma curva paramétrica é em geral um polinômio de grau (k + 1)ésimo cuja descrição é:

$$Q(u) = p_0 + p_1u + p_2u^2 + ... + p_ku^k \qquad \text{(B.2)}$$

Em computação gráfica os polinômios utilizados são em geral os de grau 3. Isto porque polinômios de grau 2 não apresentam flexibilidade suficiente. Polinômios com grau acima de 3, por outro lado, podem tornar os cálculos complexos e inserir erros indesejáveis. Polinômios de grau 3 (funções cúbicas)

são os polinômios de mais baixa ordem que possuem continuidade de primeira, segunda e terceira ordem.

Curvas complexas podem ser definidas pela junção de curvas mais simples. Para isso, pode ser necessário que as curvas apresentem continuidade na junção. A continuidade de ordem zero (C^0) refere-se ao fato de a junção ser ou não descontínua (apresentar quebra). A continuidade de primeira ordem (C^1) refere-se ao fato de a inclinação ser constante em todos os pontos da curva. A continuidade de segunda ordem (C^2) refere-se ao fato de a curvatura ser a mesma entre as curvas formadoras. Embora as continuidades não sejam obrigatórias elas são desejáveis e o seu estudo se faz caso a caso.

Considerando o conjunto de todos os (k + 1)ésimo polinômios, incluindo os de grau k e de graus inferiores, estes polinômios formam um espaço vetorial. A fórmula da equação (B.2) pode ser vista como a especificação de uma posição neste espaço vetorial com as coordenadas (p_0, p_1, p_2, ..., p_k) e a base (1, u, u^2, ..., u^k).

Os polinômios que formam a base são chamados funções de base e as coordenadas são chamadas de pontos de controle. A equação (B.2) pode ser reescrita como :

$$Q(u) = \sum_{i=0}^{k} p_i b_i(u)$$

E considerando que as funções de base serão de terceira ordem então se pode reescrever como:

$$Q(u) = \sum_{i=0}^{3} p_i b_i(u)$$

A seguir serão mostradas algumas curvas utilizadas em computação gráfica para interpolação: Bezier, Hermite, Catmull-Rom e B-Splines.

B.1.1 Curvas Bézier

As curvas Bézier foram apresentadas pelo engenheiro francês Pierre Bézier no início da década de 1960. Ao mesmo tempo foram desenvolvidas por Paul Casteljau utilizando o chamado algoritmo de Casteljau.

Uma curva Bézier de grau n pode ser generalizada como:

$$Q(u)=\sum_{i=0}^{n} P_i q_{i,n}(u) \qquad , u \in [0, 1]$$

onde o polinômio

$$q_{i,n}(u)=\binom{n}{i} u^i (1-u)^{n-i} \qquad i = 0, \ldots n$$

são conhecidos como funções de base de Bernstein de grau n.

Os pontos P_i são os pontos de controle da curva Bézier. O polígono formado pela conexão dos pontos de controle (começando em P_0 e terminando em P_n) é chamado de polígono de Bézier dentro do qual a curva fica contida.

Em sua forma cúbica a curva Bézier apresenta 4 pontos de controle, conforme mostra a figura B.1 e sua forma paramétrica é:

$$Q(u) = P_0(1 - u)^3 + 3P_1u(1 – u)^2 + 3P_2u^2(1 – u) + P_3u^3 \qquad , u \in [0, 1] \quad \text{(B.3)}$$

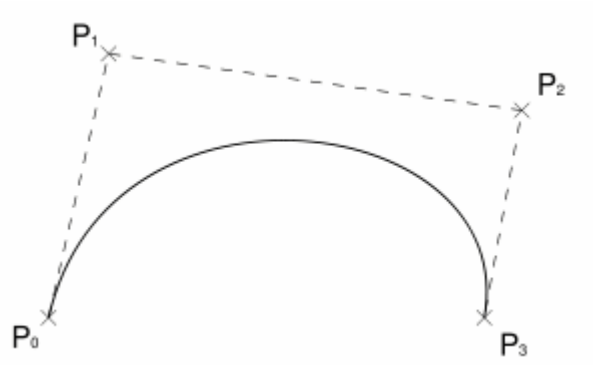

Figura B.1. Bézier de terceira ordem

B.1.2 Curvas de Hermite

As curvas de Hermite são curvas paramétricas cujas funções de base são polinômios de Hermite, nome dado em homenagem ao matemático francês Charles Hermite que os descreveu.

