Games 3D: Aspectos de Desenvolvimento

Marcos Fernandez Cuzziol

Dissertação de Mestrado produzida como exigência parcial para a obtenção do título de Mestre no curso de Mestrado em Artes, área de concentração Artes Plásticas, apresentada à Escola de Comunicações e Artes da Universidade de São Paulo.

Orientador:

Prof. Dr. Gilberto dos Santos Prado

Universidade de São Paulo
Escola de Comunicações e Artes
Departamento de Artes Plásticas

Games 3D: Aspectos de Desenvolvimento

Marcos Fernandez Cuzziol

Dissertação de Mestrado produzida como exigência parcial para a obtenção do título de Mestre no curso de Mestrado em Artes, área de concentração Artes Plásticas, apresentada à Escola de Comunicações e Artes da Universidade de São Paulo.

Orientador:
Prof. Dr. Gilberto dos Santos Prado

São Paulo
2007

COMISSÃO JULGADORA

Titulares:

(Nome)	(Assinatura)
(Nome)	(Assinatura)

Orientador:

Prof. Dr. Gilberto dos Santos Prado (Nome)	(Assinatura)

RESUMO

Esta dissertação documenta a exploração do desenvolvimento de games 3D com base em minha experiência pessoal nessa atividade, entre 1994 e 2006, estabelecendo um diálogo com os trabalhos teóricos de autores de áreas correlatas. São descrições sistematizadas de técnicas e conceitos em grande parte ignorados em publicações atuais que tratam desse tema, e que decorrem de experimentação e de descobertas efetuadas durante o processo de criação e desenvolvimento de games em meio digital.

PALAVRAS-CHAVE: GAMES, JOGOS ELETRÔNICOS, DESENVOLVIMENTO DE GAMES, REALIDADE VIRTUAL, AMBIENTES 3D, MEIO DIGITAL.

ABSTRACT

This dissertation documents the 3D game development exploration based on my personal experience in this activity, between 1994 and 2006, establishing a dialog with works of authors from correlated areas. It comprises systemized descriptions on techniques and concepts mainly overlooked in current publications devoted to the theme, and built on experimentation and discoveries achieved during the process of creation and development of games in the digital medium.

KEYWORDS: GAMES, GAME DEVELOPMENT, VIRTUAL REALITY, 3D ENVIRONMENTS, DIGITAL MEDIUM.

SUMÁRIO

1. INTRODUÇÃO

Em abril de 1996, iniciei, com Odair Gaspar, o desenvolvimento do que viria a ser o primeiro game brasileiro de ação em primeira pessoa, *Incidente em Varginha*.

Minha pesquisa prática sobre games, entretanto, começou dois anos antes com a programação de um raycaster[1], no estilo *Doom* (Id Software, 1993). Encontrar literatura sobre o desenvolvimento de games, na época, não era tarefa simples. Afinal, o primeiro game 3D de sucesso baseado em raycaster, *Wolfenstein 3D* (Id Software, 1992), havia sido publicado menos de dois anos antes. O material documental disponível era escasso e fortemente voltado à programação nas linguagens C e Assembly. Acompanhando o periódico *Dr. Dobb's Journal*, dirigido a programadores e publicado mensalmente desde 1976, era possível encontrar artigos de autoria de Michael Abrash sobre programação gráfica para PC. Grande parte desse material foi revisado para publicação em livro do mesmo autor, *Zen of Graphics Programming*, em 1994. No mesmo ano, foram publicados outros dois livros referenciais sobre programação de games 3D: *Gardens of Imagination*, de Lampton, e *Tricks of the Game Programming Gurus*, de LaMothe, Ratcliff, Seminatore e Tyler. Ambos descreviam o funcionamento e a programação necessária para games baseados no algoritmo de raycaster.

Lembro-me da alegria de encontrar esses títulos. Muitos dos problemas técnicos que enfrentei na tentativa de desenvolver meu próprio raycaster foram solucionados, de forma simples e elegante, com as informações disponíveis nessas publicações. Os avanços foram mais significativos na programação em linguagem Assembly, vital para a velocidade do programa e, portanto, para permitir seu funcionamento em tempo real. Mas, enquanto meu raycaster progredia a um nível profissional, algo se tornava cada vez mais claro: o desenvolvimento do *engine*[2] (no caso, o próprio raycaster) era apenas uma das etapas do trabalho. Ainda faltaria todo o desenvolvimento de conteúdo para completar a produção de um game 3D. Pior, com a velocidade da evolução do hardware, o tempo exigido para a criação do conteúdo, estimado entre um e dois anos, seria mais

[1] Raycaster é um algoritmo que permite a construção de imagens de aparência tridimensional a partir de um mapa bidimensional. Trata-se de uma simplificação de outro algoritmo, o raytracer, utilizado ainda hoje por diversos programas de síntese de imagens. Numa época em que os PCs não dispunham da tecnologia de placas gráficas aceleradas, o raycaster era uma das poucas soluções para a criação de ambientes tridimensionais em tempo real nessa plataforma.

que suficiente para tornar obsoleto o programa original do game. Desenvolver o programa ou o conteúdo? Com uma equipe de apenas duas pessoas, não seria possível executar as duas tarefas simultaneamente.

Era preciso escolher, mas, na época, não havia escolha possível. De que adiantaria desenvolver um programa profissional sem conteúdo? Ou um excelente conteúdo sem programa para exibi-lo? Assim, o trabalho no raycaster continuou como prioridade. Entretanto, em 1996, começaram a aparecer os primeiros engines comerciais para PC, embora sob *royalties* relativamente elevados. Após todo o trabalho de desenvolvimento investido num raycaster próprio, a decisão de abandoná-lo para adquirir um produto de mercado foi extremamente difícil. Mas era preciso iniciar o desenvolvimento do conteúdo do game, então já com tema definido: a suposta aparição de extraterrestres na cidade mineira de Varginha. O engine escolhido foi o *Acknex A3*, da Conitec Datensysteme (hoje conhecido como *3D Game Studio*). Com uma comunidade de usuários relativamente grande, o *Acknex A3* era constantemente atualizado, fato que reduzia o risco de obsolescência do produto final.

Mas a programação de um raycaster próprio, ainda que não aproveitada diretamente, rendeu bons frutos durante o desenvolvimento de *Incidente em Varginha*. O domínio dos detalhes funcionais do engine revelou-se importantíssimo na geração do conteúdo, pois evitou muitos erros comuns aos desenvolvedores iniciantes. Especificamente, o trabalho com um raycaster próprio ensinou-me, desde o início, a respeitar as características do meio, como a resolução de texturas e sprites[3] ou os limites de processamento e memória. Muitos primeiros projetos de games fracassam justamente pela falta de conhecimento dessas características do meio digital.

Desde o lançamento de *Incidente em Varginha*, tenho trabalhado em diversos títulos, alguns dos quais publicados no Brasil e no exterior. Um demo da seqüência *IV2: Sombras da Verdade* foi selecionado pela Intel para exibição no evento "Pentium III Preview Day", em San Jose, Califórnia (EUA), em 1999[4]. *Beach Volleyball* foi encomendado por um publisher europeu em 2000 e chegou ao estágio de demo. *Micro Scooter Challenge*, de 2001, foi publicado em 14 países e chegou a ser um dos games mais vendidos na Alemanha. *Super Mini Racing*, desenvolvido com a Canal Kids, foi

[2] Programa responsável, principalmente, pela geração de imagens do game (ver glossário).
[3] Sprites são figuras criadas em mapa de bits. Normalmente representam personagens ou objetos nos ambientes virtuais que se utilizam do algoritmo raycaster ou em games 2D.
[4] O demo de *IV2* foi otimizado por mim para o processador Pentium III dentro do Programa de Desenvolvedores Intel.

uma das primeiras experiências bem sucedidas com advergames 3D no Brasil, com mais de 500 mil cópias distribuídas gratuitamente nos jornais *O Estado de S. Paulo* e *Jornal da Tarde*, em 12 de outubro de 2001. *Iracema Aventura*, produzido em 2005, foi vencedor do concurso JogosBR, do ministério da Cultura. Em 2006, a seqüência *Super Mini Racing Ice* foi produzida em conjunto com a Canal Kids e está disponível para download gratuito. Além dos trabalhos na Perceptum Software Ltda., outros games produzidos no Instituto Itaú Cultural, sob minha coordenação técnica, foram *Imateriais*, de 1999; *Paulista 1919*, de 2003; e *Basílica de São Bento*, ainda inédito.

Durante todos esses projetos de games, mas de modo especial desde meu ingresso no curso de mestrado em artes da ECA/USP, em 2004, procurei embasar o trabalho prático do desenvolvimento de conteúdo no conhecimento teórico disponível. Muito se publicou sobre o desenvolvimento de games desde meados da década de 90, e o foco das publicações voltou-se para o desenvolvimento do conteúdo, em claro contraste aos primeiros livros, dedicados mais especificamente à programação de games. O material disponível atualmente é amplo, cobrindo áreas diversas, como modelagem 3D, texturas, animações, criação de personagens, etc. Hoje, um desenvolvedor pode adquirir um engine poderoso e livre de *royalties*, como o *Torque Game Engine*, da GarageGames, por US$ 100,00. Assim, trabalhar na programação de um engine vem se tornando tarefa extremamente específica, cada vez mais distante do próprio desenvolvimento de conteúdo. É apenas natural que o foco das publicações tenha mudado tanto.

Entretanto, por imprescindíveis que sejam as publicações atuais sobre desenvolvimento de games, existem vazios – e mesmo erros – derivados, a meu ver, do distanciamento crescente entre programação e conteúdo.

Esta dissertação trata de alguns desses vazios, de detalhes que ainda não se encontram em publicações sobre games. É uma tentativa de documentar descobertas feitas durante o desenvolvimento dos games supracitados, embasando-as, sempre que viável, no trabalho de autores de outras áreas. São descrições de técnicas e conceitos que podem parecer, à primeira vista, contra-intuitivos, mas que decorrem de experimentação e de características do meio digital.

Por essas mesmas razões, este trabalho não poderia, de forma alguma, abordar o tema de forma exaustiva. Não se pretende, aqui, descrever todos os aspectos do desenvolvimento de games. Para esse fim existe amplo material disponível, e vários exemplos são citados nas referências bibliográficas ao final deste volume.

Mapa de Desenvolvimento de Games 3D
Perceptum Software Ltda.
Concepção
detalhamento
Design Doc
Documento que descreve o game em detalhes. Equivalente ao roteiro de uma produção cinematográfica
aprovação do projeto
Reuniões com Patrocinador ou Publisher
aprovado?
define
Fichas de Personagens
Concepções de Ambientes
Efeitos Sonoros
Músicas
Interface
Fluxo de Jogo
Comportamento Artificial
Produção
modelador
texturizador
designer de ambientes
designer de som
designer gráfico
programador
animador
Modelagem 3D de Personagens
texturas
Modelagem 3D de Ambientes
Sons dos Personagens
Sons Ambientes
Trilhas
Interface
Programação de interface
Comportamento Artificial
Ajustes e Otimização do Engine
Animação
correções
Integração
Level Designer
Prototipagem
testadores
Relatório de Avaliação
Produto Final

Diagrama 1

O Diagrama 1 esquematiza o processo de desenvolvimento de games 3D utilizado na Perceptum. Das diversas áreas envolvidas, esta dissertação aborda apenas quatro: modelagem 3D, texturização, animação e comportamento artificial.

Um dos desafios enfrentados neste trabalho foi a escolha da linguagem de apresentação. Os conceitos descritos exigem o emprego de linguajar técnico, além de pré-requisitos, por parte do leitor, em modelagem 3D, texturização, animação e programação. Mas isso restringiria significativamente o público potencial, principalmente se levarmos em conta a área de concentração dessa dissertação de mestrado. Procurei, então, apresentar técnicas e conceitos de forma direta, tentando transmiti-los de maneira clara, sempre que possível, mesmo para eventuais leitores sem conhecimento prévio das áreas abordadas. Nos exemplos, evitei ao máximo apresentar softwares aplicativos ou compiladores específicos, por dois motivos: além de dificultar a compreensão daqueles que não são versados nesses softwares, essa prática tende a envelhecer qualquer trabalho muito rapidamente, pois novas versões de aplicativos e compiladores são constantemente publicadas. Principalmente nos primeiros capítulos, procurei ilustrar conceitos e técnicas com imagens geradas a partir de material extraído de games de minha autoria.

No capítulo 1, apresentam-se alguns conceitos sobre modelos 3D, incluindo exemplos de suas representações numéricas e outras características pouco exploradas em publicações sobre o tema. São apresentadas também técnicas que demonstram um fato aparentemente contra-intuitivo: um modelo pode parecer mais convincente com um número de polígonos reduzido do que com uma contagem de polígonos alta. Analiso brevemente o porquê desses efeitos com base em simples geometria e com o auxílio de ilustrações. Detalhes da construção de ambientes e interiores também são apresentados, juntamente com alguns erros corriqueiros nesse tipo de modelagem e suas soluções.

O capítulo 2 demonstra o uso de texturas como complemento do visual 3D de um modelo. Certos detalhes volumétricos, além de iluminação e sombra, podem ser eficientemente representados por texturas, sem a necessidade de polígonos adicionais. Analiso também um detalhe curioso: texturas de alta resolução não significam, necessariamente, imagens finais de boa qualidade. Pelo contrário, o uso de texturas de grandes dimensões pode comprometer seriamente, não apenas a demanda de memória e processamento do sistema, mas a própria qualidade final da imagem. Técnicas de mapeamento são apresentadas ao final do capítulo, juntamente com uma reflexão sobre suas principais características.

Animação é o tema do capítulo 3. São apresentados indícios de que movimentos são mais críticos que forma, pelo menos em personagens virtuais 3D. Analisa-se também a maior tolerância de personagens menos realistas a animações pouco convencionais. Em seguida, veremos as limitações da técnica de captura de movimentos e, principalmente, como certas animações "impossíveis" podem, paradoxalmente, contribuir para a sensação de realidade sobre um personagem virtual.

No capítulo 4, um aspecto aparentemente contraditório da programação é apresentado: a criação de comportamentos artificiais potencialmente imprevisíveis e emergentes a partir de seqüências de instruções rígidas. Exemplos práticos da criação de comportamentos emergentes são citados. Dada a imprevisibilidade intrínseca dos comportamentos emergentes, tais técnicas dependem normalmente de tentativa e erro, e é comum a emergência de comportamentos não desejáveis durante o desenvolvimento de um personagem. Apresento uma solução possível para a evolução de comportamentos emergentes desejáveis, baseada em algoritmos genéticos, testada com relativo sucesso no game *Incidente em Varginha*.

Nas considerações finais, apresento alguns possíveis desdobramentos do que foi visto nos capítulos anteriores. Em especial, serão discutidas possíveis aplicações de design evolutivo nas áreas de modelagem 3D e animação, o conceito de autonomia de personagens virtuais e a simulação de emoções.

No anexo 1, relaciono em glossário os termos técnicos utilizados neste trabalho. O anexo 2 traz, a título de referência, a definição completa de um dos personagens de *Incidente em Varginha*, seus sprites e sua máquina de estados finitos, escrita em linguagem C-Script. Finalmente, no anexo 3, um breve histórico da produção brasileira de games 3D é apresentado.

2. MODELOS 3D

Modelos 3D são a base dos ambientes virtuais poligonais. São construídos com polígonos, sobre os quais se aplicam texturas, imagens que conferem o visual das superfícies do modelo. Via de regra, a unidade de construção de um modelo 3D é o polígono mais simples, o triângulo.

Quando pensamos em modelos 3D, é natural que os visualizemos como imagens gráficas, como o exemplo do cubo da Figura 1. Programas de modelagem 3D apresentam modelos dessa forma e permitem que os criemos e editemos da mesma maneira, visualmente. Por isso é bastante comum (e até desejável) que modelos sejam confundidos com as imagens que deles resultam.

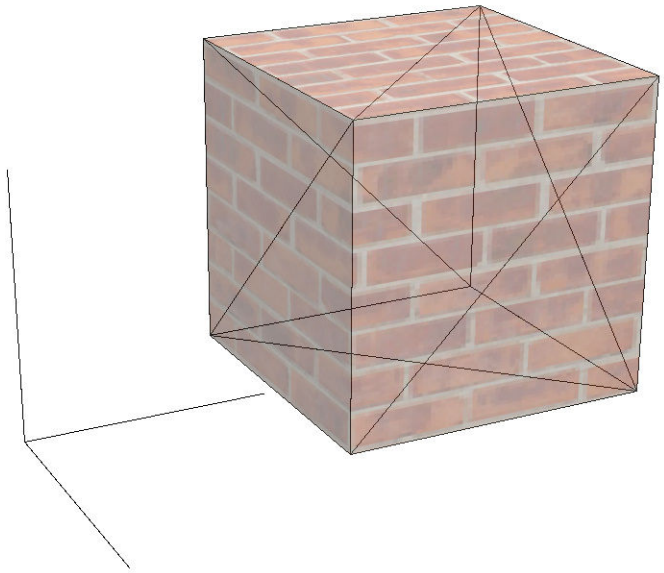


Figura 1

O mesmo ocorre com o texto, por exemplo. Os caracteres lidos nessa página, sua formatação, aparência gráfica e distribuição relativa na folha de papel, foram processados e armazenados em arquivo digital como meras abstrações numéricas, convenções que não guardam semelhança com o objeto representado. Se a digitação da letra "A" gera uma seqüência de bits equivalente ao número decimal 65, 97, ou a

qualquer outro valor de 8 ou 16 bits, pouca diferença faz – desde que esse valor possa ser convertido novamente para a imagem da letra “A”. Para que essas informações numéricas devolvam aquilo que se convencionou representarem, faz-se necessário processamento adicional em interface de saída.

Voltando ao exemplo da Figura 1, um simples cubo também é processado e armazenado digitalmente como massa de valores numéricos. Para propiciar uma idéia da complexidade dessa representação, convém analisar a definição de um objeto 3D em arquivo digital. O formato escolhido, Microsoft® DirectX ASCII, apresenta valores numéricos que definem um modelo 3D em estrutura relativamente legível. Na listagem abaixo temos a definição do cubo da Figura 1 em formato DirectX. Notas explicativas foram incluídas após as barras duplas, no lado direito da listagem:

```
xof 0302txt 0064
Header {                                                    // cabeçalho com número de objetos e materiais
          1;
          0;
          1;
}
Material CuboMaterial_0_0 {                                 // definição de propriedades do material 0
          0.650980; 0.298039; 0.196078; 1.000000;;
          0.000000;
          0.100000; 0.100000; 0.100000;;
          0.000000; 0.000000; 0.000000;;
          TextureFilename {
                    "tijolos02.jpg";                        // arquivo de imagem da textura
          }
}
Mesh CuboMesh {                                             // definição do objeto
          24;                                               // número de vértices
          -1.000000;1.000000;-1.000000;,                    // coordenadas do vértice  0
          -1.000000;3.000000;-1.000000;,                    // coordenadas do vértice  1
          -3.000000;1.000000;-1.000000;,                    // coordenadas do vértice  2
          -3.000000;3.000000;-1.000000;,                    // coordenadas do vértice  3
          -1.000000;1.000000;-3.000000;,                    // coordenadas do vértice  4
          -3.000000;1.000000;-3.000000;,                    // coordenadas do vértice  5
          -3.000000;3.000000;-3.000000;,                    // coordenadas do vértice  6
          -1.000000;3.000000;-3.000000;,                    // coordenadas do vértice  7
          -1.000000;1.000000;-1.000000;,                    // coordenadas do vértice  8
          -3.000000;1.000000;-1.000000;,                    // coordenadas do vértice  9
          -3.000000;1.000000;-3.000000;,                    // coordenadas do vértice 10
          -3.000000;1.000000;-1.000000;,                    // coordenadas do vértice 11
          -3.000000;3.000000;-1.000000;,                    // coordenadas do vértice 12
          -3.000000;3.000000;-3.000000;,                    // coordenadas do vértice 13
          -3.000000;3.000000;-1.000000;,                    // coordenadas do vértice 14
          -1.000000;3.000000;-1.000000;,                    // coordenadas do vértice 15
          -1.000000;3.000000;-3.000000;,                    // coordenadas do vértice 16
          -1.000000;1.000000;-3.000000;,                    // coordenadas do vértice 17
          -3.000000;1.000000;-3.000000;,                    // coordenadas do vértice 18
          -3.000000;3.000000;-3.000000;,                    // coordenadas do vértice 19
          -1.000000;3.000000;-1.000000;,                    // coordenadas do vértice 20
          -1.000000;1.000000;-1.000000;,                    // coordenadas do vértice 21
          -1.000000;1.000000;-3.000000;,                    // coordenadas do vértice 22
          -1.000000;3.000000;-3.000000;;                    // coordenadas do vértice 23
          12;                                               // número de faces
          3;3,2,0;,                                         // três vértices da face 0
          3;3,0,1;,                                         // três vértices da face 1
          3;5,4,8;,                                         // três vértices da face 2
```

```
        3;5,8,9;,                               // três vértices da face 3
        3;6,10,11;,                             // três vértices da face 4
        3;6,11,12;,                             // três vértices da face 5
        3;7,13,14;,                             // três vértices da face 6
        3;7,14,15;,                             // três vértices da face 7
        3;16,17,18;,                            // três vértices da face 8
        3;16,18,19;,                            // três vértices da face 9
        3;20,21,22;,                            // três vértices da face 10
        3;20,22,23;;                            // três vértices da face 11
        }
        MeshTextureCoords {                     // coordenadas de textura
                24;                             // número de vértices texturizados
                0.000000;-0.333333;,            // coordenadas de textura vértice 0
                0.000000;-0.666667;,            // coordenadas de textura vértice 1
                0.250000;-0.333333;,            // coordenadas de textura vértice 2
                0.250000;-0.666667;,            // coordenadas de textura vértice 3
                0.500000;0.000000;,             // coordenadas de textura vértice 4
                0.500000;-0.333333;,            // coordenadas de textura vértice 5
                0.500000;-0.666667;,            // coordenadas de textura vértice 6
                0.500000;-1.000000;,            // coordenadas de textura vértice 7
                0.250000;0.000000;,             // coordenadas de textura vértice 8
                0.250000;-0.333333;,            // coordenadas de textura vértice 9
                0.500000;-0.333333;,            // coordenadas de textura vértice 10
                0.250000;-0.333333;,            // coordenadas de textura vértice 11
                0.250000;-0.666667;,            // coordenadas de textura vértice 12
                0.500000;-0.666667;,            // coordenadas de textura vértice 13
                0.250000;-0.666667;,            // coordenadas de textura vértice 14
                0.250000;-1.000000;,            // coordenadas de textura vértice 15
                0.750000;-0.666667;,            // coordenadas de textura vértice 16
                0.750000;-0.333333;,            // coordenadas de textura vértice 17
                0.500000;-0.333333;,            // coordenadas de textura vértice 18
                0.500000;-0.666667;,            // coordenadas de textura vértice 19
                1.000000;-0.666667;,            // coordenadas de textura vértice 20
                1.000000;-0.333333;,            // coordenadas de textura vértice 21
                0.750000;-0.333333;,            // coordenadas de textura vértice 22
                0.750000;-0.666667;;            // coordenadas de textura vértice 23
        }
        MeshMaterialList {                      // lista de materiais do objeto
                1;                              // número de sub-objetos
                12;                             // número de faces
                0,                              // material da face 0
                0,                              // material da face 1
                0,                              // material da face 2
                0,                              // material da face 3
                0,                              // material da face 4
                0,                              // material da face 5
                0,                              // material da face 6
                0,                              // material da face 7
                0,                              // material da face 8
                0,                              // material da face 9
                0,                              // material da face 10
                0;;                             // material da face 11
                {CuboMaterial_0_0}
        }
}
```

O cubo é definido com 24 vértices, ao contrário dos 8 esperados, porque a cada triângulo podem estar associados material e mapeamento de textura próprios. São então necessários vértices diferentes para triângulos adjacentes, pois as coordenadas de textura podem ser diversas.

É com esse conjunto de valores que um processador deve lidar ao manipular um simples cubo no espaço virtual. Para ser exato, o conjunto de valores será

significativamente maior. Não estão incluídas na definição do modelo as seqüências numéricas que representam sua textura (no caso, armazenadas no arquivo externo "tijolos02.jpg"), os valores e vetores de iluminação, as texturas de sombreamento nem o mapeamento de reflexos ou de normais. Todos esses valores devem ser gerenciados pelo processador em tempo real. Cada coordenada deve passar por transformações matriciais e ser projetada segundo a posição atual da câmera. Cada pixel da textura deve ser recalculado de acordo com as posições das luzes dinâmicas, do efeito de fog, dos mapeamentos de sombras e, caso aplicável, de vetores normais. Enfim, a imagem resultante deve ser recalculada dezenas de vezes por segundo para permitir a interação com os modelos digitais em ambiente virtual.

Mesmo para os processadores mais rápidos, o número limite de triângulos por segundo ainda é um fator característico e restritivo na criação de modelos para games 3D. É certo que a capacidade de processamento aumenta constantemente, mas, com ela, aumentam também as expectativas do público quanto ao visual desses ambientes virtuais e, por conseqüência, o número total de triângulos empregados.

Além de limitações de memória e processamento, existe outra característica do meio digital a ser considerada na contagem de polígonos de um modelo 3D. Imagens finais geradas por um game são matrizes de pixels, usualmente com resoluções na faixa de 800x600 a 1280x1024 unidades. O algoritmo de renderização precisa sempre decidir, de forma extremamente rápida, se um triângulo ocupa ou não determinado pixel. Quando a área ocupada por um triângulo é pequena (digamos, de 1 a 10 pixels), é comum que ele apareça e desapareça da imagem final, dependendo apenas de pequenas variações do ângulo de observação, criando um efeito desagradável de cintilação. Por esse motivo, modelos altamente detalhados, com muitos triângulos, tendem a produzir mais defeitos na imagem final que modelos de menor resolução. Veremos mais sobre essa característica dos games 3D no próximo capítulo.

Triângulos são excelentes para modelar objetos geométricos facetados, mas não são apropriados para objetos orgânicos, como personagens e vegetação, por exemplo. Desnecessário dizer que são justamente esses objetos mais complexos que dão vida a um ambiente virtual.

É claro que, se os triângulos forem suficientemente pequenos, será possível representar formas curvas de grande complexidade. Mas, além dos já mencionados problemas de imagem gerados por triângulos pequenos, é preciso lembrar que o processador deverá gerenciar cada um deles, seus vértices, a textura a eles aplicada, sua

iluminação, etc., em tempo real. O problema é, então, representar formas curvas e complexas com o menor número de triângulos possível.

Existem duas estratégias básicas para manter baixo o número de triângulos manipulados pelo processador na geração de um ambiente virtual. Primeira: modelar cada objeto com número reduzido de polígonos. Segunda: através de rotinas de programação, selecionar apenas os polígonos que participam da imagem final. Ambos os métodos são complementares. Veremos a seguir alguns exemplos de técnicas empregadas para esses fins.

2.1. Modelagem com baixo número de polígonos

Em lugar de lutar contra a granulação apresentada pelos triângulos, deve-se respeitá-la, usando-a em favor da forma representada. Problema e solução são similares aos da criação de ícones para ambientes Windows, Mac-OS ou Linux, quando uma imagem reconhecível deve ser criada com um número muito reduzido de pixels. Faz-se necessário aproveitar muito bem cada elemento construtivo do objeto.

Uma esfera, por exemplo, é normalmente representada por estrutura de meridianos e paralelos. Programas de modelagem 3D normalmente geram esferas utilizando essa estrutura. A representação funciona, mas apenas quando o número de triângulos é suficientemente alto. Para contagens progressivamente mais baixas, o modelo parecerá menos homogêneo, mais angular, até o ponto de perder qualquer função de representação de uma esfera. Nesses casos, típicos da produção de conteúdo para games 3D, o modelo será muito mais eficiente se construído como estrutura geodésica. A Figura 2 apresenta comparação entre as duas estruturas e os respectivos números de triângulos, em contagens semelhantes. Na estrutura geodésica, os triângulos mantém sempre as mesmas dimensões relativas, contribuindo para a homogeneidade do modelo.

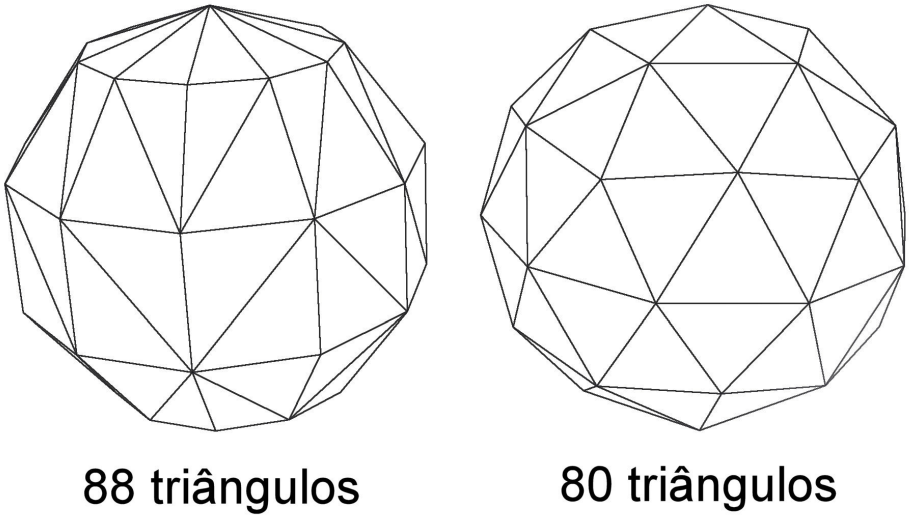


Figura 2

Da mesma maneira, um cilindro pode ser representado de forma mais eficiente por um número ímpar de lados que pelo número par imediatamente superior. Assim, um braço com seção triangular pode parecer mais homogêneo que o mesmo braço com seção quadrada, economizando, grosso modo, 25% de triângulos. Para entender como isso funciona, basta lembrar que a variação do diâmetro aparente de um quadrado (a diferença entre uma diagonal e um lado) é de aproximadamente 42% (um para raiz de dois), enquanto que para um triângulo eqüilátero essa variação é de pouco mais de 15% (coseno de 30 para um). Em outras palavras, ao se rotacionar um braço de seção quadrada, a variação do diâmetro aparente será significativamente maior que a de uma seção triangular. Como resultado, é sempre mais eficiente escolher um número ímpar de lados para a seção do cilindro representado (Figura 3). *Quake* (ID Software, 1996) faz uso dessa técnica nas garras do monstro "Shambler", por exemplo. Games mais recentes exploram capacidades de hardware muito superiores, com personagens que atingem contagens de milhares de polígonos. Entretanto, as regras de otimização continuam importantes para o detalhamento desses modelos mais complexos. Não se trata unicamente de economizar processamento e memória, mas de respeitar características próprias do meio digital. Dispondo de recursos suficientes, podemos modelar individualmente cada fio de cabelo de um personagem, por exemplo. Ainda assim, a representação será significativamente melhor e mais eficiente se os fios de cabelo forem modelados com seção triangular do que se o forem com seção quadrada.

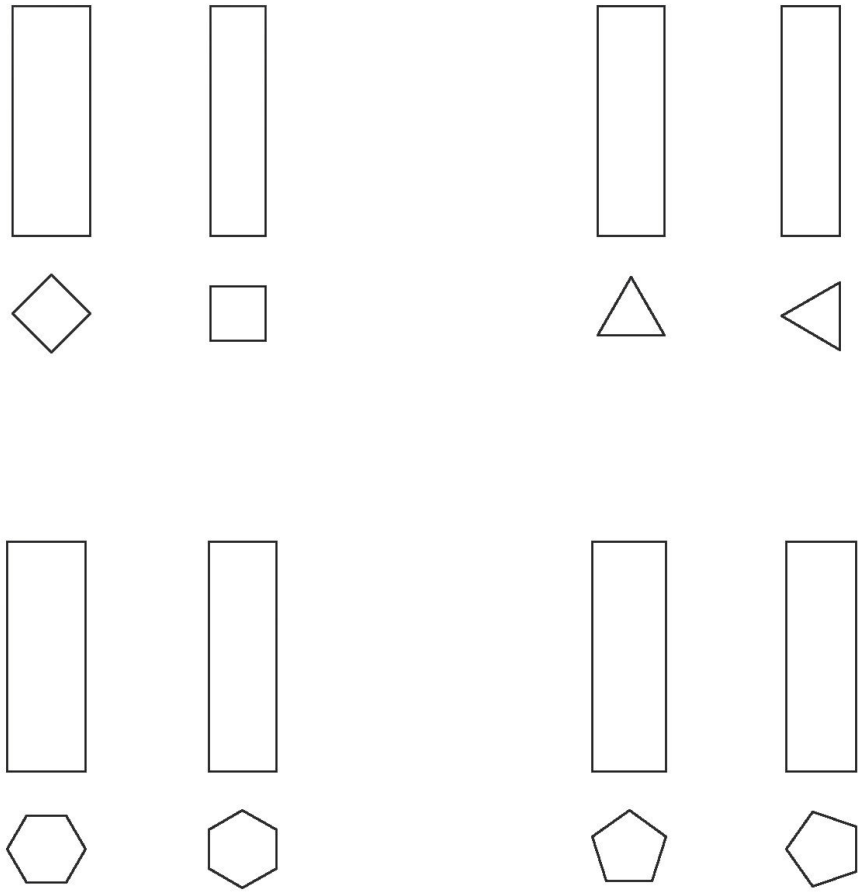


Figura 3

Modelar árvores e vegetação em geral de forma convincente é um dos grandes desafios na criação de ambientes 3D realistas. Mesmo uma árvore modesta possui dezenas de milhares de folhas em planos diversos, unidas por galhos que se bifurcam a partir do tronco. A estrutura é complexa e sua representação 3D exige algo da ordem de dezenas ou centenas de milhares de triângulos. Com um processador veloz, é até possível criar um ambiente com alguns modelos de árvores nesse estilo em tempo real, mas o visual gerado será, muito provavelmente, decepcionante: os contornos de folhas e galhos serão abruptos e a representação, artificial (Figura 4).

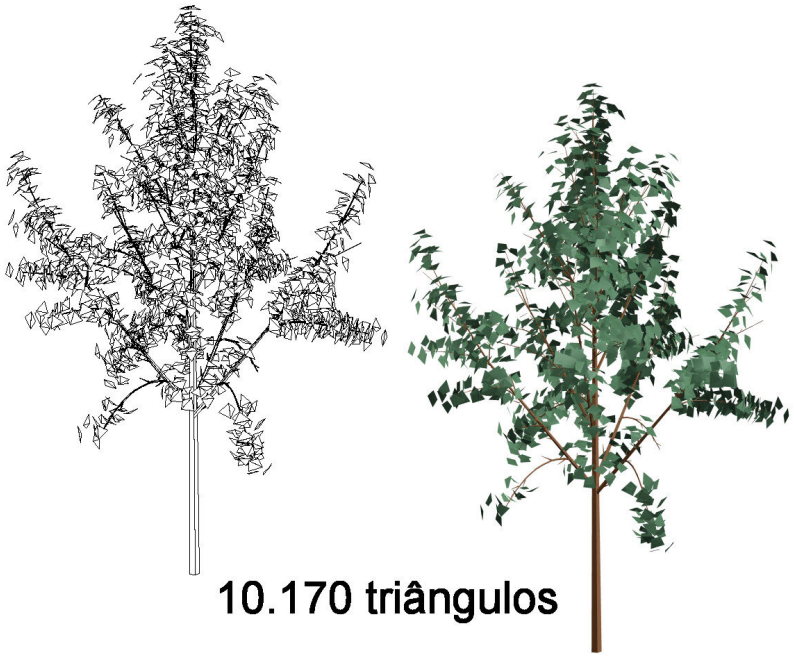


Figura 4

Por outro lado, games como *Battlefield 1942* (EA, 2002) ou *Iracema Aventura* (Perceptum, 2005) apresentam modelos de árvores em grande quantidade e com contagem de polígonos baixíssima – entre 2 e 400 triângulos. O segredo é o mesmo da pintura realista ou do desenho de observação: não se deve tentar recriar em detalhes a estrutura da árvore, e sim apenas o que se percebe dela. Assim, quando uma árvore é vista a grande distância, o movimento do interator no ambiente virtual pouco interfere no ângulo de visualização do modelo, que pode então ser uma imagem com transparência aplicada a um retângulo. Quando a árvore está próxima, o movimento do interator pode alterar significativamente o ângulo de visualização e uma única imagem chapada não seria convincente. Nesse caso, utiliza-se um modelo para o tronco e galhos principais, sendo a copa representada por texturas com transparência aplicadas a poucos triângulos. As transparências possibilitam que os contornos complexos da folhagem sejam mais orgânicos e suaves, o que resulta em imagens finais mais convincentes (Figura 5).

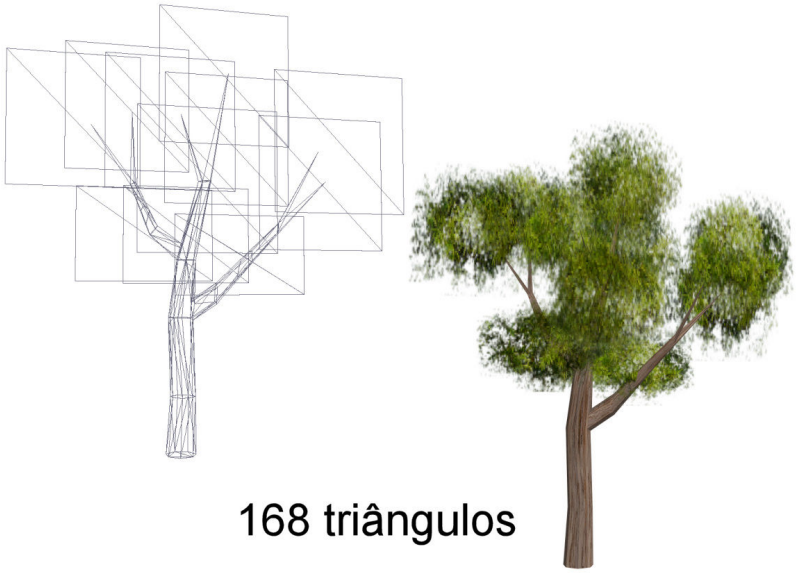


Figura 5

Além da utilização de modelos 3D com baixo número de polígonos, é importante reduzir o trabalho do processador na criação das várias imagens que devem ser exibidas por segundo. Isso é feito principalmente através de testes de visibilidade, com o objetivo de que apenas os triângulos visíveis sejam manipulados em tempo real. O exemplo mais simples é o da remoção das faces traseiras dos modelos, aquelas que não são visíveis por estarem orientadas na direção oposta à câmera. Técnicas mais complexas são apresentadas abaixo. Não nos interessam aqui as técnicas em si, mas, antes, as peculiaridades que tais procedimentos trazem à linguagem dos ambientes virtuais interativos.

2.2. Fustrum

Fustrum é o volume geométrico que representa o campo visual do observador. É a projeção da tela no ambiente virtual e forma uma pirâmide truncada, conforme exemplificado na Figura 6.