A forma de parametrização para curvas de terceira ordem proposta por Hermite utiliza dois pontos de controle e dois vetores tangentes de controle para cada polinômio. Os módulos dos vetores atuam como pesos na determinação da curva.

Para cada subintervalo, sendo dados um ponto inicial p_0 e um ponto final p_1, um vetor tangente inicial t_0 e um vetor tangente final t_1, o polinômio pode ser definido por:

$$Q(u) = h_0(u)p_0 + h_1(u)p_1 + h_2(u)t_1 + h_3(u)t_2 \qquad t \in [0,1] \qquad \text{(B.4)}$$

onde as funções de base são h_0, h_1, h_2, h_3 :

$h_0 = 2u^3 – 3u^2 + 1;$
$h_1 = -2u^3 + 3u^2;$
$h_2 = u^3 – 2u^2 + u;$
$h_3 = u^3 – u^2$

A figura B.2 mostra uma curva de Hermite.

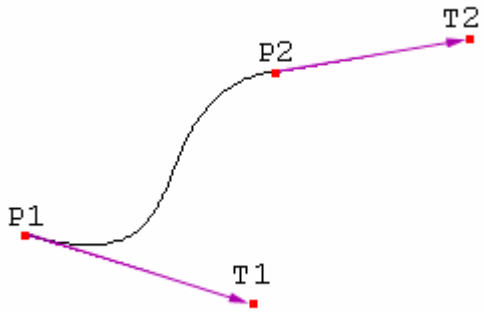

Figura B.2. Curva de Hermite

B.1.3 *Spline* de Catmull-Rom

Uma *Spline* ou *Spline Cúbica Natural* é uma curva baseada em uma expressão matemática que geram curvas com continuidade C^2. Muitas curvas paramétricas são consideradas *splines* por alguns autores embora não o sejam por outros.

As curvas de Hermite, por exemplo, são consideradas *splines* por alguns autores (*Splines* Cúbicas de Hermite) [Advanced Animation and rendering Techniques]. Das curvas de Hermite pode-se obter uma especificação interessante chamada *Cardinal Spline*.

Uma *Cardinal Spline* é um tipo especial de curva de Hermite onde os vetores tangentes são dados por:

$$t_i = ½ (1 – c) (p_{i+1} – p_{i-1}) \qquad \text{(B.5)}$$

onde o t_0 e t_i são dados e c é um fator que modifica o tamanho do vetor tangente (chamado fator de tensão).

Fazendo o fator de tensão igual a zero os vetores tangentes são dados por:

$t_i = ½ \ (p_{i+1} – p_{i-1})$ (B.6)

As curvas passam, então, a ser chamadas de *Splines* de Catmull-Rom. Elas apresentam uma imporante característica: a curva gerada passa pelos pontos de controle. Para calcular um ponto da curva são necessários dois pontos: um antes e outro após, o ponto desejado. O parâmetro t é indica a posição do novo ponto, como mostrado na figura B.3.

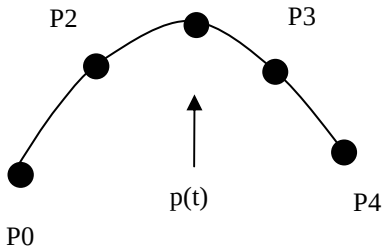

Figura B.3. *Spline* de Catmull-Rom

B.1.4 B-Splines uniformes cúbicas

A função B-spline uniforme cúbica é definida por um conjunto de segmentos onde o i-ésimo segmento é denotado por Qi. Um segmento possui 4 pontos de controle denotados por p_i, p_{i+1}, p_{i+2} e p_{i+3}. O segmento seguinte Q_{i+1} possui também os pontos p_{i+1}, p_{i+2} e p_{i+3} que compartilha com o segmento Q_i, além do ponto p_{1+4}.

Um segmento Q_i é definido como:

$$Q_i(u) = \sum_{k=0}^{3} p_{i+k} B_k(u)$$

(B.7)

Onde u é definido no intervalo 0 <= u <= 1 e as funções de base B(u) dadas por (k = 0, ...,3) são:

$$B_0(u) = \frac{(1+u)^3}{6}$$

$$B_1(u) = \frac{3u^3 - 6u^2 + 4}{6}$$

$$B_2(u) = \frac{-3u^3 + 3u^2 + 3u + 1}{6}$$

$$B_3(u) = \frac{u^3}{6}$$

A B-*spline* uniforme cúbica possui continuidade de ordem zero, primeira e segunda ordens. Ela possui controle local o que significa que uma alteração em um ponto de controle propaga-se apenas para os seu vizinhos mais próximos. Além disso, ao contrário da *Spline* de Catmull-Rom, não passa pelos pontos de controle.

B.2 Parametrização de rotações

Nesta seção será mostrada de forma mais aprofundada 3 formas de parametrização de rotações em computação gráfica: ângulos de Euler, rotação ao redor de um eixo e quatérnios.

B.2.1 Ângulos de Euler

Ângulos de Euler é a forma mais intuitiva de parametrizar rotações em 3 dimensões. Esse nome foi dado em homenagem ao matemático suíço Leonard Euler.

Nesta representação, uma rotação genérica é descrita como uma seqüência de rotações ao redor de três eixos mutuamente ortogonais (eixos coordenados). Os ângulos de rotação são tomados no sentido anti-horário e são às vezes chamado x-roll (giro ao redor de x), y-roll (giro ao redor de y) e z-roll (giro ao redor de z). Os ângulos de Euler podem ser visto na figura B.4.

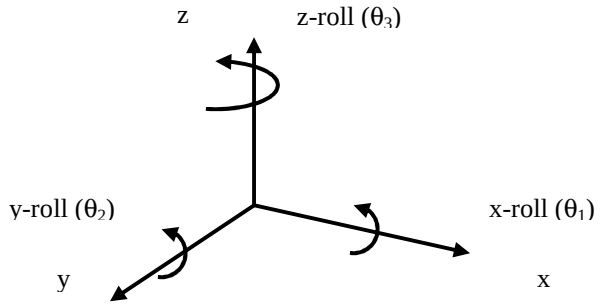

Figura B.4. Ângulos de Euler

A forma matricial da rotação em torno do eixo x é:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_1 & sen\theta_1 & 0 \\ 0 & -sen\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A forma matricial da rotação em torno do eixo y é:

$$\begin{bmatrix} \cos\theta_2 & 0 & -sen\theta_2 & 0 \\ 0 & 1 & 0 & 0 \\ sen\theta_2 & 0 & \cos\theta_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A forma matricial da rotação em torno do eixo z é:

$$\begin{bmatrix} \cos\theta_3 & sen\theta_3 & 0 & 0 \\ -sen\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As operações de rotação não são comutativas, ou dito de outra forma, a ordem em que essas rotações são feitas para determinar a rotação final conta.

Dependendo da ordem em que se realizam as rotações podem-se obter resultados diferentes.

Considerando por exemplo a ordem x, depois y, depois z, a parametrização fica sintetizada em uma única matriz que é a multiplicação das três matrizes de rotação ao redor e cada eixo: R (θ_1, θ_2, θ_3) que resulta em:

$$\begin{bmatrix} \cos\theta_2 \cos\theta_3 & \cos\theta_2 sen\theta_3 & -sen\theta_2 & 0 \\ sen\theta_1 sen\theta_2 \cos\theta_3 - \cos\theta_1 sen\theta_3 & sen\theta_1 sen\theta_2 sen\theta_3 + \cos\theta_1 \cos\theta_3 & sen\theta_1 \cos\theta_2 & 0 \\ \cos\theta_1 sen\theta_2 \cos\theta_3 + sen\theta_1 sen\theta_3 & \cos\theta_1 sen\theta_2 sen\theta_3 - sen\theta_1 \cos\theta_3 & \cos\theta_1 \cos\theta_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Por ser tão intuitiva a representação de rotações por ângulos de Euler é muito utilizada. Mas há dois problemas que fazem com essa solução seja problemática em alguns casos: *gimbal lock* e interpolação de rotações. Ambos ocorrem porque a representação de ângulos de Euler ignora a interação entre os ângulos considerando-os independentes entre si.