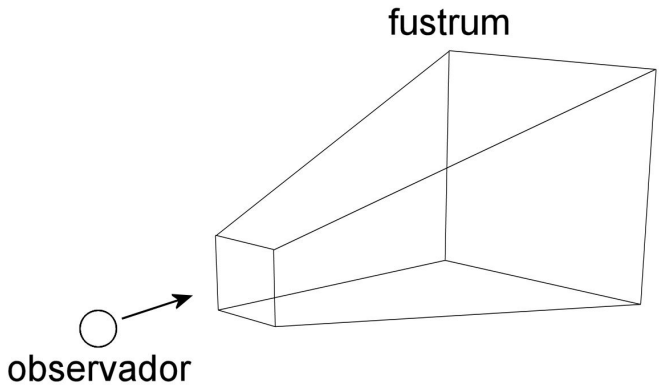


Figura 6

Há um plano de corte próximo e outro distante, além das paredes formadas pela projeção visual das laterais da tela. Triângulos posicionados no interior da figura precisam ser considerados na construção da imagem. Todos os outros triângulos podem ser desconsiderados, do que resulta redução significativa no processamento exigido para a construção da cena.

Entretanto, os dois planos de corte apresentam um problema. Quando o observador desloca-se através do ambiente virtual, triângulos aparecem e desaparecem ao entrarem e saírem do fustrum pelos planos de corte, destruindo qualquer impressão de continuidade. Uma solução aparentemente óbvia seria expandir o plano distante ao infinito e trazer o plano próximo para junto do vértice da pirâmide. Além do acréscimo de triângulos resultante do volume infinito do fustrum, essa solução cria outro problema mais grave: freqüentemente, o fustrum também é usado para o cálculo do buffer de ordenamento, chamado por vezes de Z-Buffer. Esse buffer define quais triângulos estão na frente de quais outros e é fundamental para que a imagem criada seja coerente. Quando o fustrum é estendido, a resolução do Z-Buffer cai significativamente, pois precisa lidar com distâncias potencialmente infinitas. Isso acarreta diversas falhas na imagem, principalmente em triângulos que, mesmo a pequena distância uns dos outros, recebem a mesma ordem no buffer, gerando efeitos de banda e superposição.

Para evitar problemas semelhantes, bastam alguns cuidados básicos. O plano próximo deve ser localizado imediatamente dentro do volume de colisão do avatar do observador. Isso impede que qualquer objeto "sólido" (com colisão) atravesse o plano[5].

[5] Embora o efeito de corte ainda se faça presente em objetos passáveis, como folhagens, ele parece ser plenamente aceito como parte da linguagem nessas situações, como pode ser constatado na vegetação do bestseller *FarCry* (Crytek, 2005)

Já para reduzir o efeito pop-up dos objetos que entram pelo plano distante, a técnica mais apropriada é o fog. Trata-se de mesclar uma cor de fog com os pixels da imagem, de forma progressiva, dependendo de sua distância ao observador, reproduzindo efeito semelhante à neblina real. Basta que a cor do fog reflita a mesma cor do horizonte virtual e que a distância para fog máximo seja ligeiramente inferior ao plano de corte distante do fustrum. Os objetos aparecerão gradativamente ao longe, como que saindo da neblina.

2.3. BSP e Portais

Construções estáticas – como salas, muros e corredores – propiciam diferentes técnicas para redução do número de triângulos processados. Se o observador tem sua visão restrita por estar dentro de um ambiente fechado, não faz sentido processar a totalidade dos triângulos à sua frente, ainda que dentro do fustrum. De fato, a partir de uma sala, será apenas necessário processar o que é visto na própria sala e o que se vê através de eventuais portas e janelas.

Esse é o objetivo de técnicas muito utilizadas nos games atuais, derivadas dos algoritmos conhecidos como BSP (binary space partition) e Portais. O ambiente é subdividido em pequenas áreas, que são ordenadas. Dependendo do algoritmo, as passagens entre áreas, ou portais, são identificadas. Tudo isso é feito uma única vez para cada ambiente, antes da execução do programa principal: é a chamada compilação do nível que, dependendo da complexidade do ambiente, pode levar horas, ou mesmo dias. O programa pode, então, em tempo real, determinar quais são as áreas visíveis de qualquer outra e apenas processar os triângulos necessários para a construção da cena. Essas técnicas reduzem o processamento necessário em proporção quase exponencial, permitindo a criação de ambientes complexos, com centenas de áreas separadas – ambientes que, em virtude da alta contagem total de triângulos, não seriam viáveis sem a utilização de tais algoritmos. Os níveis das séries *Quake* e *Half-Life* são exemplos de utilização intensa de BSP e Portais.

Entretanto, existem detalhes importantes que precisam ser considerados no projeto dos ambientes. Pouco adiantariam BSPs e Portais se os ambientes não isolassem convenientemente a visibilidade. O uso indiscriminado de portas, janelas e transparências entre ambientes separados aumenta a visibilidade e o tempo de compilação do nível, diminuindo a eficiência dos algoritmos em proporção exponencial

ao número de áreas potencialmente visíveis. Esse é um erro muito comum entre modeladores iniciantes.

Outro erro comum é menosprezar a importância da modelagem dos elementos estáticos do ambiente, normalmente trabalhosa e maçante. Existem razões para que essa modelagem seja assim. Para que os algoritmos de separação de áreas possam funcionar com um mínimo de confiabilidade, é preciso obedecer a regras estritas na construção dos ambientes. Por esse motivo, o programa de modelagem 3D para ambientes estáticos limita muito a liberdade de trabalho do modelador, permitindo apenas a modelagem com a chamada geometria CSG (construtive solid geometry). A adoção da CSG permite operações matemáticas que não seriam possíveis com a geometria arbitrária utilizada por programas convencionais de modelagem. Vale lembrar que essa restrição não se aplica a toda geometria de um ambiente virtual 3D, mas apenas àquela que pode ocultar outras geometrias, como paredes e outras estruturas que bloqueiem significativamente a visão do observador. De qualquer forma, não respeitar a construção CSG acarreta, além de erros de compilação, problemas intermitentes como vazamentos de memória e saídas abruptas do game sem motivo aparente.

2.4. Níveis de detalhe

Quando mencionamos o exemplo da modelagem de árvores, vimos que o modelo deve ter em torno de 400 triângulos quando visto de perto, mas apenas 2 quando ao longe. O mesmo acontece com outros modelos 3D em um ambiente virtual. Quando o modelo está próximo do ponto de observação, ele ocupa uma área maior da tela e a necessidade de detalhes volumétricos – conseqüentemente, de triângulos – é grande. Já a distância, esse mesmo modelo pode ocupar apenas poucos pixels da tela. Além dos já citados problemas de imagem gerados por modelos nessas situações, seria um grande desperdício de processamento manter o número original de triângulos. A solução adotada é criar uma série de modelos distintos para cada objeto. Um personagem, por exemplo, pode ser representado por três modelos diferentes: um com 1200 triângulos, para distâncias curtas; outro com 600 triângulos, para médias distâncias; e um terceiro com 200 triângulos, para distâncias maiores. O programa deve simplesmente substituir o modelo de acordo com a distância do ponto de observação (Figura 7).

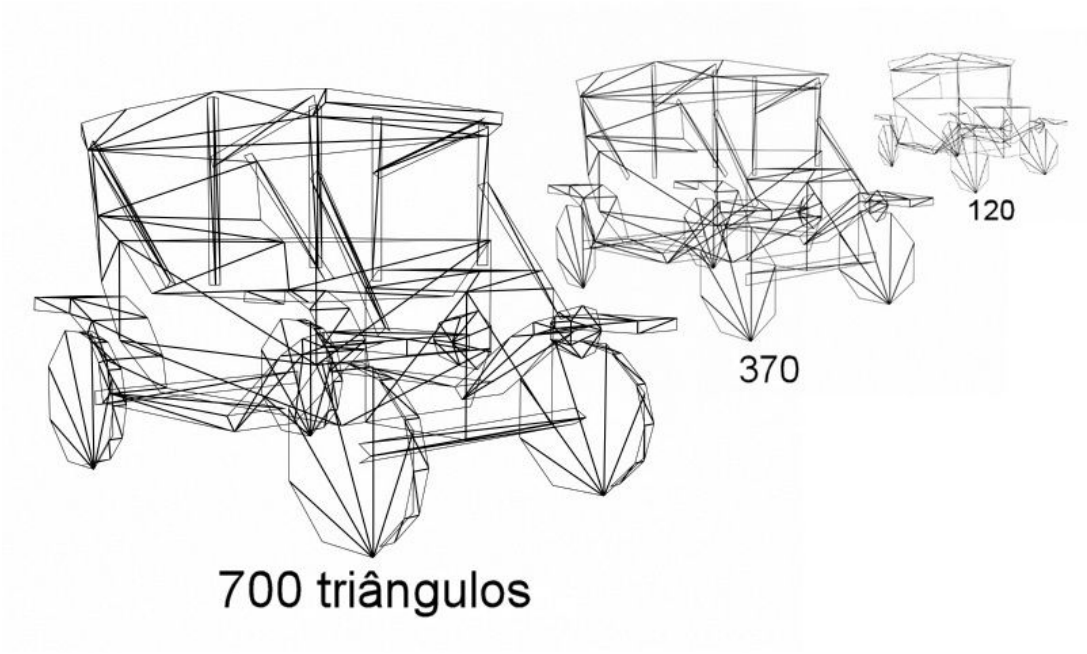


Figura 7

3. TEXTURAS

Quando bem planejadas e produzidas, texturas podem fazer mais que simplesmente representar o material de um modelo 3D. Na modelagem em baixo número de polígonos, as texturas são fundamentais para completar o visual tridimensional de objetos e ambientes. Modelo e textura devem ser balanceados para que se obtenha um visual final convincente sem que processamento e memória disponíveis sejam sobrecarregados inutilmente.

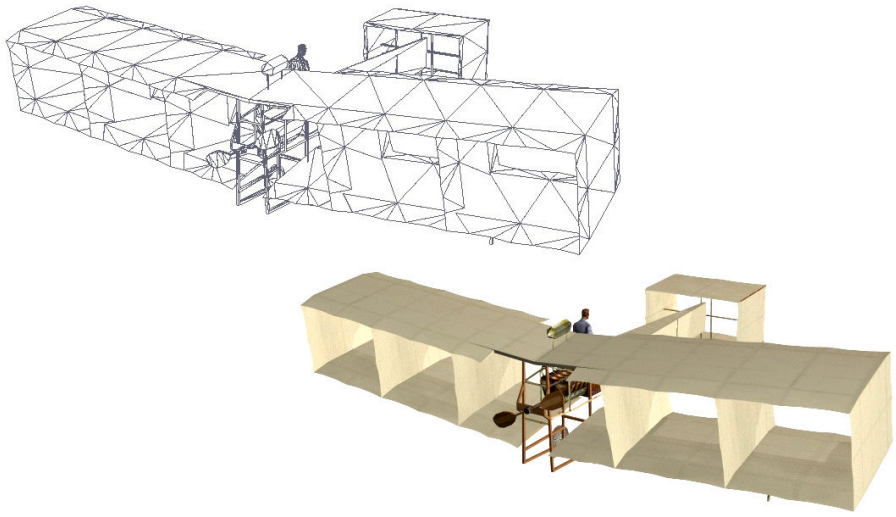


Figura 8: 14 Bis (Perceptum, 2006) – Efeitos de sombra e semi-transparência representados unicamente na textura

Detalhes como dobras de roupas e baixos relevos podem ser eficientemente reproduzidos nessas imagens planas, sem exigir qualquer aumento na contagem de polígonos. No planejamento de texturas, é muito importante observar a integração dos detalhes com a iluminação do ambiente virtual, de modo que as imagens respeitem a direção geral e a intensidade da luz (Figura 8 e Figura 9). Erros na direção da iluminação tendem a destruir o efeito 3D do objeto texturizado, causando uma aparência chapada. Obviamente, sombras e brilhos representados unicamente por texturas são estáticos, ou seja, não respondem a mudanças de iluminação. Porém, para a grande

maioria das aplicações, os efeitos gerados por essa técnica simples são perfeitamente adequados.

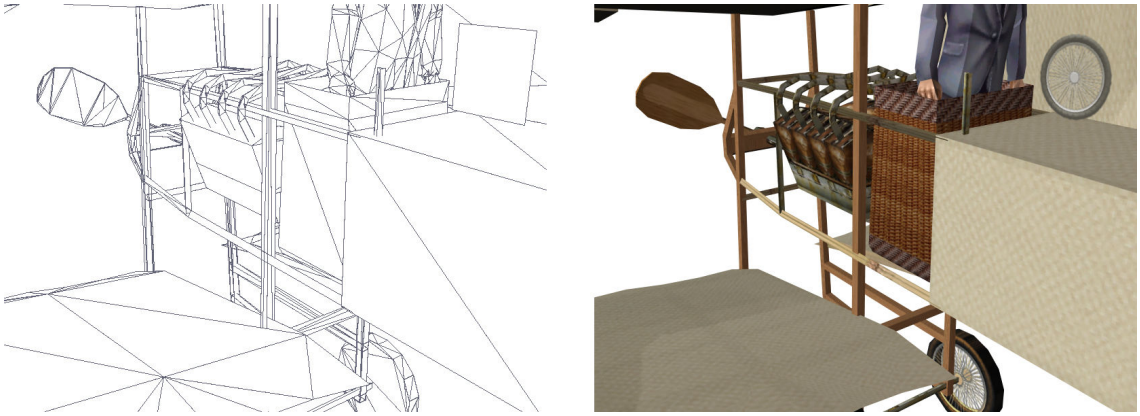


Figura 9: 14 Bis (Perceptum, 2006) – Texturas representam detalhes volumétricos como o relevo do cesto de vime, as dobras do paletó e os raios das rodas

A partir de informações detalhadas do objeto (fotografias, modelos 3D de alta definição ou ilustrações), é possível gerar texturas pré-iluminadas e com detalhes de relevo para posterior aplicação ao modelo 3D de baixa contagem de polígonos. Dessa forma, pode-se representar ambientes e objetos com iluminação relativamente complexa mesmo em *engines* que não disponham de recursos de iluminação mais sofisticados. No game *Super Mini Racing* (Perceptum, 2001), essa técnica foi utilizada para simular efeitos de iluminação mais ricos do que permitiriam, em princípio, o hardware e o programa gráfico disponíveis.

Figura 10: *Super Mini Racing* (Perceptum, 2001) – Efeitos de luz e sombra pré-renderizados nas texturas das paredes e dos carros

Outra função interessante, embora não documentada, das texturas é a suavização das linhas retas características dos modelos 3D de baixa resolução. No exemplo da Figura 11, podemos notar que os arcos têm aparência suave e contínua, integrada com a iluminação e sem quebras visíveis. Isso ocorre porque arcos, janelas e portas não estão modelados, mas, antes, representados por texturas como descrito mais acima. O modelo 3D é apenas um paralelepípedo, sem qualquer detalhe adicional. Entretanto, repare na base do paralelepípedo, que faz contato com a textura do chão. A linha de transição é abrupta e denuncia a simplicidade do modelo.

Figura 11

Esse problema pode ser minimizado com uma pequena alteração na textura do modelo. Basta mesclar levemente a base da textura dos arcos com a textura utilizada no chão, conforme o exemplo da Figura 12.

Figura 12

3.1. O Mito da Resolução

Livros sobre modelagem 3D normalmente recomendam texturas de alta resolução. Mesmo na literatura específica sobre modelagem 3D para games, a parcimônia no tamanho das texturas é indicada apenas como meio de contornar as limitações de memória e processamento. A concepção geral é a mesma: quanto maior a definição das texturas, melhor a qualidade final da imagem.

> Não faz muito tempo, imagens de 512 x 512 pixels eram consideradas texturas grandes. Para os games da próxima geração (daqui a 2 ou 3 anos), já estamos utilizando imagens de 1024 x 1024 como tamanho padrão e freqüentemente alcançamos 2048 x 2048. Some-se a isso os avanços tanto em hardware como em software e o salto de qualidade visual de um game torna-se simplesmente inacreditável. (AHEARN, 2006)[6]

Excetuando-se casos muito específicos, como os de texturas aplicadas a terrenos e outras grandes superfícies, essa é uma concepção errônea. É certo que avanços no hardware permitirão continuamente a utilização de texturas de grandes dimensões, mas isso não significa que elas devam ser empregadas indistintamente. Essa concepção, importada da modelagem 3D tradicional, pode causar problemas que vão muito além da sobrecarga de memória e de processamento. Texturas de grandes dimensões podem destruir a qualidade de imagem do game.

Ao contrário do que se poderia supor, texturas de altíssima resolução não geram, necessariamente, imagens em tempo real de alta qualidade. Notei pela primeira vez essa característica contra-intuitiva durante o desenvolvimento de *Incidente em Varginha* (Perceptum, 1998). Devido à limitada memória disponível nos sistemas da época, precisei reduzir as dimensões das principais texturas de diversos ambientes. No caso específico de algumas paredes da estação Sé do Metrô (o quarto nível do game), a redução foi drástica: de 512 x 512 pixels, as texturas caíram para 128 x 128. Para minha surpresa, a qualidade final da imagem melhorou significativamente após a redução.

Passei dias tentando encontrar uma explicação para o que parecia ser um paradoxo: imagens finais melhores resultavam do emprego de texturas de qualidade inferior. Minha primeira providência foi testar o *engine* de renderização[7], pois supus que o mesmo estivesse apresentando problemas com texturas de maior resolução. Entretanto,

---

[6] "Not long ago, a 512 x 512 image was considered a large texture. For the next-generation games (2 to 3 years out), we are already using 1024 x 1024 images as a standard size and frequently go up to 2048 x 2048. Add to that hardware and software advances and the jump of visual quality of a game is simply incredible" (AHEARN, 2006).

[7] Acknex A3, Conitec Datasystems, 1997.

o *engine* funcionava corretamente. A resposta, mais simples, estava na própria natureza digital do meio.

Texturas são imagens compostas por unidades digitais, os pixels, exatamente as mesmas unidades básicas da tela em que são exibidas. O problema não existiria se essas imagens fossem apresentadas em uma escala de um para um, ou seja, para cada pixel da tela corresponderia um pixel da imagem. Mas não é isso o que ocorre com as imagens utilizadas como texturas num ambiente virtual 3D. Uma textura pode estar distante do ponto de observação, caso em que será necessário encaixar vários pixels da textura dentro de um único pixel da tela, geralmente de forma fracionária. Ela pode também estar muito próxima, quando um pixel da textura ocupará vários pixels da tela. Além disso, a textura estará freqüentemente inclinada em relação ao ponto de observação, com os pixels mais distantes ocupando menos área que os pixels mais próximos.

Uma solução seria identificar quais porções da textura encaixam em cada pixel da tela, calcular a média de cor desse trecho da textura e aplicá-la ao pixel da tela em questão. É exatamente isso o que fazem programas de geração de imagens sintéticas, como *Maya* ou *3D Max*, com resultados excelentes. Entretanto, essa técnica é ainda inviável nos games 3D, cujas imagens precisam ser geradas em tempo real, devido ao custo de processamento que ela exige.

Por esse motivo, nos games 3D é comum utilizar-se o pixel mais próximo: apenas um pixel da textura é escolhido para representar seus vizinhos. Quanto maior for a resolução gráfica da textura, mais pixels dessa imagem precisarão ser encaixados em cada pixel da tela e maior será a arbitrariedade da escolha do representante. Texturas de grande definição produzem, dessa forma, imagens com vários problemas de *aliasing*: cintilações, padrões repetidos (Moirë), cores que "correm", etc. Uma solução intermediária é a aplicação de filtros em tempo real, que consideram as cores dos pixels imediatamente vizinhos de forma simplificada. Mas o custo de processamento exigido é relativamente alto para os resultados alcançados.