O termo *gimbal lock* advém de um mecanismo chamado *gimbal* (balanceiro de giroscópio) que é um mecanismo mecânico formado por três anéis concêntricos, cuja finalidade é simular as rotações em 3 dimensões utilizando como base os ângulos de Euler. O *gimbal lock* ocorre quando a seqüência de rotações é feita de forma que dois dos três anéis ficam alinhados e fica impossível representar rotações em um dos eixos, determinando a perda de um grau de liberdade.

A figura B.5 mostra um conjunto de transformações que resulta em um *gimbal lock*. É feita uma rotação em torno do eixo y de 90^0. Essa rotação afeta o eixo z que cai sobre o eixo x. Agora qualquer rotação em torno do eixo z será na verdade uma rotação em torno do eixo x e não se pode mais realizar rotações em torno do eixo z, ou seja um grau de liberdade foi perdido na representação.

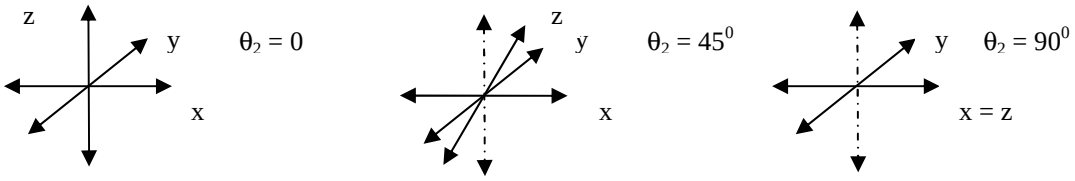

Figura B.5. *Gimbal lock*

A questão da interpolação entre rotações também é um problema na representação de ângulos de Euler. Intuitivamente se pode pensar em tratar as rotações de forma independente e que por analogia à translação, é possível interpolar linearmente cada um dos 3 ângulos (rotações) e depois combiná-los para produzir cada matriz de rotação intermediária durante a interpolação.

O erro neste procedimento está no fato de que rotações não são independentes entre si. Deve-se lembrar que a interpolação linear de translações é representada como uma adição entre matrizes. A interpolação linear seria representada por multiplicações entre matrizes, e multiplicações de matrizes não comutam. Isso significa que dependendo da ordem escolhida para realizar as rotações, resultados intermediários diferentes serão obtidos para um mesmo movimento.

Além disso, a interpolação linear de rotações, gera posições intermediárias que não são naturalmente esperadas causando, efeitos visuais indesejáveis. A figura B.6 mostra em 2D uma rotação de 90^0 graus feita em torno do ponto P. Observe como vetor v sofre uma distorção de tamanho ao longo do movimento. A linha azul indica o caminho que deveria ser gerado e a linha vermelha indicado que foi de fato gerado pela interpolação linear.

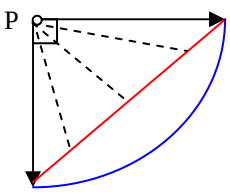

Figura B.6. Distorção causada pela interpolação linear de uma rotação

B.2.2 Axis-Angle

Uma forma de representação para rotações deve ser capaz de representar rotações sem os problemas observados na representação de ângulos de Euler.

O teorema Euler conhecido como *angular displacement* (deslocamento angular) diz que sempre é possível transicionar entre duas orientações com uma rotação simples em torno de um eixo arbitrário (não necessariamente um eixo coordenado).

Baseando-se neste teorema, pode-se definir uma representação para rotações conhecida como Axis-angle (ângulo-eixo) ou ainda Rotation about na arbitrary axis (rotação em torno de um ângulo arbitrário). Essa representação é definida como a notação R(θ, n) onde θ é uma rotação e n é um eixo. Um exemplo pode ser visto na figura B.7 onde o vetor r sofre um deslocamento angular de θ em torno do eixo n.

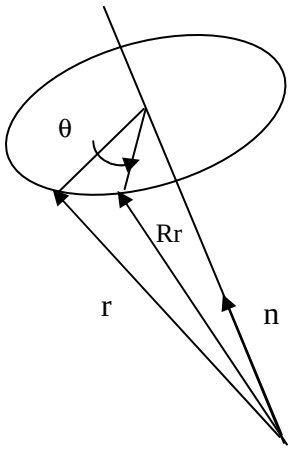

Figura B.7. Deslocamento angular do vetor r de valor θ ao retor no eixo n

O delocamento angular pode ser calculado por:

$$Rr = (\cos\theta)r + (1 - \cos\theta)n(n \bullet r) + (\operatorname{sen}\theta)\, n \times r \qquad \text{(B.8)}$$

A representação *axis-angle* é isenta de *gimbal lock* e permite gerar interpolações naturais entre rotações. Contudo a representação *axis-angle* resulta em expressões extensas e poucos intuitivas quando baseadas em eixos cartesianos e ângulos.

Uma forma de tirar proveito da idéia da representação *axis-angle* com uma representação mais adequada para a computação gráfica e conseqüentemente a área de jogos 3D é a utilização de quatérnios.

B.2.3 Quatérnios

Quatérnios são um conjunto pertencente ao ramo da álgebra chamado "álgebra não-comutativa". O conjunto dos quatérnios é na verdade um caso específico de uma classe genérica dos números **hypercomplexos** descobertos pelo matemático irlandês Sir William Rowan Hamilton em 1834 [Hamilton].

Um quatérnio pode ser definido como se segue:

$$H_q = (q_v, q_w) = (q_x, q_y, q_z, q_w) = iq_x + jq_y + kq_z + q_w,$$

$$i^2 = j^2 = k^2 = -1,\ jk = -kj = i,\ ki = -ik = j,\ ij = -ji = k \qquad \text{(B.9)}$$

O vetor $q_v = (q_x, q_y, q_z)$ é chamado de parte imaginária e o número q_w é chamado de parte real do quatérnio H_q. Os números i, j e k são chamados unidades imaginárias[moller].

Seguem algumas definições importantes de quatérnios:

Adição: $H_q + H_r = (q_v + r_v, q_w + r_w)$

Conjugado: $H_q^* = (-q_v, q_w)$

Norma: $n(H_q) = q^2_x + q^2_y + q^2_z + q^2_w$

Identidade: $I = (0, 1)$

Inverso: $H_q^{-1} = H_q^* / n(H_q)$

Multiplicação: $H_q H_r = (q_v \times r_v + r_w q_v + q_w r_v, q_w r_w - q_v \bullet r_v)$

Convém notar que a expressão de multiplicação de quatérnios é não comutativa (observar o fator de produto interno). Além disso, quatérnios forma um grupo já que a multiplicação de dois quatérnios gera um quatérnio como resultado.

Um subgrupo dos quatérnios, os quatérnios unitários, guarda estreita similaridade com o grupo das matrizes de rotação e por esse motivo é capaz de representar qualquer rotação em 3 dimensões, com a vantagem de ser uma forma de representação muito compacta [moller].

Um quatérnio unitário é um quatérnio cuja norma é igual a 1. A partir dessa definição pode-se escrever um quatérnio unitário na forma:

$$H_q = (\text{sen}\theta n, \cos\theta), \qquad |n| = 1 \qquad \text{(B.10)}$$

Sendo n um vetor tridimensional cuja norma é igual a 1. Além disso, vale lembrar que se o quatérnio é unitário então $H_q H_q^* = 1$ e logo $H_q^* = H_q^{-1}$. Representando um ponto P (p_x, p_y, p_z) como um quatérnio sem parte real: $H_p = (p_v, 0)$, e realizando a multiplicação : $R_q(p) = H_q H_p H_q^*$, onde H_p é um quatérnio unitário, resulta:

$$R_q(p) = (0, (\cos 2\theta)p_v + (1 - \cos 2\theta)(n \bullet p_v)n + (\text{sen}2\theta)n \times p_v) \qquad \text{(B.11)}$$