O processo pode ser mais facilmente compreendido através do exemplo da Figura 13. Para três tamanhos diferentes de texturas (2048 x 2048, 128 x 128 e 64 x 64), apresentam-se resultados de imagem na forma de cubos texturizados. A coluna "limite de textura" representa o melhor resultado possível para cada uma considerando-se uma

resolução de imagem final ilimitada[8]. Na prática, é claro, a imagem deve ser exibida em tela de resolução finita.

A coluna "limite de resolução" apresenta o melhor resultado possível para imagens finais em resolução aproximada de 160 x 120 pixels, utilizando a média de cor para renderizar os cubos. Vale notar que, nesse caso, os resultados finais são muito próximos e praticamente independentes das dimensões das texturas originais.

Finalmente, a coluna "limite de processamento", igualmente em resolução de 160 x 120 pixels, mostra as imagens finais utilizando o pixel mais próximo para renderizar os cubos. Nesse caso, típico mesmo para a tecnologia de games atuais, a textura de 64 x 64 apresenta um resultado muito mais fiel ao limite de resolução que a de 2048 x 2048. Foi esse o efeito que me pareceu paradoxal em 1997.

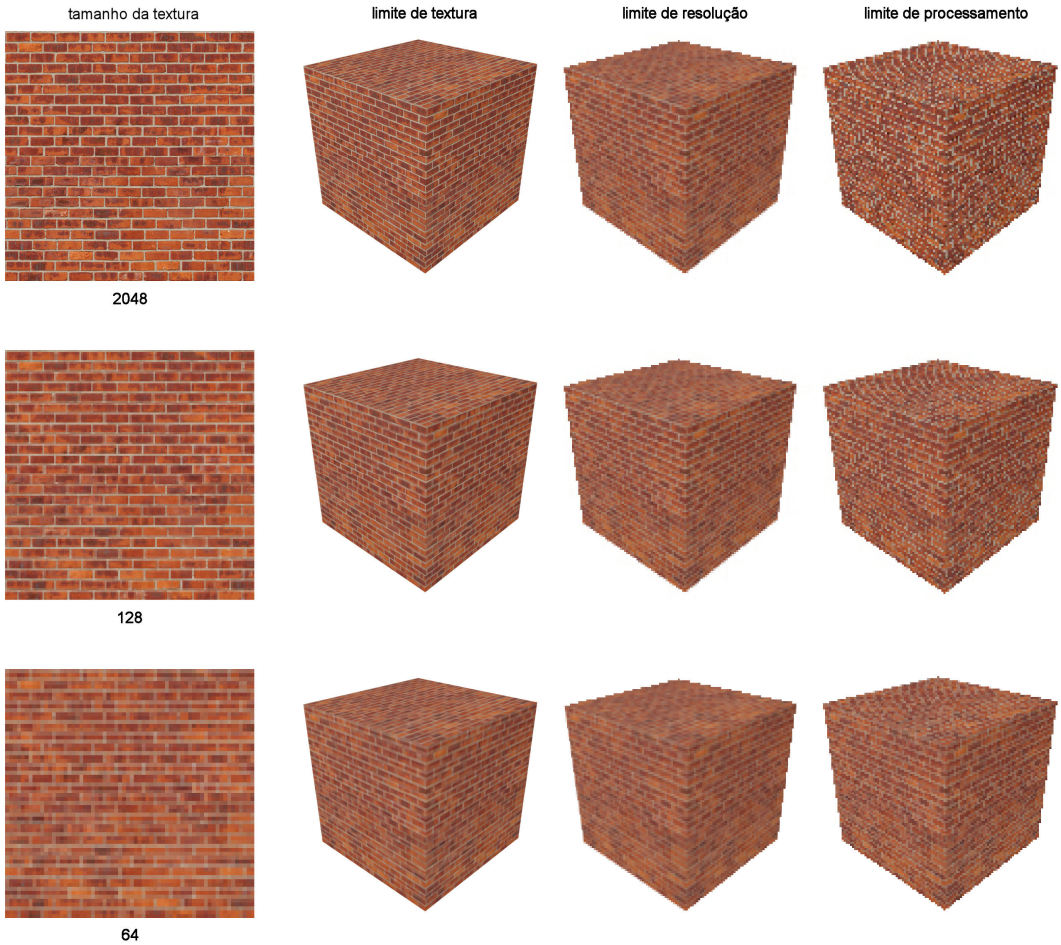


Figura 13: quanto maior o tamanho da textura, mais evidentes os erros de *aliasing* provocados por limitação de processamento

[8] Para ser exato, há que se considerar aqui a resolução da página impressa (600 dpi). Entretanto, esse limite afeta apenas a textura e a imagem final em "limite de textura" do cubo de dimensão 2048. As outras texturas, bem como as colunas "limite de resolução" e "limite de processamento", não sofrem com a resolução limitada da página impressa.

Mesmo considerando os avanços desde então (sobre os quais falaremos brevemente a seguir), é importante notar que existe um limite para a resolução de exibição de imagem. Pouco adianta utilizarmos uma textura de 2048 x 2048 se ela será exibida num espaço de 120 pixels da tela, por exemplo. Uma textura de 128 x 128 cumprirá a mesma função, de forma mais eficiente, sem apresentar os erros de *aliasing* descritos acima, exigindo apenas 1/256 dos recursos de memória e processamento.

3.2. Mapeamentos MIP e de Normais

Como vimos, a qualidade da imagem final depende da relação entre a dimensão da textura e a quantidade de pixels utilizada para sua exibição. Por esse motivo, uma textura de altíssima definição parecerá boa quando vista de muito perto, mas pobre e defeituosa quando vista ao longe. A textura precisa representar, tanto quanto possível, apenas os detalhes que deveriam ser vistos à distância em questão. Para isso, são criadas várias versões da mesma textura, em resoluções progressivamente menores (normalmente 512x512, 256x256, 128x128, 64x64 e 32x32 pixels) e o programa deve trocá-las de acordo com a distância do ponto de observação. Essa técnica é denominada mapeamento MIP (do latim Multum In Parvo, muito em pouco espaço, referindo-se diretamente ao processo de encaixe dos pixels). Desenvolvido por Lance Willians, o conceito de mapeamento MIP foi publicado em seu artigo "Pyramidal Parametrics", de 1983.

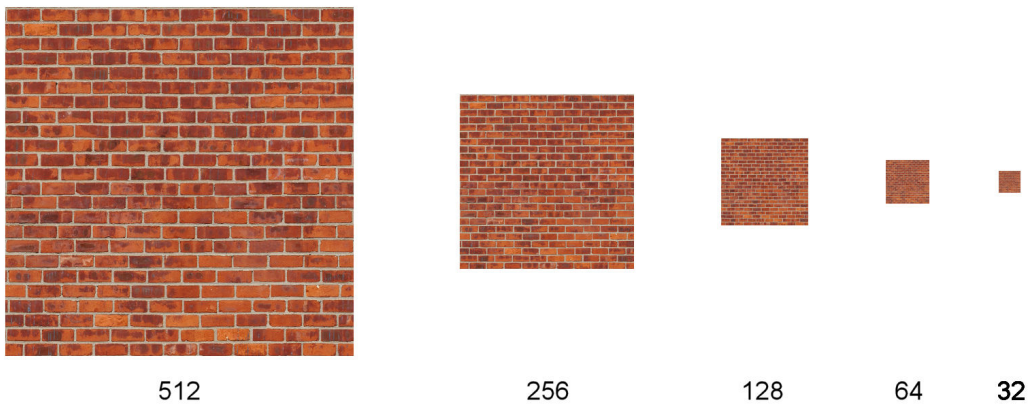


Figura 14: exemplo de mapeamento MIP

As diversas versões reduzidas das texturas MIP são geradas automaticamente por *engines* 3D atuais[9], bastando informar a textura de maior resolução. Mas ainda é preciso muito cuidado na seleção da textura principal. Se a dimensão da maior textura for, por exemplo, 2048 x 2048, a segunda textura MIP será gerada automaticamente como 1024 x 1024, a terceira como 512 x 512, e assim sucessivamente. Do exagero no tamanho das texturas sempre decorrerão problemas de *aliasing*, memória e processamento, como os descritos acima, em maior ou menor grau.

Mais recente, a técnica conhecida como mapeamento de normais contribui significativamente com o detalhamento volumétrico de modelos 3D. Trata-se da aplicação, sobre a textura original, de uma textura especial que informa os ângulos normais de cada pixel da superfície. Em sua estrutura de dados, a textura de mapeamento normal é uma imagem comum, mas as componentes R, G e B são interpretadas como coordenadas de vetores normais, não como pixels. É possível então calcular como a luz é refletida em cada pixel da textura original e conferir uma impressão muito realista de relevo a uma superfície virtualmente plana. Diferentemente da técnica de pré-renderização descrita no início deste capítulo, o mapeamento de normais reage a mudanças de iluminação em tempo real, integrando corretamente as informações volumétricas da textura com o ambiente virtual, de forma dinâmica. Esse tipo de processamento tem seu custo, pois é complexo e geralmente efetuado no nível do hardware de vídeo, mas permite a representação de detalhes volumétricos em alta definição sem exigir qualquer aumento da contagem de triângulos dos objetos. Originalmente desenvolvido para estações gráficas de processamento paralelo, o mapeamento de normais está se tornando padrão na indústria de games e é empregado em vários títulos recentes como *Half-Life 2* (Valve, 2004), *Far Cry* (Crytek, 2004) e *Call of Duty 2* (Infinity Ward, 2005). Assim como acontece com as texturas convencionais, o mapeamento de normais deve ser dosado para contribuir com o efeito final sem sobressair. Em novos títulos, entretanto, é comum que o mapeamento de normais seja exagerado. Compreensivelmente, esse é um modo de os desenvolvedores colocarem em evidência a tecnologia disponível em seus produtos, mas tal prática não deve sobreviver ao curto período em que essa técnica ainda será vista como novidade.

Triângulos, texturas ou mapeamentos, cada elemento tem suas características específicas e limitações que devem ser tanto conhecidas como respeitadas. Um dos

[9] *Torque Game Engine 1.5* (Garagegames 2006) ou *3D Game Studio A6* (Conitec 2006), por exemplo.

principais trabalhos do modelador é justamente balancear essas características e administrar recursos disponíveis de memória e processamento para que o efeito final apresente, simultaneamente, qualidade e viabilidade.

4. ANIMAÇÃO

4.1. Movimento versus *Forma*

Ao jogar *Rainbow Six* (Red Storm Entertainment, 1998), em 1999, notei uma característica intrigante em seus personagens. Analisando-os de perto, constatava-se que tanto os modelos 3D como suas texturas simplesmente não convenciam. Braços, pernas, troncos e cabeças eram extremamente simples e angulares. As proporções do corpo não pareciam corretas e, para piorar o efeito, as texturas eram muito chapadas, conferindo ao modelo um visual final plano e disforme. Em suma, a modelagem 3D deixava muito a desejar. Mas essa impressão desaparecia misteriosamente quando os modelos se movimentavam. Em diversos tipos de ação – de modo especial quando subiam escadas de serviço, com mãos e pés apoiando-se convincentemente em degraus verticais – os personagens pareciam adquirir vida própria e toda aquela má impressão causada pela modelagem 3D de baixa qualidade desaparecia sem deixar vestígios. Em determinadas circunstâncias, o jogo parecia mostrar imagens filmadas de atores reais, embora modelos e texturas, obviamente, continuassem os mesmos. A diferença estava no trabalho de animação, muito bem produzido, que de alguma forma compensava a pobreza dos modelos.

No capítulo 6 do livro *Image and Brain*, Stephen Kosslyn analisa a importância dos movimentos na identificação de objetos em imagens degradadas. A explicação pode ser diretamente aplicada à curiosa constatação descrita acima:

> O movimento desempenha um papel crítico em dois tipos distintos de processo. Primeiro: ele ajuda a delinear o próprio objeto, fornecendo outro indício para a localização de bordas e regiões. A teoria da Gestalt formula a 'lei do destino comum', estabelecendo que estímulos que se movem do mesmo modo são agrupados na mesma unidade perceptiva. Assim, mesmo que a imagem de entrada esteja tão degradada que os contornos de um objeto não sejam visíveis quando o mesmo está parado, eles podem se tornar evidentes tão logo o objeto se mova. Johansson (1950, 1973, 1975) provavelmente forneceu a mais cativante demonstração desse fenômeno ao vincular luzes às articulações de pessoas e observá-las no escuro. Quando as pessoas estavam paradas, a organização dos segmentos não era clara; mas tão logo elas se moviam, tornava-se óbvio quais luzes pertenciam ao mesmo segmento do corpo. De fato, Ulman (1979) e outros demonstraram que o movimento pode, por si, delinear a estrutura tridimensional de um objeto.

> Segundo: indícios dos movimentos podem ser usados para reconhecer um objeto. De fato, em suas experiências com luzes vinculadas às articulações de pessoas, Johansson descobriu que observadores podiam usar indícios dos movimentos para classificar uma figura como homem ou mulher. Mais ainda, Cutting e Kozlowski (1977; bem como Cutting e Proffitt 1981) mostraram que pessoas podem utilizar tais indícios para identificar indivíduos específicos. Cutting (1978) mostrou que observadores podem determinar se uma pessoa carrega uma carga leve ou pesada baseados apenas em seis pontos de luz. Está claro que muitos objetos apresentam padrões de movimento distintivos, como uma folha que cai, um gato que salta, ou um bastão de baseball em seu movimento de rebatida (ver também Cutting 1982). (KOSSLYN 1996, pp. 153-154)[10]

No caso específico do desenvolvimento de games, parece razoável concluir que a animação de um objeto seja mais crítica que sua aparência estática ou que a definição da imagem final. Após experimentar animações diversas para diferentes estilos de personagens nos games *Incidente em Varginha*, *IV2*, *Scooter Challenge* e *Iracema Aventura*, uma conclusão inescapável é a de que as animações de personagens são mais importantes que suas aparências estáticas. Para personagens de games de ação, o reconhecimento de padrões do cérebro humano seria mais exigente com o movimento do que com a forma. Obviamente, isso não implica que a modelagem deva ser descuidada, mas, antes, que a animação deva receber atenção especial.

4.2. Percepção de Realidades

Outro desdobramento interessante dessas experiências com a animação teve início com a percepção de que personagens não-realistas, como os competidores de *Scooter Challenge*, são muito mais tolerantes a animações pouco comuns, ou mesmo absurdas, que personagens realistas. Na prática, um personagem de proporções e aspecto humanos parecia tornar evidente quaisquer desvios de animação – como o desrespeito às leis da

[10] "Motion plays a critical role in two distinct types of processes. First, it helps to delineate an object itself, providing another cue as to the location of edges and regions. The Gestalt psychologists formulated the 'law of common fate', which states that stimuli that move in the same way are grouped into the same perceptual unit. Thus, even if the input image is so degraded that the contours are not visible when the object is motionless, they may become immediately apparent as soon as the object moves. Johansson (1950, 1973, 1975) probably provided the most compelling demonstration of this phenomenon by attaching lights to people's joints and observing then in the dark. When the people were motionless, it was not clear how the segments were organized; but as soon as they moved, it was obvious which lights belonged to the same segment. Indeed, Ulman (1979) and others have shown that motion alone can delineate the three-dimensional structure of an object.
Second, motion cues can be used to recognize an object. Indeed, in his experiments in which lights were attached to people's joints, Johansson found that observers could use motion cues to classify a figure as a man or woman. Moreover, Cutting and Kozlowski (1977; and also Cutting and Proffitt 1981) showed that people can use such motions cues to identify specific individuals. Cutting (1978) showed that observers can use as few as six points of light to determine whether a person is carrying a light or heavy load. It is

física ou aos limites angulares das articulações, por exemplo. Como conseqüência, animar personagens humanos exigia muito mais trabalho e atenção a detalhes. Supus que isso acontecia por sabermos o que esperar de movimentos humanos e usarmos esse conhecimento como comparação, muitas vezes de forma inconsciente, entre nossa realidade e a realidade do game. O mesmo não ocorreria, por exemplo, com o porco-espinho azul Sonic, ou com o encanador baixinho Mario, que são livres para executar os mais absurdos movimentos. Mas, para quem jogou *Sonic the Hedgehog* (Sega, desde 1991) ou *Super Mario Bros.* (Nintendo, desde 1985) por algum tempo, esses personagens apresentam movimentos esperados e convincentes, totalmente integrados com as leis do ambiente ao seu redor, segundo sua própria realidade. Tais animações não são absurdas – elas são parte integrante do comportamento esperado desses personagens. Essa estranha familiaridade com animações e comportamentos não naturais parece reforçar o conceito de "criação ativa da crença", proposto por Janet Murray:

> A prazerosa rendição da mente a um mundo imaginário é geralmente descrita, nas palavras de Coleridge, como "a suspensão intencional da descrença". Mas essa é uma formulação muito passiva, mesmo para os meios de comunicação tradicionais. Quando entramos num mundo ficcional, fazemos mais do que apenas "suspender" uma faculdade crítica; também exercemos uma faculdade criativa. Não suspendemos nossas dúvidas tanto quanto criamos ativamente uma crença. Por causa de nosso desejo de vivenciar a imersão, concentramos nossa atenção no mundo que nos envolve e usamos nossa inteligência mais para reforçar do que para questionar a veracidade da experiência. (MURRAY, 2003, p. 111)

Entretanto, parece ser mais fácil criar ativamente uma crença quando trabalhamos com figuras não-realistas. Os concorrentes de *Scooter Challenge* têm proporções, texturas e animações típicas de personagens de desenhos animados, mas correm em ambientes relativamente realistas. Eles impulsionam seus patinetes em ruas de cidades como Londres ou Cairo com velocidades que ultrapassam 140 Km/h (na escala do mundo virtual) sem que isso cause qualquer estranheza na grande maioria dos jogadores. Como experiência, substituí um desses personagens por uma figura de características humanas, extraída de *IV2*, mantendo as mesmas animações do corredor original. Embora nenhum outro parâmetro tivesse sido alterado, imediatamente foi possível perceber que o pé direito do personagem, que deveria gerar o impulso necessário para a corrida, parecia deslizar no piso, pois não acompanhava a velocidade da corrida, que

---

clear that many objects have distinctive patterns of motion, such as a falling leaf, a pouncing cat, and a swooping bat (see also Cutting 1982)." (KOSSLYN, 1996, pp. 153-154).

parecia excessivamente maior. A figura humana, de alguma forma, destruía um equilíbrio que havia sido atingido facilmente com personagens não-realistas. Substituir os personagens de *Scooter Challenge* por figuras humanas exigiria novas animações e, possivelmente, sensível redução da velocidade dos patinetes.

4.3. Captura de Movimentos

A solução definitiva para a animação convincente de personagens humanos parecia ser a captura de movimentos. Trata-se de uma técnica de digitalização de movimentos de atores reais para posterior aplicação em animações de personagens virtuais. Utilizada amplamente em games e filmes de animação, a técnica possibilita criar complexas animações de personagens, evitando em grande parte o árduo trabalho da animação manual. É interessante notar que o processo de captura mais comum emprega marcadores luminosos nas articulações dos atores, de forma análoga às experiências de Johansson (1950, 1973, 1975). A imagem dos marcadores é captada por diversas câmeras posicionadas em ângulos diferentes. Um software analisa então as diferenças de imagem entre as câmeras para determinar a posição espacial de cada marcador.

A análise de Kosslyn sobre a importância da percepção de movimentos na identificação de objetos em imagens degradadas, citada anteriormente, parece indicar que a captura de movimentos seria uma solução para tornar convincentes as animações de qualquer personagem de proporções e formas humanas (ainda que levemente). Isso é verdade, mas apenas quando tratamos com imagens degradadas, como ambientes escuros, resolução baixa ou texturas excessivamente chapadas, por exemplo.

Quando as imagens finais são nítidas e detalhadas, como é o caso em games e filmes atuais, a captura de movimentos introduz um problema. Personagens com movimentos capturados transmitem nítida impressão de ausência de peso, conforme descrito por Richard Williams no livro *The Animator's Survival Kit*. Mesmo quando o movimento é precisamente capturado a partir de um ator, há um estranho aspecto de leveza no personagem virtual que recebe a animação. De fato, esse efeito de ausência de peso pode ser notado em várias cenas de filmes que utilizam captura de movimentos para a animação de modelos 3D. Williams apresenta algumas soluções para o problema, mas não fornece uma explicação para suas causas.

Baseado nas informações de Kosslyn sobre imagens degradadas, proponho a seguir uma explicação para a aparente ausência de peso nos movimentos capturados. O cérebro

humano é extremamente eficiente na avaliação de movimentos, percebendo detalhes sutis dos quais, muitas vezes, não nos damos conta conscientemente, sendo capaz de extrair informações adicionais a partir de movimentos em imagens mal resolvidas. É lícito supor que, quanto mais nítidas forem as imagens, mais informações sobre os movimentos podem ser processadas. Pequenas oscilações da musculatura de uma pessoa que caminha, assim como oscilações mais evidentes de roupas e cabelos, quando não reproduzidas em um modelo virtual, deixam a animação final com aspecto de ausência de peso.

Recriar, em tempo real, os movimentos de musculatura, roupas e cabelos raramente é viável. Isso exigiria não apenas um número muito elevado de polígonos, como também o gerenciamento da animação de cada um deles. Uma solução simples, utilizada amplamente em desenhos animados e muito eficiente em games 3D, é exagerar ligeiramente o efeito do peso (WILLIAMS, 2002). Corpo e membros devem oscilar um pouco a mais que o movimento capturado de um ator real, como se a inércia fosse maior. Esse recurso confere aos personagens certa impressão de peso, mas as ações que envolvem impactos, como correr ou lutar, não são afetadas e continuam suaves demais.

4.4. Movimentos Impossíveis

Ao animar um dos personagens de *Incidente em Varginha 2* (Perceptum, inédito), em 1999, utilizei pela primeira vez uma biblioteca de movimentos capturados. A animação de corrida, embora precisamente capturada de um ator real, parecia-me especialmente sem graça. O efeito de ausência de peso era evidente e o personagem parecia flutuar, movendo as pernas sem qualquer impacto aparente com o solo. Reforcei o movimento de subida e descida do corpo e isso ajudou, mas o contato dos pés com o solo ainda parecia extremamente suave. Editando o movimento das pernas de forma mais ou menos aleatória, notei repentinamente que a batida de um dos pés no solo estava muito mais convincente, causando nítida impressão de impacto. Não tardou para que o motivo fosse descoberto: esse era o efeito de um erro de animação. Durante a edição do movimento, dobrei acidentalmente um dos joelhos no sentido inverso, apenas um frame antes do toque do pé no solo (Figura 15). Foi o suficiente para gerar um efeito curioso: a perna que continha o erro parecia realmente bater no chão, enquanto que a outra, com o movimento capturado original, ainda parecia flutuar.

Figura 15

Como a distorção da articulação ocorria em um único frame da seqüência animada, era impossível percebê-la com tal. Observando-se a animação em tempo real, não se via a dobra invertida do joelho, mas havia a forte impressão de que a *barra da calça* oscilava rapidamente com o impacto. Ora, para manter uma contagem de polígonos baixa, a barra da calça era modelada integralmente com o sapato e o resto da perna (como pode ser visto na Figura 15): não havia como fazê-la oscilar separadamente. Ainda assim, a impressão de movimento independente da barra da calça era inegável. Creio que o cérebro re-interprete esse tipo de distorção – inverossímil, mas extremamente rápida – como algo conhecido: nesse caso, como a oscilação rápida da roupa durante o impacto do pé no solo.

Na verdade, meu erro resultou num encontro acidental com certa técnica que já vem sendo utilizada há décadas na animação. Em *The Animator's Survival Kit* (WILLIAMS, 2002), Richard Williams descreve o artifício da antecipação invisível. A antecipação tradicional é o movimento de preparação ao movimento principal. Por exemplo, antes de saltar para frente, o personagem deve inclinar-se para trás[11]. Já a antecipação invisível consiste, simplesmente, em fazer com que um movimento rápido seja precedido por outro, de preparação, porém rápido demais para ser percebido como antecipação tradicional. No exemplo da Figura 15, a antecipação invisível configura-se quando o joelho se dobra levemente ao contrário, em preparação para o momento do impacto, no qual ele deverá dobrar-se no sentido correto e de forma mais violenta.

[11] O movimento de antecipação tem função narrativa, pois indica ao observador o que está por acontecer.

Quanto à minha impressão inicial, de que personagens realistas não toleravam distorções desse tipo, tudo parece depender do momento e da velocidade com que elas ocorrem. Uma articulação dobrada ao contrário pode arruinar o efeito de uma animação se for percebida, mas pode contribuir enormemente com a impressão de realismo se for empregada na velocidade e no momento corretos. Segundo Richard Williams, "podemos tomar grandes liberdades com ações rápidas – mesmo com figuras realistas" (WILLIAMS, 2002)[12]. Tenho utilizado o artifício de antecipação invisível desde 1999, com bons resultados em movimentos de impacto, sempre trabalhando com distorções de, no máximo, 1/20 de segundo. Apesar de minha inclinação a compreender o efeito como re-interpretação do cérebro, não há explicação teórica conhecida para o realismo resultante. Mas é certo que a antecipação invisível deixa os movimentos mais convincentes, tanto em personagens de desenhos animados como em personagens 3D supostamente realistas.

[12] "We can take great liberties with fast actions – even with realistic figures" (WILLIAMS, 2002).

5. COMPORTAMENTO ARTIFICIAL

Uso aqui o termo comportamento artificial por considerá-lo mais abrangente e menos arrogante que inteligência artificial. Trata-se da simulação de comportamentos e tomada de decisões de personagens e objetos virtuais. É importante notar que algoritmos de comportamento artificial são utilizados em todas as ocasiões de interação em um game, seja no movimento da câmera de acordo com os comandos do interator, na simples abertura de uma porta ou na reação tática de personagens ditos inteligentes. A programação de comportamentos artificiais é, por essa razão, condição necessária à interatividade nos games.

5.1. Previsibilidade e Controle

De todas as técnicas e conhecimentos empregados no desenvolvimento de games 3D, a programação parece ser a menos compreendida e a mais sujeita a pré-concepções, mesmo entre boa parte dos desenvolvedores. É possível que isso decorra do fato de a programação ter duas atribuições distintas num game 3D, que podem ser ilustradas pelo seguinte comentário de Chris Crawford:

> Enquanto designers de games, nós somos deuses absolutos. Um tipo de deus diz, "O.K., agora esta folha cairá um pouquinho aqui e então esse vento soprará um pouco lá adiante". O outro tipo de deus diz, "Aqui estão as leis da física. Virem-se". (CRAWFORD *apud* SCHIESEL, 2005)[13]

Como o primeiro deus, o designer de games – ou, mais especificamente, o programador – rege todo o rigoroso controle necessário à atualização do ambiente virtual. Cada vértice e cada triângulo visível precisa ter seus parâmetros (posição, orientação, velocidade, etc.) controlados com precisão e em tempo real. O mesmo vale para as texturas, as animações, os sons, a luz, enfim, para todos os elementos que compõem o ambiente virtual. Nessa função, a programação é necessariamente rígida e previsível, o que vale dizer que o controle está totalmente nas mãos do programador.

[13] As a game designer you are an absolute god. One kind of god says, "O.K., now this leaf will fall a little bit here, and then this wind will blow a bit over there." The other kind of god says, "Here are the laws of physics. Go for it". (CRAWFORD *apud* SCHIESEL, 2005)

Mas estender esse conceito ao comportamento de personagens gera apenas uma ilusão de controle, viável unicamente em sistemas muito restritos, e tende a tornar a experiência linear e decepcionante. Não é viável controlar comportamentos artificiais desse modo, salvo em situações muito específicas e lineares (como os diálogos entre alguns personagens da versão original de *Half-Life*, por exemplo). Além disso, a técnica lineariza também a interação com o usuário, a quem cabe assistir ao game quase que como a um filme, com poucas opções de intervenção.

Como o segundo deus, entretanto, o programador explora o fenômeno da emergência ao dotar objetos virtuais e personagens de comportamento artificial baseado na interação constante entre diferentes instruções. Nesse segundo caso, a programação é rígida, mas não necessariamente previsível. Numa aparente contradição com a primeira função, o controle do programador é significativamente reduzido e exercido de forma indireta quando se trabalha com sistemas emergentes. Falaremos sobre essa função da programação neste capítulo.

Na prática, entretanto, utiliza-se sempre a combinação dessas duas técnicas. Pequenos trechos lineares programados explicitamente, ou "estados", são alternados por regras de transição. Os estados garantem um mínimo de controle ao programador, enquanto que as regras possibilitam alguma imprevisibilidade inteligente. Quando bem equilibrada, a superposição de regras e estados pode criar comportamentos artificiais emergentes complexos.

Um preconceito comum sobre softwares é o de que eles se comportam de modo estritamente previsível, de acordo com um projeto. O programador planeja a seqüência de instruções que compõe o software e, sob esse ponto de vista, teria controle total sobre sua criação. Qualquer desvio do projeto inicial só poderia ser atribuído a um erro de programação. Afinal, por mais complexo que seja o programa, ele não passa de uma seqüência de instruções que deve ser obedecida à risca pelo processador, característica que, aparentemente, não deixaria espaço para surpresas.

De fato, todo resultado apresentado por um software está prévia e completamente definido por sua seqüência de instruções. Mas antecipar com precisão o comportamento final de um programa não é tão simples como possa parecer: em determinadas aplicações, previsões tornam-se absolutamente inviáveis e desenvolvimentos inesperados emergem da contínua interação entre regras – o que pode comprometer o trabalho do programador.

5.2. Comportamento Emergente

Sistemas complexos, formados por agentes capazes de se comunicar entre si, ainda que não disponham de qualquer comando hierárquico, podem gerar ordem e complexidade inesperadas a partir da repetição de regras básicas. Tal fenômeno é conhecido como emergência. Exemplos clássicos de sistemas complexos emergentes são o formigueiro e o cérebro, ambos compostos pela interação maciça entre numerosos agentes relativamente limitados: formigas e neurônios.

Por sua construção igualmente baseada em agentes e regras, games são sistemas de software especialmente apropriados à emergência. Alguns exemplos já clássicos são as criações de Will Wright, *Sim City* (Maxis, desde 1989) e *The Sims* (Maxis, desde 2000). Planejado para lançamento em 2007, *Spore*, do mesmo autor, promete revolucionar os sistemas de criação de personagens e comportamentos[14].

Games atuais são desenvolvidos com técnicas de orientação a objetos. Essas técnicas dividem o programa em subsistemas independentes, e é a interação entre esses objetos que constrói a funcionalidade do todo. Assim como ocorre nos games, alguns sistemas de software buscam exatamente a emergência para simular comportamentos complexos que não poderiam ser programados de forma explícita. Usando como exemplo o programa StarLogo, software modelador de comportamentos emergentes escrito por Mitch Resnick, a descrição de Steven Johnson para tais sistemas é precisa:

> Sistemas como StarLogo não são anarquias absolutas: eles obedecem a regras que definimos previamente, mas essas regras governam apenas os micro-motivos. O macro-comportamento é outra história. Não se controla isso diretamente. Tudo o que podemos fazer é configurar as condições que, cremos, tornarão possível tal comportamento. Então, apertamos 'play' e vemos o que acontece. (JOHNSON, 2001, p. 169) [15]

De fato, na programação de comportamento de personagens, o controle do programador limita-se às regras básicas de interação entre objetos, sejam eles personagens ou outros elementos do game. O único modo de saber ao certo qual o impacto dessas regras no comportamento macro é rodar o programa e assistir a seus desdobramentos. Macro-comportamentos exibidos por personagens são emergentes por

[14] Palestra de Will Wright na Game Developer's Conference, 2005 (disponível em Google Vídeo, 20/11/2006).

[15] Systems like StarLogo are not utter anarchies: they obey rules that we define in advance, but those rules only govern the micromotives. The macrobehaviour is another matter. You don't control that directly. All you can do is set up the conditions that you think make that behavior possible. Then you press play and see what happens. (JOHNSON, 2001, p. 169)

não serem explicitamente programados ou previsíveis, mas é importante lembrar que tais comportamentos estão solidamente codificados nas regras originais, ainda que o programador responsável não os possa prever.

O algoritmo mais comum para criação de comportamento artificial de personagens é a máquina de estados finitos. Nele, uma série de estados lineares simples (por exemplo: andar, esperar, correr, atirar, morrer, etc.) são chaveados por regras de transição. Assim, por exemplo, um personagem que está no estado "esperar" pode mudar para os estados "andar" ou "correr" desde que um objetivo válido apareça em seu campo de visão. Comportamentos simples, como perseguir outros personagens ou fugir deles, são derivados dos estados disponíveis e das regras de transição. É importante notar também que as máquinas de estados finitos não se limitam a personagens. Uma porta, por exemplo, pode ser representada por quatro estados (fechada, abrindo, aberta, fechando), que são disparados de acordo com os comandos do interator e com o estado atual da porta. Entretanto, a interação entre personagens normalmente resulta em comportamentos mais complexos, muitas vezes indesejáveis.

Para ilustrar a aplicação prática desses algoritmos, bem como a emergência de novos comportamentos a partir de estados e regras, segue uma descrição parcial de dois estados e suas transições, utilizados em personagens de *Incidente em Varginha* (soldados).

Estado 1: Sair de linha visual quando sob ataque

> Quando atacado, o personagem deve mover-se até que não exista linha visual entre ele e seu oponente. Esse estado simula a procura por abrigo e, ao mesmo tempo, o movimento torna o personagem um alvo mais difícil, mesmo enquanto ele permanece em linha visual com o oponente.

Estado 2: Procurar linha visual quando atacando

> Quando atacando, o personagem deve mover-se até que exista linha visual entre ele e seu oponente. Esse estado simula a busca por posições de ataque viáveis, evitando, por exemplo, que o personagem procure inutilmente atingir seu oponente atirando contra uma parede ou coluna.

Separadamente, os dois estados podem ser considerados como programação explícita, não emergente. Com freqüência, entretanto, esses estados sobrepõem-se: o personagem

ataca enquanto é atacado. As transições entre estados podem criar comportamentos diversos, não explícitos na programação, que emergem dos estados básicos, das transições entre estados e das condições de interação. Por exemplo:

- Prioridade para o estado 1 com período de execução de 0.1 segundos, resulta em comportamento covarde. O personagem esconde-se ao primeiro sinal de ataque. Imediatamente após sair da linha visual, o estado 2 leva-o a deixar o esconderijo para onde retorna um décimo de segundo após. E assim sucessivamente, enquanto o personagem estiver simultaneamente atacando e sob ataque. Na prática, o personagem oscila rapidamente à beira de um ponto de abrigo, como uma esquina ou uma coluna, de modo inverossímil. Cabe comentar que a inverosimilhança refere-se apenas à imitação de um personagem real, uma vez que o comportamento é adequado às (bem como conseqüência das) regras do mundo virtual.
- Prioridade para o estado 1 com período de execução de 4 segundos resulta em comportamento prudente, mas verossímil. O personagem ataca por alguns segundos e busca abrigo. Volta a atacar e a buscar abrigo, num ciclo contínuo enquanto a superposição de estados perdurar.
- Prioridade para o estado 2 com período de execução de 0.1 segundos: comportamento agressivo. O personagem simplesmente ataca, não procurando abrigo. Caso seu oponente fuja, saindo da linha visual, procura imediatamente segui-lo, de acordo com o estado 2.

Levando-se em conta ainda que as próprias transições podem ser modificadas durante o desenrolar do game, esses dois estados simples podem criar comportamentos complexos e interessantes. Soldados que atacam de posições protegidas, nunca totalmente expostos, que podem se acovardar repentinamente, ou tornarem-se ainda mais agressivos quando atingidos. Esses comportamentos decorrem dos dois estados e das transições escolhidas, mas dependem também da ação do oponente. Por exemplo, enquanto não for atacado, o personagem não buscará abrigo, independentemente das transições escolhidas. Todos esses fatores tornam os comportamentos emergentes difíceis de prever e de projetar.

Dependendo do número de estados (que totalizam várias dezenas em games comerciais) e das variações de transição, a complexidade dos comportamentos aumenta

tanto que é praticamente impossível prever precisamente a reação de um personagem. Mudanças mínimas nas regras de transição podem gerar alterações significativas no comportamento geral. A solução seria a simulação exaustiva das reações dos personagens, para cada pequena alteração nos algoritmos de controle. Isso é inviável para personagens com grande número de estados, pois as combinações possíveis são muito numerosas e, normalmente, difíceis de reproduzir sob ambiente de testes. Resta ao desenvolvedor aplicar os testes viáveis e esperar pelo melhor.

Outros fatores do ambiente virtual também influenciam o comportamento de personagens. Como exemplo, consideremos a própria linha de visão citada acima. De forma geral, para saber se um personagem enxerga seu oponente, o programa deve traçar uma linha matemática entre os olhos do personagem e a cintura do oponente, seu ponto central. Se essa linha for interrompida por qualquer objeto sólido (parede, coluna, outro personagem, etc.), assume-se que o oponente não está visível; caso contrário, o oponente estará visível. Esse algoritmo funciona muito bem na grande maioria das situações, mas há exceções. Imaginemos, por exemplo, que exista um corrimão exatamente à frente da cintura do oponente. Nesse caso, ainda que o oponente esteja em grande parte exposto, a linha de visão do personagem será interrompida e este não o enxergará. Por outro lado, se o oponente também for um personagem controlado pelo computador, ele enxergará seu antagonista normalmente e o atacará, pois a linha matemática que define o campo visual parte da altura de seus olhos, que estão desimpedidos. Mesmo atacado, o primeiro personagem não buscará abrigo, pois sua linha de visão continua interrompida, condição estabelecida pelo programador do algoritmo como indicador de que se está em lugar seguro. Ao contrário, guiado pelo estado 2, o personagem procurará uma boa posição de tiro, andando de um lado para o outro na tentativa de desobstruir a linha de visão. Situações similares são observáveis mesmo em superproduções como *Half-Life* (Valve, 1998) e *Medal of Honor* (EA, 2002).

Existem soluções para o problema específico apresentado acima. Ele é gerado não pelas transições ou estados, mas pelo sistema simplificado de simulação de visão. Se a linha matemática de sondagem fosse dirigida aos olhos do oponente (e não à sua cintura), ela não seria obstruída nesse caso específico. Mas o seria em outros, potencialmente mais críticos. Por exemplo, uma pequena tabuleta entre os personagens e à altura de seus olhos impediria que qualquer deles enxergasse o outro. Outra solução seria aumentar a resolução da visão simulada, acrescentando uma segunda linha matemática de sondagem que ligasse os olhos do personagem aos de seu oponente. A

combinação de duas linhas em ângulos diferentes evitaria grande parte dos problemas de obstrução, mas dobraria o custo de processamento do sistema de visão. Ainda assim, ocorreriam problemas: o oponente estaria abrigado quando exatamente atrás de um poste de pequeno diâmetro que, mesmo deixando-o exposto em grande parte, obstruiria as duas linhas de visão. Talvez pudéssemos utilizar quatro linhas, acrescentando uma para cada ombro. Ou seis, com mais duas para os pés. Mas e se o oponente estivesse atrás do poste e de lado? Acrescentaríamos mais duas linhas, para os extremos do peito e das costas? E se apenas os braços estivessem visíveis?