Comparando as equações B.8 e B.11 pode-se concluir que o deslocamento angular (θ,n) do ponto P pode ser representado levando o ponto

para o espaço de quatérnios, representando o deslocamento angular por um quatérnio unitário (sen(θ/2)n, cos(θ/2)) e realizando a operação $H_q H_p H_q^*$. Embora esse processo pareça custoso a princípio, na verdade ele é muito adequado, sobretudo porque o espaço de quatérnios permite a combinação de rotações de forma automática.

Supondo a combinação de duas rotações (θ_1 e θ_2), ter-se-ia a príncipio a seguinte operação $R_{Hr}(R_{Hq}(P)) = R_{Hs}(P)$. Baseando-se nas propriedades dos quatérnios unitários chega-se a: $H_s = H_r H_q$. Ou seja, a composição de duas rotações é a multiplicação dos quatérnios unitários que representam tais rotações.

A interpolação entre rotações no espaço de quatérnios pode ser realizada utilizando uma operação chamada *Spherical Linear Interpolation* (*Slerp*). Essa função possibilita encontrar uma posição intermediária definida pelo parâmetro t ($t \in [0,1]$), entre duas orientações representadas por dois quatérnios unitários (H_q e H_r).

Na prática o que a *Slerp* calcula são os quatérnios unitários interpolados que formam o menor arco em uma esfera unitária de 4 dimensões (que é o espaço geométrico formado pelos quatérnios unitários) que vai de H_q (t = 0) até H_r (t = 1).

O cálculo realizado pela *Slerp* é:

$$\text{slerp}(H_q, H_r, t) = \frac{sen(\theta(1-t))}{sen\theta} H_q + \frac{sen(\theta t)}{sen\theta} H_r \qquad \text{(B.12)}$$

A representação de deslocamento de ângulos como quatérnios permite realizar a parametrização das rotações sem que se incorra no problema de *gimbal lock*, com interpolação de rotações sem distorções (com o uso de *Slerp*) e com a praticidade das operações no espaço de quatérnios.

Dessa forma, dada uma animação baseada em rotações, um *keyframe* de animação pode ser representado por uma matriz de rotação. A seqüência de matrizes de rotação será convertida para uma seqüência de quatérnios. Os quatérnios interpolados são calculados e convertidos novamente para a representação matricial.

A conversão de um quatérnio unitário para uma matrix é feita na forma:

$$M^{Hq} = \begin{bmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y-q_wq_z) & 2(q_xq_z+q_wq_y) & 0 \\ 2(q_xq_y+q_wq_z) & 1-2(q_x^2+q_z^2) & 2(q_yq_z+q_wq_x) & 0 \\ 2(q_xq_z-q_wq_y) & 2(q_yq_z+q_wq_x) & 1-2(q_x^2+q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(B.13)}$$

Para se converter uma matriz genérica M^{Hq} mostrada na equação B.14, em um quatérnio H_q, deve-se resolver as equações mostradas em B.15 e com o maior valor obtido (q_x, q_y, q_z ou q_w) calcular os outros três valores através das equações mostradas em B.16.

$$M^{Hq} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \quad \text{(B.14)}$$

$$4q^2_x = + m_{00} + m_{11} + m_{22} + m_{33}$$
$$4q^2_y = - m_{00} + m_{11} - m_{22} + m_{33}$$
$$4q^2_z = - m_{00} - m_{11} + m_{22} + m_{33}$$
$$4q^2_x = \text{traço } (M^{Hq}) \quad \text{(B.15)}$$

$$m_{21} + m_{12} = 4q_wq_x$$
$$m_{20} + m_{02} = 4q_wq_y$$
$$m_{10} + m_{01} = 4q_wq_z \quad \text{(B.16)}$$