Aumentar a resolução da simulação é possível. Mas o custo de processamento aumenta rapidamente e pode comprometer todo o sistema. Pode-se argumentar que o poder de processamento cresce constantemente e que, eventualmente, simulações precisas de processos físicos ou orgânicos serão possíveis. Ainda que isso aconteça, bons games não se baseiam na simulação precisa, mas na consistência da realidade virtual apresentada. Criar algoritmos na base da força bruta, tentando prever cada situação possível, dificulta a aplicação de conceitos de orientação a objetos e de máquinas de estados finitos. Por outro lado, algoritmos elegantes, com relativamente poucos estados e regras de transição mais simples, tendem a gerar resultados mais consistentes e emergentes. Cabe ao desenvolvedor conhecer as características do meio digital, utilizando-as a seu favor ou subvertendo-as.

Em 1996, durante o desenvolvimento do game *Incidente em Varginha* (Perceptum, 1998), pude testemunhar alguns exemplos de comportamentos emergentes indesejáveis, embora fascinantes. Um dos mais complexos dizia respeito aos soldados inimigos do game. Tentando reforçar o comportamento de grupo, incluí na máquina de estados finitos desses personagens a seguinte regra: se o soldado estivesse dentro de uma faixa de distância determinada de outro soldado, ele deveria primeiro aproximar-se do companheiro para apenas depois atacar o jogador. A intenção era simplesmente agrupar os personagens antes de atacar, mas não era exatamente isso o que ocorria. Um dos soldados voltava as costas ao inimigo para se aproximar do companheiro. O outro soldado, obedecendo à mesma regra, vinha na direção oposta. Quando ambos atingiam a distância mínima, o estado de ataque deveria ser disparado. Entretanto, com a inércia do movimento, os soldados ultrapassavam novamente a distância mínima estabelecida e, basicamente, trocavam de posições. O segundo soldado voltava então as costas ao jogador para obedecer a regra acima antes de atacar. E todo o processo repetia-se indefinidamente.

Resultavam daí estranhas coreografias entre esses personagens, algo que seria extremamente difícil de programar intencionalmente. Com poucos soldados, o comportamento beirava o ridículo, com personagens girando uns em torno dos outros em pequenos círculos. Porém, quando algumas dezenas de soldados estavam em ação, situação comum no game, a complexidade dos movimentos era desconcertante. Eles giravam lentamente em várias figuras circulares – de dois, quatro ou oito personagens – que interagiam entre si formando padrões ainda mais complexos. A sincronização emergente de movimentos levava vários personagens a emitir a mesma fala ao mesmo tempo, criando coros de vozes que proferiam as limitadas frases dos soldados de forma assustadora, como se participassem de alguma dança ritual.

Por mais fascinante que fosse, é óbvio que esse comportamento de grupo não era desejável para tais personagens. Após semanas de experimentações – tanto para explorar as possibilidades do comportamento como para entendê-lo parcialmente –, uma pequena alteração na regra foi suficiente para que tudo voltasse ao planejado. Mas não foi possível evitar uma nítida sensação de perda ao efetuar a correção.

Dado o caráter imprevisível dos comportamentos emergentes, não é possível planejá-los a partir de diretrizes básicas. Tendo apenas as regras locais como interface, ao programador caberia apenas recorrer ao método da tentativa e erro, conforme ilustrado acima. Mas essa limitação pode ser eliminada com a inclusão de um laço de realimentação (*feedback*) entre comportamentos e regras.

5.3. Seleção Artificial

Difundidos pela pesquisa de John Holland nas décadas de 60 e 70, os algoritmos genéticos são especialmente aptos a esse tipo de solução. A pesquisa de Holland tinha dois objetivos principais: contribuir para a compreensão dos processos de adaptação natural e projetar sistemas artificiais que apresentassem propriedades similares a sistemas naturais.

No lugar de planejar e escrever um programa, imagine que simplesmente sorteássemos instruções ao acaso para formá-lo. Cada número sorteado definiria uma instrução específica (o que é muito conveniente, pois as instruções de um programa não passam de valores numéricos interpretados por um processador). Definimos dessa maneira um genoma muito simples para o programa: uma seqüência de variáveis que,

ao assumir valores específicos (por exemplo, após o sorteio), representa um genótipo, um dos muitos programas possíveis dentro do genoma proposto.

Dificilmente um programa escrito assim faria qualquer sentido. Ainda menos provável seria que ele funcionasse como solução para um problema específico. Potencialmente, entretanto, existem seqüências de instruções permitidas pelo genoma que solucionariam diversos problemas diferentes. Convencionalmente, encontrar a seqüência correta de instruções para solucionar tais problemas seria função de um programador. É justamente aqui que um novo paradigma se faz presente:

> O novo paradigma baseia-se fortemente nas regras da seleção natural, procriando novos programas a partir de uma variada reserva de genes. As primeiras poucas décadas do software foram essencialmente criacionistas em sua filosofia – uma vontade toda-poderosa conclamava o programa à existência. Mas a próxima geração é profundamente darwiniana. (JOHNSON, 2001, p. 169)[16]

Seqüências de instruções inicialmente aleatórias podem evoluir até uma solução adequada. Para tanto, é necessário um laço de realimentação entre os resultados dos programas e as seqüências de valores que os compõem. Essa realimentação é executada pelo algoritmo genético, que nesse caso pode ser descrito de modo bastante simplificado como:

1. Gerar uma população de programas aleatórios.
2. Avaliar o resultado de cada programa (função de avaliação).
3. Caso exista resultado satisfatório, parar execução e apresentar solução.
4. Selecionar os programas mais eficientes e eliminar os restantes.
5. Gerar nova população de programas através de hibridização, mutação ou clonagem dos genótipos dos programas selecionados.
6. Retornar ao passo 2.

É interessante notar que não há nada de essencialmente diferente num algoritmo genético: ele é apenas uma seqüência de instruções, como qualquer outro programa. Mas sua aplicação é relativamente aberta. Como os resultados dos programas são selecionados segundo uma função de avaliação, é possível evoluir programas com

[16] "The new paradigm borrows heavily from the playbook of natural selection, breeding new programs out of a varied gene pool. The first few decades of software were essentially creationist in philosophy – an almighty power will the program into being. But the next generation is profoundly Darwinian." (JOHNSON, 2001, p. 169)

finalidades completamente diferentes por meio de uma simples troca da função de avaliação no mesmo algoritmo genético.

Funções de avaliação de comportamentos emergentes seriam, então, suficientes para refinar sucessivamente as regras básicas que os definem. As regras não seriam programadas, mas, antes, cultivadas por evolução artificial. Ao programador do sistema caberia apenas definir uma função de avaliação apropriada para o comportamento projetado. Ele não precisaria sequer compreender as soluções decorrentes.

5.4. Evoluindo Comportamentos

Em 1997, após a constatação das possibilidades emergentes das máquinas de estados finitos, trabalhei para que o comportamento artificial dos personagens de *Incidente em Varginha* fosse, ainda que parcialmente, evoluído através de algoritmos genéticos simplificados. Um programa próprio simulava, em 2D, diversos ambientes do game. Nessa simulação, apelidada de "Tanque", os personagens eram representados por meros ícones, mas o comportamento geral era regido pelas mesmas regras de suas respectivas máquinas de estados finitos. Comportamentos como perseguição, fuga, ataque, etc., eram simulados em tempo real sobre um mapa 2D. Um personagem específico, também controlado pelo computador, representava o jogador. No ecossistema virtual do "Tanque", o jogador exercia o papel de predador, perseguindo e atacando constantemente a população de inimigos, definindo assim a seleção dos mais aptos.

Os diversos estados permaneciam fixos, mas as variáveis e as regras de transição sofriam pequenas mutações aleatórias de geração para geração. Por exemplo, a função TropEscape[17], que executa o estado de fuga para os soldados, é definida pelo seguinte script:

```
ACTION TropEscape {
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          TropHide;
        SET MY.EACH_CYCLE,      CycleTropHide;
        SET MY.IF_HIT,          TropHit;
        SET MY.CAREFULLY,       1;
        SET MY.BERKELEY,        0;

        SET MY.SKILL4, 4;    // skill4 = state
        SET MY.TEXTURE, Trop1Tex;
        SET MY.DIST,  30;
        SET MY.SPEED, 0.4;
        SET MY.TARGET, REPEL;
```

[17] Ver anexo 2

}

No “Tanque”, o valor da variável my.speed, por exemplo, podia sofrer pequenas alterações aleatórias de uma geração para outra, aumentando ou diminuindo a velocidade do personagem no estado de fuga. O mesmo ocorria com outras variáveis e regras de transição.

Após um tempo de simulação pré-estabelecido, os personagens sobreviventes eram selecionados e serviam de base para uma nova geração. Não se tratava estritamente de um algoritmo genético, pois o próprio genoma era muito simplificado, representando apenas valores de um grupo parcial de variáveis, sem interferir diretamente no algoritmo do programa propriamente dito. Ainda assim, resultados extremamente interessantes foram obtidos.

A intenção era explorar comportamentos emergentes inesperados. Esse objetivo certamente foi atingido, mas poucos resultados foram práticos, pelos motivos que estão expostos a seguir. No geral, a experiência foi bem sucedida, servindo para enfatizar, na prática, a importância da função de avaliação nesse tipo de processo.

A primeira função de avaliação empregada foi simplesmente o tempo de sobrevivência dos personagens. Conforme descrito acima, os personagens que sobreviviam eram selecionados como base para a próxima geração. Isso resultou em comportamento extremamente adaptado, mas inútil: os personagens evitavam o confronto a qualquer custo. Tão logo entravam na linha de visão do jogador, os inimigos corriam em velocidade extrema até que ficassem protegidos por algum obstáculo, como uma coluna ou parede, e então paravam novamente. Ao serem encurralados, eles corriam em ziguezague, colidindo contra paredes e, eventualmente, escapavam ao jogador. A velocidade desses movimentos era tão elevada que tornava praticamente impossível qualquer tentativa de combater esses personagens que agiam como baratas velocíssimas. O comportamento resultante, embora inapropriado, satisfazia plenamente o quesito sobrevivência. Era necessária uma nova função de avaliação.

Uma segunda tentativa foi feita com a seguinte função de avaliação: maximizar tanto a sobrevivência como os danos infringidos ao jogador. Após várias dezenas de gerações, resultaram personagens que se escondiam quase como os anteriores, mas com uma diferença fundamental: enquanto corriam para um abrigo, viravam-se repentinamente e acertavam o jogador de modo quase infalível. A variável que controlava a precisão do

tiro foi alterada no processo, e isso resultava em tiros certeiros, mesmo em movimento. Novamente, a função de avaliação estava completamente atendida, mas o resultado não era o esperado.

Funções mais complexas foram então desenvolvidas, enquanto determinados parâmetros, como velocidade máxima e precisão, foram retirados do processo evolutivo. Uma função que apresentou resultados interessantes foi a maximização, não do período de sobrevivência de um único personagem, mas da sobrevivência de um grupo de personagens. Daí resultaram comportamentos de grupo relativamente sofisticados, especialmente perceptíveis nos personagens Drakalis. Dependendo apenas de dentes e garras, sem armas para agir a distância, os Drakalis desenvolveram surpreendentes táticas de caça em grupo. Freqüentemente, um ou dois elementos pareciam fugir timidamente do jogador, enquanto conduziam-no para o restante do grupo, que permanecia escondido. O jogador via-se então cercado, sem grandes possibilidades de defesa.

Apesar dos sucessos ocasionais, o "Tanque" apresentava um problema sério: ele evoluía personagens quase invencíveis. Obter personagens extremamente adaptados foi um excelente resultado de pesquisa, mas inimigos assim tornariam o game desinteressante. Gerar personagens mais verossímeis, por outro lado, exigiria funções de avaliação relativamente complexas e, possivelmente, menos objetivas. A solução adotada em 1997 foi, simplesmente, parar o processo evolutivo antes que a adaptação à função de avaliação fosse excessiva, limitando o número de gerações de personagens. As variáveis geradas eram, então, arredondadas para valores com um dígito significativo (por exemplo: 0.372 tornava-se 0.4), o que também reduzia a eficiência da adaptação.

A concepção do "Tanque" baseou-se no trabalho *Evolving Virtual Creatures*, de Karl Sims (SIMS, 1994). Nele, criaturas artificiais apresentam comportamentos emergentes em ambiente simulado. Tanto a morfologia das criaturas como suas redes neurais, responsáveis pelo acionamento da musculatura, são geradas automaticamente por algoritmos genéticos. O corpo das criaturas é formado por blocos unidos por articulações, sobre as quais atuam músculos virtuais, e sensores. O número de blocos, suas posições relativas e dimensões, as contrações da musculatura e suas reações aos dados coletados pelos sensores são evoluídos por algoritmos genéticos. Diferentes funções de avaliação direcionam a evolução das criaturas a comportamentos específicos, como nadar, andar, saltar ou perseguir um determinado ponto no espaço 3D. Por

exemplo, a função de avaliação para o comportamento "nadar" examina a velocidade da criatura na simulação de líquido. As criaturas que atingem maiores velocidades são selecionadas e seus códigos genéticos servem de base para uma nova geração de criaturas, através de hibridização, mutação ou clonagem. O processo é repetido até que a velocidade atinja um valor estipulado.

Karl Sims implementou seu programa num computador Connection Machine CM-5. O resultado final, apresentado em vídeo[18] de 1994, mostra criaturas que se movimentam de forma orgânica: algumas delas nadam agitando uma cauda, outras fazem uso de diversas nadadeiras, outras ainda oscilam o corpo em movimento senoidal, como cobras. As funções de avaliação para os outros comportamentos também geraram resultados interessantes. Determinada criatura anda sem pernas, apenas agitando um apêndice à sua frente que, por inércia, desloca o corpo principal em movimentos oscilatórios, numa estratégia similar à utilizada para deslocar móveis pesados sem a ajuda de rodas.

As criaturas não foram planejadas no sentido tradicional. Morfologias e comportamentos evoluíram para atender às funções de avaliação. É importante entender o que isso significa. As regras que definem o movimento foram vinculadas a uma estrutura genética, que nada mais faz além de permitir uma ampla gama de combinações de regras diferentes – regras cegas, sem objetivos pré-definidos e, em sua imensa maioria, inúteis. Ao programador coube definir esse código genético (que, por sua vez, define as regras) e a função de avaliação. Regras aleatórias e suas respectivas seqüências de movimentos evoluíram em várias gerações, através de hibridização e mutação do código genético, selecionadas pela função de avaliação específica. E, nesse processo, soluções emergiram. O mesmo código genético básico resultou em regras adaptadas para nadar, andar, saltar e perseguir, dependendo apenas da função escolhida – que avaliou, respectivamente, a velocidade no líquido, a velocidade no solo, a altura de salto ou o tempo para atingir uma distância mínima do objetivo.

Como vimos, a aplicação de conceitos de emergência e evolução no desenvolvimento de softwares apresenta-se como possível solução para a criação de comportamentos complexos baseados em regras. Enquanto o genoma digital permite ampla gama de regras diferentes, a função de avaliação propicia ao algoritmo genético os meios para selecionar as regras que gerem comportamentos mais eficientes. Esse laço cibernético, criado pelo algoritmo entre o comportamento final e o genoma das regras,

[18] Disponível em http://web.genarts.com/karl/evolved-virtual-creatures.html, em 20/11/2006.

elimina a necessidade de se programar as próprias regras. Nesse sentido, a codificação genética pode ser considerada um meta-programa. Ela define todo um conjunto de possíveis programas e cabe ao algoritmo genético selecionar exemplares bem sucedidos. As técnicas de hibridização e mutação permitem a seleção dos melhores candidatos sem que seja necessário examinar todas as possibilidades da codificação genética.

6. CONSIDERAÇÕES FINAIS

Talvez o comportamento artificial seja responsável pela característica que melhor distingue os games atuais de outras formas de expressão. Não que esse tipo de comportamento seja exclusividade do meio digital. Máquinas de calcular mecânicas são capazes de comportamento artificial programado, assim como o é o projeto das articulações dos *Bichos* de Lygia Clark, por exemplo. Mas comportamentos criados em meio digital, em função da grande variedade de combinações possíveis de regras e estados, são propícios à emergência. Também é possível evoluir comportamentos emergentes que atendam a objetivos pré-estabelecidos sem que eles sejam explicitamente programados, algo inédito em outros meios. Pelo que sabemos, tais características eram restritas, até então, a fenômenos naturais ligados à vida.

Por sua vez, o processo de evolução não precisa ser restrito ao comportamento de personagens. Algoritmos genéticos podem ser empregados no design emergente de outros componentes de um game.

Designs evoluídos através de algoritmos genéticos já são realidade, embora a maioria das aplicações concentre-se ainda em projetos militares e de exploração do espaço. A ST5 (Space Technology 5[19]), lançada pela NASA em 22 de março de 2006 para analisar a magnetosfera terrestre, foi a primeira missão a levar ao espaço objetos evoluídos artificialmente. Cada um dos três pequenos satélites lançados por um foguete Pegasus XL estava equipado com uma pequena antena de aparência estranha, assimétrica, completamente projetada por algoritmos genéticos. O design pouco usual provou ser altamente eficiente e a missão foi completada com sucesso em 20 de junho de 2006.

Algoritmos genéticos podem ser extremamente exigentes em termos de processamento, principalmente quando a função de avaliação é complexa. A evolução do design das antenas da ST5, por exemplo, exigiu mais de 10 horas de processamento paralelo num cluster de 80 PCs convencionais, em virtude da necessidade de simulação do ganho eletromagnético decorrente de cada formato diferente de antena. Outros designs podem exigir poder de processamento muito maior, como o projeto de um caça que apresente aerodinâmica supersônica e, simultaneamente, baixa reflexão a emissões

[19] http://nmp.jpl.nasa.gov/st5/ABOUT/about-index.html

de radar, por exemplo, pois tanto a aerodinâmica como a reflexão ao radar precisariam ser simuladas pela função de avaliação para cada novo formato gerado. Aplicações de design evolutivo em games devem restringir-se, pelo menos no início, a funções de avaliação relativamente simples, além de não serem executadas em tempo real. É bastante provável, entretanto, que novos hardwares extrapolem rapidamente essas limitações.

6.1. Automação de Modelagem e Texturização

Processos atuais de criação de conteúdo para games dependem de intervenção manual constante. A automação da geração de modelos utilizando processos disponíveis em outras áreas de modelagem 3D simplesmente não é viável. Como exemplo, a tecnologia de scanners a laser da empresa canadense Arius3D[20] gera automaticamente modelos fiéis de objetos reais – mas a resolução média de milhões de polígonos por objeto é incompatível com a renderização de imagens em tempo real. Como vimos nos capítulos anteriores, resoluções muito altas não apenas causam problemas de uso de memória e processamento, elas também afetam negativamente a própria aparência de objetos virtuais nos games 3D. Além disso, scanners 3D dependem da existência física de um padrão, exigência que nem sempre pode ser atendida para objetos e personagens de games. A redução automática da resolução de objetos digitalizados também não rende resultados aceitáveis, como pode ser comprovado nos softwares existentes para esse fim: a forma final é por vezes exageradamente alterada, ou o número de polígonos continua alto demais, ou, mais freqüentemente, ocorre uma combinação desses dois efeitos.

Mas seria possível que um sistema representasse, na geração de conteúdo para aplicações em tempo real, papel similar ao da fotografia na geração de imagens, produzindo automaticamente modelos com contagens apropriadas de polígonos?

Tal sistema é perfeitamente concebível. Partindo-se de modelo arbitrário, formado por triângulos definidos em código genético próprio, uma função poderia avaliar, simultaneamente, o número total de triângulos e a fidelidade do modelo à forma almejada. Tal fidelidade poderia ser estimada através da comparação de imagens do objetivo com renderizações do modelo, obviamente nos mesmos ângulos e parâmetros de câmera. Para tanto, bastaria uma série de imagens da forma a ser modelada, em

diversos ângulos e em fundo neutro. A função de avaliação renderizaria o modelo nos mesmos ângulos, sobre um fundo similar, e compararia os contornos de ambas. O algoritmo genético geraria uma população de modelos, selecionaria os mais adaptados, nova população seria gerada a partir de hibridização, mutação e clonagem, e o processo seria repetido até que determinado modelo apresentasse a fidelidade e o número de polígonos desejados. Alterações na tolerância à fidelidade e no número permitido de triângulos poderiam gerar modelos de diferentes resoluções, aplicáveis a diferentes níveis de detalhes do mesmo objeto virtual.

A comparação entre modelo e imagens da forma a ser modelada exige alguns cuidados adicionais. A cor do fundo, por exemplo, não deve ser repetida no interior da área a ser modelada, para evitar erros de interpretação de contornos[21]. A sobreposição dos dois contornos poderia, então, ser avaliada por mera comparação. O processo de modelagem manual empregado nos games da Perceptum também faz uso de comparação de contornos em diferentes ângulos a partir de concepções artísticas dos modelos, conforme ilustrado na Figura 16.

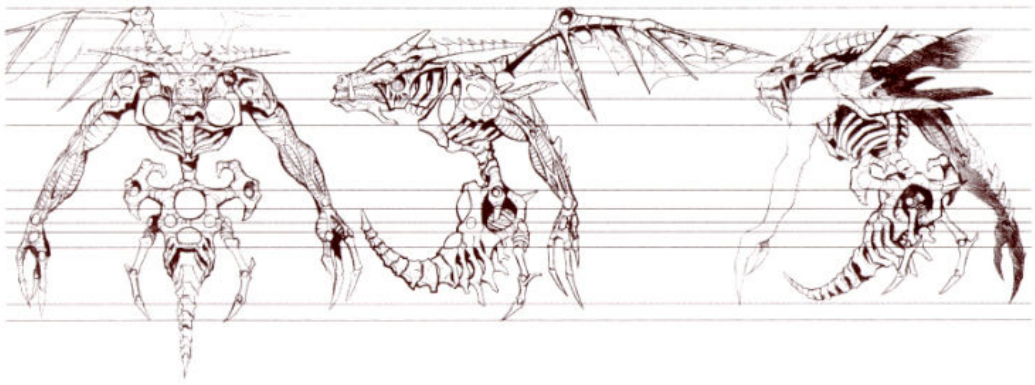


Figura 16: Drakalis (Perceptum, 2000) – arte de André Vazzios

No processo automatizado, se as áreas interiores aos contornos das duas imagens ocupassem exatamente os mesmos pixels, a avaliação seria máxima. Qualquer diferença entre áreas afetaria negativamente o resultado, que seria mínimo no caso de não existir qualquer sobreposição. Mas a quantidade de ângulos de projeção diferentes utilizados nessas comparações também precisaria ser levada em conta. Avaliar a fidelidade de um modelo apenas visto de frente e perfil, por exemplo, não garantiria que o contorno dos

[20] http://www.arius3d.com/
[21] Problema muito similar encontra a mesma solução na técnica de *chroma key*, empregada em cinema e vídeo para sobreposição de imagens de origens diferentes numa mesma cena.

ângulos intermediários fosse correto: o modelo poderia ser perfeitamente fiel nos dois ângulos avaliados, mas distorcido quando visto de outras direções. Experiências práticas com a modelagem manual de personagens do game *IV2: Sombras da Verdade*, sugerem um mínimo de 7 ângulos diferentes, tomados de 15 em 15 graus, na horizontal, além de 2 ângulos adicionais tomados na vertical, de 45 em 45 graus, num total de 9 imagens (Figura 17).

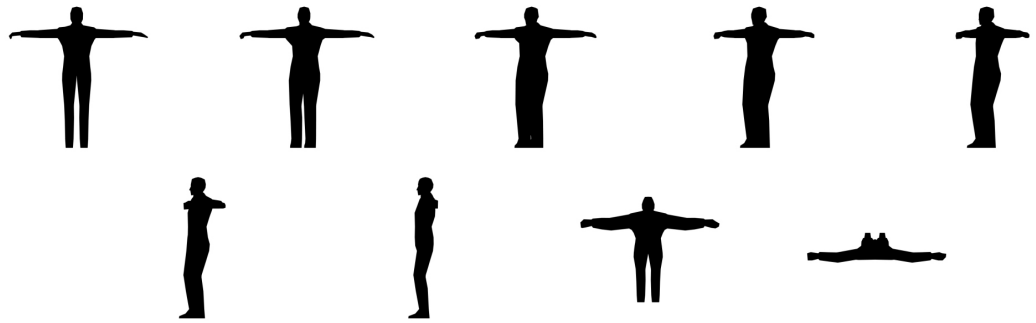


Figura 17

Para simplificar o processamento do algoritmo genético, formas simétricas poderiam ser definidas como tal no próprio código genético do modelo 3D. O genoma seria responsável apenas por metade do objeto – a outra metade seria espelhada – garantindo a simetria final.

Uma conseqüência interessante do método proposto seria a geração automática de texturas. Com base nas diversas imagens da forma a ser modelada (sejam elas fotografias de um objeto real, concepções artísticas ou renderizações de objetos em alta resolução), seria tarefa relativamente simples gerar texturas para o modelo 3D, já em resolução apropriada e corretamente mapeadas. Caso o objeto de base seja um modelo em alta resolução, com detalhes volumétricos de superfície e relevo, também é concebível que essas informações sejam extraídas como mapeamento de normais no modelo 3D resultante.

6.2. Evolução de Animações

Estratégia similar à empregada por Karl Sims no trabalho *Evolved Virtual Creatures* pode ser perfeitamente adaptada para dotar personagens virtuais de animações convincentes. Para tanto, os personagens precisariam de uma estrutura de articulações e musculatura, com limitações de ângulos e forças definidas. Sensores realimentariam o

ângulo de cada articulação ao sistema de disparo da musculatura, exatamente como proposto por Karl Sims em seu artigo para a SIGGRAPH 94. A massa de cada segmento do corpo seria, então, levada em conta na simulação do movimento, e a seqüência de atuação da musculatura evoluiria através de algoritmos genéticos, gerando movimentos fisicamente convincentes.

Mas a função de avaliação seria, necessariamente, mais complexa. Avaliar simplesmente a velocidade final do personagem, estratégia bem sucedida em *Evolved Virtual Creatures*, poderia eventualmente gerar movimentos de andar e correr. Entretanto, quedas, rolagens e saltos atenderiam ao mesmo requisito básico de velocidade final. Levando-se em conta o processamento exigido para a simples manutenção do equilíbrio de um personagem, as últimas opções seriam, sem dúvida, os resultados mais prováveis.

Uma solução possível é a minimização da energia utilizada. Um personagem que cai, por exemplo, pode conseguir deslocar-se rapidamente num primeiro momento, mas terá necessariamente de gastar muito mais energia para erguer-se e continuar em movimento. A minimização da energia necessária pode ser uma alternativa interessante para gerar movimentos complexos baseados em *key-frames*[22], automatizando um dos processos mais trabalhosos da animação tradicional. Entre duas ou mais posições pré-definidas, o algoritmo genético animaria o personagem de modo a minimizar a energia muscular empregada no movimento de transição.

O processo proposto exigiria processamento intenso, mas, além de automatizar grande parte do trabalho da animação manual, teria vantagens sobre o atual método de captura de movimentos:

- A animação gerada respeitaria a distribuição aparente de massa do personagem, algo que nem sempre é viável na captura de movimentos.
- As animações não ficariam restritas a seres reais. Diversas configurações não-naturais de membros poderiam ser animadas.
- O recurso da antecipação invisível, descrito no terceiro capítulo, poderia ser aplicado de forma procedural, antes de cada movimento de impacto.

[22] Na animação tradicional, *key-frames* são as poses principais, normalmente desenhadas pelo animador líder, que definem todas as características principais do movimento. As poses intermediárias, que representam a grande maioria dos desenhos, são então produzidas uma a uma pela equipe de animação.

- O custo de produção de movimentos – fator limitante para as pequenas empresas desenvolvedoras de games – tenderia a ser significativamente menor.

6.3. Autonomia

Vimos que o comportamento de personagens pode ser desenvolvido através de design evolutivo para adaptar-se a condições pré-determinadas. É possível que, no futuro, projetar tais sistemas comportamentais seja mais uma questão de cultivo que de programação. Grande parte do trabalho deve se concentrar na definição de resultados – através do projeto da função de avaliação apropriada – não na elaboração do software. O restante deverá ser dividido entre preparação e manutenção do ecossistema virtual que favoreça a evolução das criaturas digitais desejadas.

Personagens de games não são imitações de seres humanos. Como vimos nos capítulos anteriores, o visual de um personagem poligonal é construído por abstrações matemáticas, vértices e arestas que formam polígonos, e por texturas digitais que simulam superfícies. Seus movimentos são criados por deformação de polígonos e, ainda que digitalmente capturados, não guardam necessariamente relação com os movimentos reais de uma pessoa. Seus comportamentos são gerados por algoritmos emergentes, cujo domínio escapa normalmente ao próprio programador. Um personagem assim desenvolvido atua, aparentemente, como ser autônomo, interagindo com outros personagens, com interatores humanos e com o ambiente virtual ao seu redor.

Mas seria possível entender a emergência de comportamentos não intencionalmente programados como autonomia? Há sentido em se falar sobre autonomia quando o sistema que apresenta emergência foi criado por uma seqüência de instruções rígidas, um *script*, além de estar completamente contido em alguns milhares de bytes?

Os personagens desenvolvidos no "Tanque" sempre me pareceram autônomos. Tanto os mais bem sucedidos, que apresentavam comportamento de grupo, como aqueles que não se saíram tão bem, comportando-se como baratas ou ninjas velocíssimos, todos agiam de forma aparentemente autônoma e estavam muito bem adaptados às condições estabelecidas no algoritmo genético. Assistir pela primeira vez ao comportamento de tais personagens sempre me causava uma sensação curiosa. Apesar de ter definido

completamente as condições para sua criação, eu não tinha idéia do que eles fariam a seguir, nem, muitas vezes, de como alterações de parâmetros simples podiam gerar comportamentos tão sofisticados. Seria muito difícil programá-los de forma explícita. Havia também forte dependência das condições iniciais: alterações relativamente pequenas nos valores iniciais podiam gerar soluções bastante diferentes[23]. De fato, quando determinado comportamento não parecia adequado, eu freqüentemente alterava os parâmetros iniciais e refazia a simulação no Tanque. Mesmo quando os parâmetros iniciais permaneciam os mesmos, pequenas variações nas mutações randômicas levavam a resultados diferentes. Por outro lado, estava claro que o algoritmo não tinha qualquer opção além de gerar exatamente aqueles comportamentos. Era estranho observar comportamentos aparentemente autônomos sendo gerados por um script rígido e relativamente simples.

Talvez a questão seja o próprio conceito de autonomia. Existem níveis de autonomia em qualquer sistema cibernético. Um simples controlador de temperatura seria autônomo, pois corrige a temperatura ambiente sem intervenção de seus criadores. Mas, obviamente, não é esse nível de autonomia que nos interessa.

Sistemas cibernéticos ocupam níveis de complexidade. A principal divisão foi proposta por von Foerster, que apontou a diferença entre a cibernética "dos sistemas observados" e a cibernética "dos sistemas observantes", também chamada de "cibernética de segunda ordem" (von FOERSTER, 1974). Se a primeira dá conta de sistemas simples, como um controlador de temperatura, a segunda aplica-se a sistemas mais complexos, orientados a linguagem, incorporando explicitamente o observador na descrição do sistema e intercalando laços de controle.

Dessa forma, laços de controle básicos – por exemplo, os estados do soldado do exemplo do capítulo anterior – passam a ser incorporados num laço de controle superior – no mesmo exemplo, a máquina de estados finitos – que pode alterar objetivos do sistema como um todo. Os laços superiores englobam as ações do observador do sistema (no caso, o interator). Tais sistemas apresentam níveis de autonomia mais complexos e sua interação com o observador pode incluir linguagem simbólica, mesmo que de forma rudimentar.

[23] Tal dependência de condições iniciais é característica dos sistemas não lineares, ou caóticos. Nos sistemas lineares, pequenas alterações nos parâmetros de entrada geram alterações semelhantes no resultado. Nos sistemas não-lineares, alterações mínimas nos parâmetros de entrada podem causar alterações extremas no resultado.

Games que utilizam esses princípios – integração das ações do interator aos sistemas de comportamento de personagens, linguagem e interações entre personagens e observador, superposição de estados e emergência de comportamentos complexos a partir de regras mais simples – são sistemas cibernéticos de segunda ordem. Como tal, a autonomia de seus personagens pode apresentar – e com freqüência o faz – níveis que antes pareciam pertencer apenas ao reino animal, e ao homem em particular.

6.4. Emoções Artificiais

Mas, para que exista uma ligação empática entre interator humano e personagem, é necessário algo além do que já descrevemos: seres virtuais devem demonstrar, ainda que de forma simplificada, emoções. Segundo Couchot:

> Por que a autonomia e a simulação de emoções por computador apresentam tanto interesse nas relações homem-máquina? Por uma razão aparentemente simples, mas, na realidade, muito complexa sob um ponto de vista neuropsicológico. Nos múltiplos dispositivos digitais dedicados à comunicação homem-máquina e homem(s)-máquina(s)-homem(s), a autonomia do comportamento e a simulação de emoções manifestadas por artefatos virtuais com os quais o interator entra em contato – em tempo real – provocam nele uma sensação de *empatia* muito forte. Ao se identificar com um ser sintético dotado de comportamento não programado, experimentando emoções demonstradas em modo de apercepção – isto é, fora de toda representação simbólica de linguagem –, o interator conduz-se com tal ser como com um outro igual a ele próprio, sem contudo confundir-se com ele. Tudo se passa como se o computador se tornasse uma forma artificial – simulada – do outro. A relação homem-máquina tinge-se de intersubjetividade, ao passo que o computador é percebido como um elemento quase vivo da comunidade humana. E o diálogo homem-máquina é proporcionalmente facilitado e reforçado. (COUCHOT, catálogo *Emoção Art.ficial 3.0 – Interface Cibernética*, a ser publicado em 2007)[24]

As idéias apresentadas por Couchot são especialmente verificáveis no desenvolvimento de games, pois a simulação de emoções é condição fundamental para a criação de personagens virtuais convincentes.

---

[24] Pourquoi l'autonomie et la simulation des émotions par l'ordinateur présentent-elles autant d'intérêt dans la relation homme-machine? Pour une raison apparemment simple mais en réalité fort complexe d'un point de vue neuropsychologique. Dans les multiples dispositifs numériques dédiés à la communication homme-machine et homme(s)-machine(s)-homme(s), l'autonomie du comportement et la simulation des émotions manifestées par les artéfacts virtuels avec lesquels l'interacteur entre en contact — en temps réel — provoque chez ce dernier un très fort sentiment d'*empathie*. En s'identifiant à un être de synthèse doté d'un comportement non programmés, en ressentant les émotions qu'il affiche sur le mode de l'*aperception* — c'est-à-dire hors de toute représentation symbolique langagière —, l'interacteur se conduit avec lui comme avec un autre soi-même, sans toutefois se confondre avec. Tout se passe finalement comme si l'ordinateur devenait une forme artificielle — simulée — d'autrui. La relation homme-machine se colore d'intersubjectivité. Tandis que l'ordinateur est ressenti comme un élément quasi vivant de la communauté humaine. Le dialogue homme-machine en est d'autant plus facilité et renforcé. (COUCHOT, catálogo *Emoção Art.ficial 3.0 – Interface Cibernética*, a ser publicado em 2007)

Não há um modo simples para simular tais emoções. O processo exige coordenação de animações corporais e faciais, em tempo real, e está intimamente vinculado ao sistema de comportamento artificial do personagem. Esse é outro motivo da dificuldade de se desenvolver personagens humanos críveis. Assim como estamos naturalmente familiarizados com os movimentos humanos, conforme explicado no capítulo 3, também sabemos o que esperar de suas reações emocionais – e é extremamente difícil simulá-las de forma convincente, visto que dependem tanto de linguagem corporal como de expressões faciais, de entonação de voz e de *timing* preciso. Não é por outro motivo que a grande maioria dos personagens de games de ação pode ser facilmente classificada entre monstros, zumbis, alienígenas, robôs ou mesmo criaturas humanas caricatas. O modelo emocional de tais personagens é extremamente simples e relativamente fácil de ser codificado. Entretanto, personagens recentes que apresentam comportamento emocional muito mais complexo – como, por exemplo, Alyx Vance, do game *Half-Life 2* (Valve, 2004) – parecem comprovar que a sensação de empatia entre interator e personagens humanos virtuais é, enfim, possível.

Obviamente, a complexa gama de expressões emocionais necessárias tem seu custo: os personagens de *Half-Life 2* possuem um conjunto de 40 atuadores independentes apenas para a musculatura facial (número maior que o necessário para animar todo o corpo de um personagem humanóide). Há também o processamento adicional de comportamento, que deve disparar as animações faciais e corporais corretas nos momentos exatos – caso contrário, qualquer credibilidade da reação emocional pode ser facilmente perdida. O desenvolvimento de comportamento artificial com simulação de emoções ainda está em seus primeiros passos, mas provavelmente compensará o alto custo de recursos de processamento e memória: personagens que demonstrem emoções, capazes de gerar empatia com o interator, poderão conduzir um enredo interativo de forma convincente.

A correta utilização de modelagem 3D, texturização e animação é importante para que recursos do sistema não sejam desperdiçados por mero desconhecimento das características do meio digital e, especificamente, dos games 3D. Recursos adicionais de processamento e memória sempre serão necessários para a evolução de comportamento artificial, para a simulação de emoções e, futuramente, para enredos realmente interativos. Creio que sejam esses, nessa ordem, os próximos desafios de desenvolvimento de games 3D.

7. REFERÊNCIAS BIBLIOGRÁFICAS

7.1. Publicações

ABRASH, Michael. **Zen of Graphics Programming: The Ultimate Guide to Writing Fast PC Graphics**. Scottsdale: Coriolis Group, 1994, 750 p.

AHEARN, Luke. **3D Game Textures: Create Professional Game Art using Photoshop**. Oxford: Focal Press, 2006, 369 p.

ANDERS, Peter. **Envisioning cyberspace: Designing 3D electronic spaces**. New York: McGraw-Hill, 1999, 228 p.

ANDERSON, Greg *et al*. **More Tricks of the Game Programming Gurus**. Indianapolis: Sams Publishing, 1995, 1000 p.

COAD, Peter; NICOLA, Jill. **Object Oriented Programming**. Upper Saddle River: Pearson Education, 1993, 583 p.

COUCHOT, Edmond. **A Tecnologia na Arte: da Fotografia à Realidade Virtual**. Porto Alegre: Editora UFRGS, 2003, 320 p.

COUCHOT, Edmond; HILLAIRE, Norbert. **l'Art Numérique: comment la technologie vient au monde de l'art**. Paris, Editions Flamarion, 2003, 260 p.

CRAWFORD, Chris. *apud* SCHIESEL Seth. "Redefining the Power of the Gamer". **The New York Times**. New York, 7 jun. 2005, Arts.

CUTTING, J. E. "Blowing in the wind: Perceiving structure in trees and bushes". **Cognition**, v. 12, n. 1, pp. 25-44, 1982.

CUTTING, J. E. "Generation of synthetic male and female walkers through manipulation of a biomechanical invariant". **Perception**, v. 7, n. 4, pp. 393-405, 1978.

CUTTING, J. E.; KOZLOWSKI, L. T. "Recognizing friends by their walk: Gait perception without familiarity cue". **Bulletin of the Psychonomic Society**, v. 9, n. 5, pp. 353-356, 1977.

CUTTING, J. E.; PROFFITT, D. R. "Gait perception as an example of how we may perceived events". In: **Intersensory perception and sensory integration**. WALK, R. D. e PICK, H. L. (ed.). New York: Plenum, 1981, pp. 249-273.

DE PAULA ASSIS, Jesus. “Roteiros em ambientes virtuais interativos”. **Cadernos da Pós-Graduação**, Campinas, v. 3, n. 1, pp. 93-110, 1999.

GAME Brasilis – Catálogo de jogos eletrônicos brasileiros. São Paulo: Senac, 2003.

GRAU, Oliver. **Virtual Art: From Illusion to Immersion**. Cambridge: Leonardo Books, 2003, 416 p.

HOLLAND, J. H. **Adaptation in natural and artificial system**. Ann Arbor: The University of Michigan Press, 1975, 228 p.

JENKINS, Henry. “Art Form for the Digital Age”. **Technology Review**. Cambridge: MIT, set./out. 2000.

JOHANSSON, G. “Visual motion perception”. **Scientific American**, v. 232, n. 6. pp. 76-88, 1975.

JOHANSSON, G. “Visual perception of biological motion and a model for its analysis”. **Perception and Psychophysics**, v. 14, n. 2, pp. 201-211, 1973.

JOHANSSON, G. **Configuration in event perception**. Uppsala: Almqvist & Wiksell, 1950.

JOHNSON, Steven. **Emergence: The Connected Lives of Ants, Brains, Cities, and Software**. New York: Scribner, 2001, 288 p.

KELMAN, Nic. **Jeux vidéo: L'art du XXIe siècle**. Paris: Assouline, 2005, p. 319.

LAMOTHE, Andre; RATCLIFF, John; TYLER Denise. **Tricks of the Game Programming Gurus**. Indianapolis: Sams Publishing, 1994, 768 p.

LAMPTON, Christopher. **Gardens of Imagination: Programming 3D Maze Games in C/C++**. Corte Madera: Waite Group Press, 1994, 500 p.

MACHADO, Arlindo. **Máquina e imaginário: o desafio das poéticas eletrônicas**. São Paulo: Edusp, 1993, 316 p.

MANOVICH, Lev. **The language of new media**. Cambridge: MIT Press, 2001, 352 p.

MINSKY, Marvin. **Computation: Finite and Infinite Machines**. Upper Saddle River: Prentice Hall, Inc., 1967, 317 p.

MURRAY, Janet H. **Hamlet on the Holodeck: The future of narrative in cyberspace**. New York: The Free Press, 1997, 326 p.

NEWMAN, James. **Videogames**. London: Routledge, 2004, 198 p.

PRADO, Gilbertto. **Arte Telemática: dos intercâmbios pontuais aos ambientes virtuais multiusuário**. São Paulo: Itaú Cultural, 2003, 125 p.

ROLLINGS, Andrew; MORRIS, Dave. **Game Architecture and Design**. Scottsdale: Coriolis Group, 2000, 742 p.

SIMS, Karl. "Evolving 3D Morphology and Behavior by Competition". In: BROOKS, R.; MAES, P. (ed.). **Artificial Life IV Proceedings**. Cambridge: MIT Press, 1994, pp. 28-39.

SIMS, Karl. "Evolving Virtual Creatures". In: SIGGRAPH 1994. **Computer Graphics Annual Conference Series**, jul. 1994, pp. 15-22.

ULLMAN, S. **The interpretation of visual motion**. Cambridge: MIT Press, 1979, 229 p.

VENTURELLI, Suzete. **Arte: espaço_tempo_imagem**. Brasília: Editora UnB, 2004, 186 p.

von FOERSTER, Heinz (ed.). **Cybernetics of Cybernetics**. Urbana: University of Illinois, 1974.

WILLIAMS, Lance. "Pyramidal Parametrics". In: SIGGRAPH 1983. **Proceedings of the 10th annual conference on Computer graphics and interactive techniques**. New York: ACM Press, jul. 1983. pp. 1-11.

WILLIAMS, Richard. **The Animator's Survival Kit: A Manual of Methods, Principles, and Formulas for Classical, Computer, Games, Stop Motion, and Internet Animators**. London: Faber & Faber, 2002, 352 p.

7.2. Obras digitais

Battlefield 1942, EA, 2002

Call of Duty 2, Infinity Ward, 2005

Delta Force, NovaLogic, 1998

Doom, Id Software, 1993

Far Cry, Crytek, 2004

Half-Life, Valve, 1998

Half-Life 2, Valve, 2004

Incidente em Varginha, Perceptum, 1998

IV2: Sombras da Verdade, Perceptum, inédito

Iracema Aventura, Perceptum, 2005

Quake, Id Software, 1996

Rainbow Six, Red Storm Entertainment, 1998

Scooter Challenge, Perceptum, 2001

Sim City, Maxis, 1989

Sonic the Hedgehog, Sega, 1991

Super Mario Bros, Nintendo, 1985

Super Mini Racing, Perceptum, 2001

Super Mini Racing Ice, Perceptum, 2006

The Sims, Maxis, 2000

7.3. Sites

Gamasutra: http://www.gamasutra.com/

Journal of Game Development: http://www.jogd.com/subguide.html

Game Developer Magazine: http://www.gdmag.com/

Game/AI: http://www.ai-blog.net/

The Game AI Page: http://www.gameai.com/

Casual Game Design: http://www.casualgamedesign.com/

Gamecultura: http://www.gamecultura.com.br/

ABRAGAMES: http://www.abragames.com.br/

GarageGames: http://www.garagegames.com/

3D Game Studio: http://www.3dgamestudio.com/

Genesis3D: http://www.genesis3d.com/

3DRad: http://www.3drad.com/

Geo-metriks: http://www.geo-metricks.com/

Perceptum: http://www.perceptum.com/

SouthLogic: http://www.southlogic.com/

Oniria: http://www.oniriasoft.com.br/

Délirus: http://www.delirus.com.br/

Espaço: http://www.hades2.com/

44 Bico Largo: http://www.44bicolargo.com.br/

Devworks: http://www.devworks.com.br/

Locz: http://www.locz.com.br/loczgames/

Jynx: http://www.jynx.com.br/

ANEXO 1: *GLOSSÁRIO*

Advergame: game produzido como veículo publicitário. Financiado por anunciantes, é normalmente distribuído de forma gratuita.

Algoritmo genético: ferramenta de design evolutivo baseada nos princípios da seleção natural. Capaz de gerar soluções de forma automática com base em codificação genética. O algoritmo gera uma população de candidatos e seleciona os mais aptos, de acordo com uma função de avaliação pré-definida. Nova população é gerada com base na hibridização, mutação ou clonagem dos códigos genéticos dos selecionados. O processo se repete até que uma solução adequada seja encontrada.

BSP Tree (Binary Space Partitioning Tree): estrutura de dados que resulta da subdivisão recursiva de um ambiente virtual construído em geometria CGS. Uma vez gerada, permite definir, em tempo real, quais regiões são visíveis de um ponto qualquer no espaço.

Captura de movimentos: processo de digitalização de movimentos de atores reais para posterior aplicação em personagens virtuais. A captura de movimentos possibilita que um personagem tenha suas articulações animadas exatamente de acordo com dados reais digitalizados, conferindo precisão e realismo razoáveis ao movimento, de forma quase automática.

Comportamento artificial: comportamento de personagens e objetos virtuais gerado por software. Simula ações ou tomadas de decisão e está intimamente ligado à interatividade nos games. Os resultados de diferentes comportamentos artificiais vão das ações mais simples (o abrir e fechar de uma porta, por exemplo) até simulações complexas (como decisões táticas ou estratégicas de personagens virtuais). Entre os algoritmos mais utilizados no comportamento artificial estão as máquinas de estados finitos.

CSG (Constructive Solid Geometry): geometria 3D construída unicamente por sólidos fechados e convexos. Utilizada na construção de ambientes de games que utilizam algoritmos de Portais ou BSP.

Emergência: geração de ordem e complexidade a partir da interação constante entre agentes que compõem um sistema.

Engine (ou Game Engine): componente de software utilizado em games e outras aplicações gráficas em tempo real. Programa responsável principalmente pela geração de imagens, pode incluir o gerenciamento de sons, animações e física.

Fog: técnica que reproduz em ambientes virtuais efeitos semelhantes à neblina real. Basicamente o algoritmo de fog mescla uma cor com os pixels da imagem, de forma progressiva, dependendo de sua distância ao observador.

Fustrum: volume geométrico que representa o campo visual do observador num ambiente 3D. É a projeção imaginária da tela no ambiente virtual e forma uma pirâmide truncada, limitada por dois planos de corte, um próximo e outro distante.

Mapeamento de texturas: coordenadas que definem a orientação de aplicação de textura sobre os polígono que constituem um modelo 3D.

Mapeamento de normais: em sua estrutura de dados, a textura de mapeamento normal é uma imagem comum, mas as componentes R, G e B são interpretadas como coordenadas de vetores normais, não como pixels. É possível então calcular como a luz é refletida em cada elemento da matriz e conferir uma impressão muito realista de relevo a uma superfície virtualmente plana.

Mapeamento MIP: técnica utilizada para reduzir erros de *aliasing* gerados por texturas. Emprega várias versões da mesma textura em dimensões diferentes. Quanto mais próxima a distância da textura ao ponto de observação, maior a dimensão da versão exibida.

Máquina de estados finitos: algoritmo utilizado na criação de comportamento artificial. Seleciona entre uma série de estados pré-definidos com base em regras de transição e informações sobre o ambiente virtual.

Níveis de detalhe: técnica de otimização que emprega várias versões de um modelo 3D com resoluções (contagem de polígonos) diferentes.

Pixel: unidade básica de imagens digitais. Contração de *picture element*, representa um pequeno retângulo de cor na matriz digital da imagem.

Polígonos: blocos construtores dos modelos 3D poligonais. Game Engines e placas aceleradoras normalmente lidam com a forma mais simples de polígono, o triângulo (qualquer polígono pode ser subdividido em triângulos).

Portais: regiões de passagem entre subdivisões BSP. Basicamente, são áreas que permitem a visualização de diferentes regiões do espaço virtual.

Raycaster: algoritmo que permite a construção de imagens de aparência tridimensional a partir de um mapa bidimensional. Trata-se de uma simplificação de outro algoritmo, o raytracer, utilizado ainda hoje por diversos programas de síntese de imagens. Numa época em que os PCs não dispunham da tecnologia de placas gráficas aceleradas, o raycaster era uma das poucas soluções para a criação de ambientes tridimensionais em tempo real nessa plataforma.

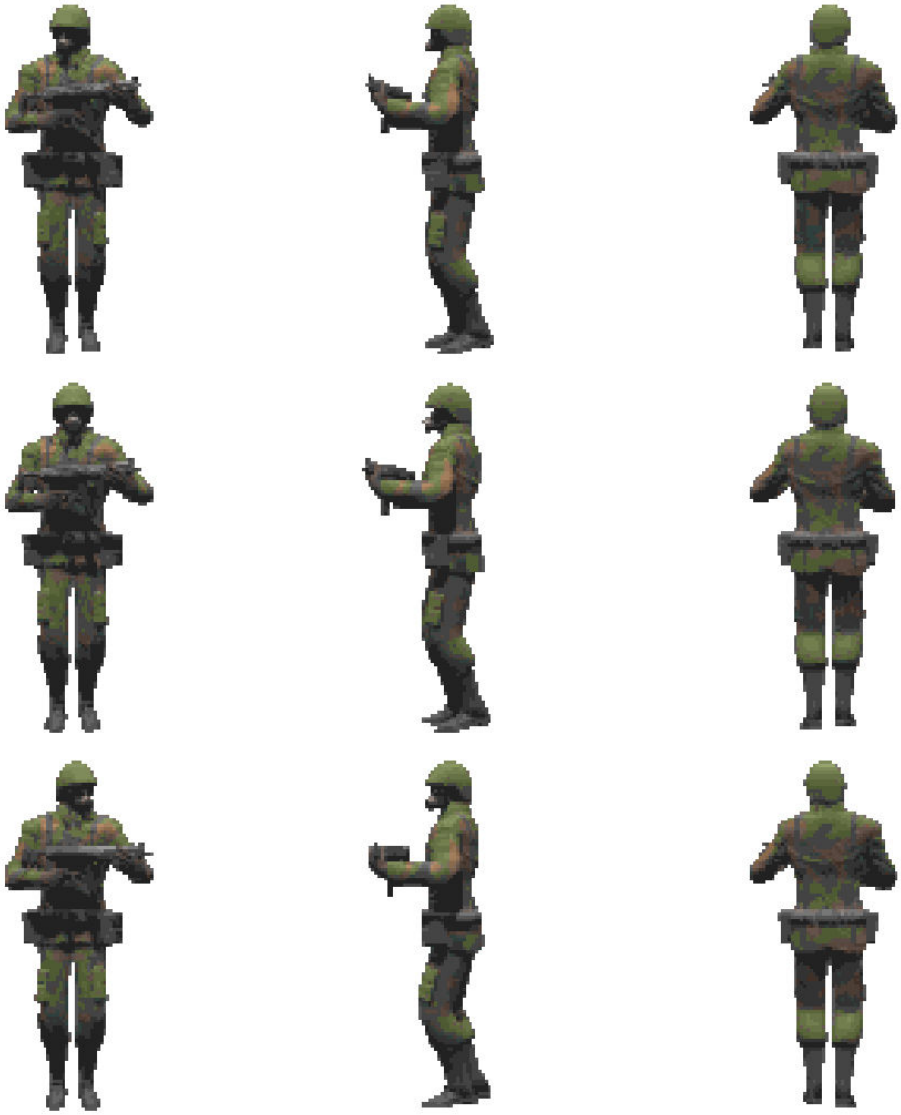
Raytracer: algoritmo para construção de imagens a partir de modelos tridimensionais. Seu modelo matemático traça raios de luz no sentido inverso, ou seja, do ponto de formação da imagem até as fontes de luz. Dessa forma apenas os raios de luz que contribuem para a formação da imagem são considerados. Trata-se de processo relativamente lento e, portanto, ainda não aplicável à criação de imagens em tempo real.

Renderização de imagens: processo que converte os modelos numéricos que compõem um ambiente virtual na matriz de pixels que será exibida como imagem.

Sprites: figuras criadas em mapa de bits. Normalmente representam personagens ou objetos nos ambientes virtuais que se utilizam do algoritmo raycaster ou em games 2D.

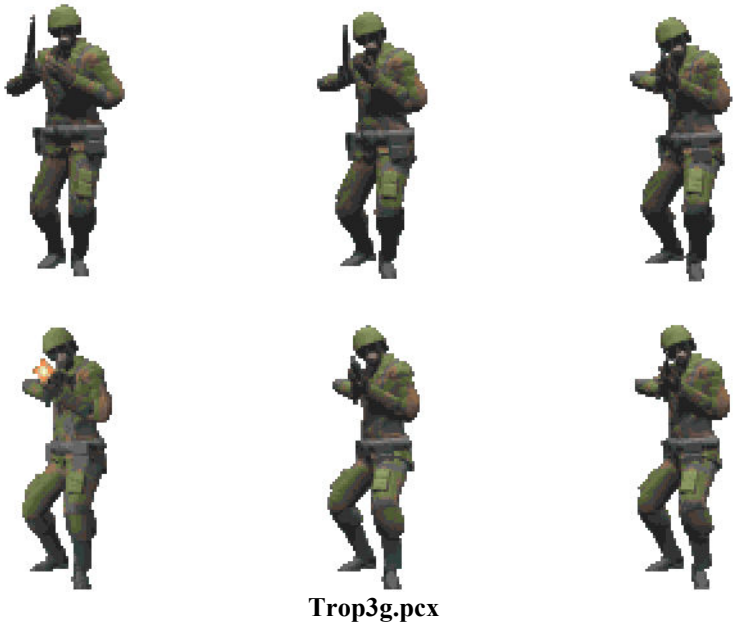
ANEXO 2: *DEFINIÇÃO COMPLETA DE PERSONAGEM*

Trooper (soldado)
Incidente em Varginha, 1997
Simulação 3D em sprites com animação em frames individuais
Máquina de Estados escrita em C-Script para Acknex game engine



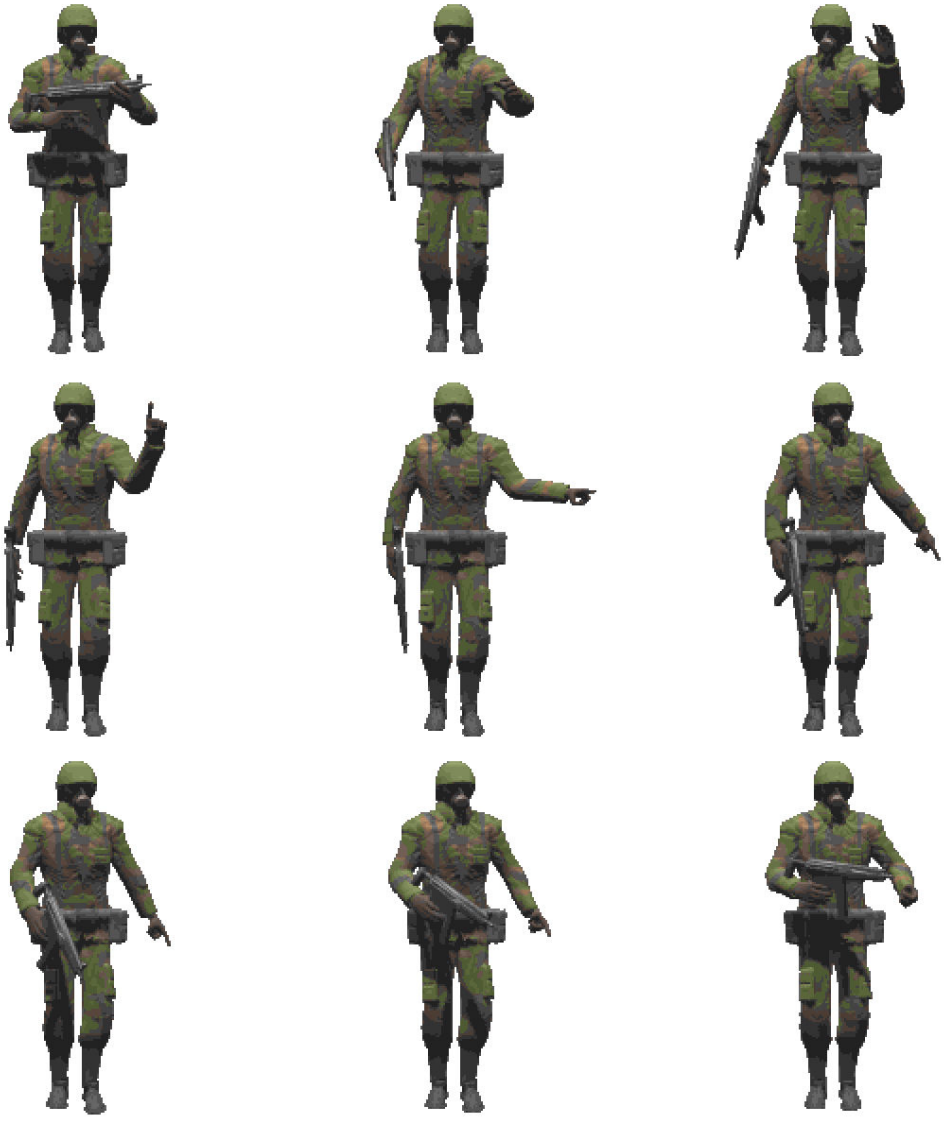
Trop0g.pcx

Trop1g.pcx

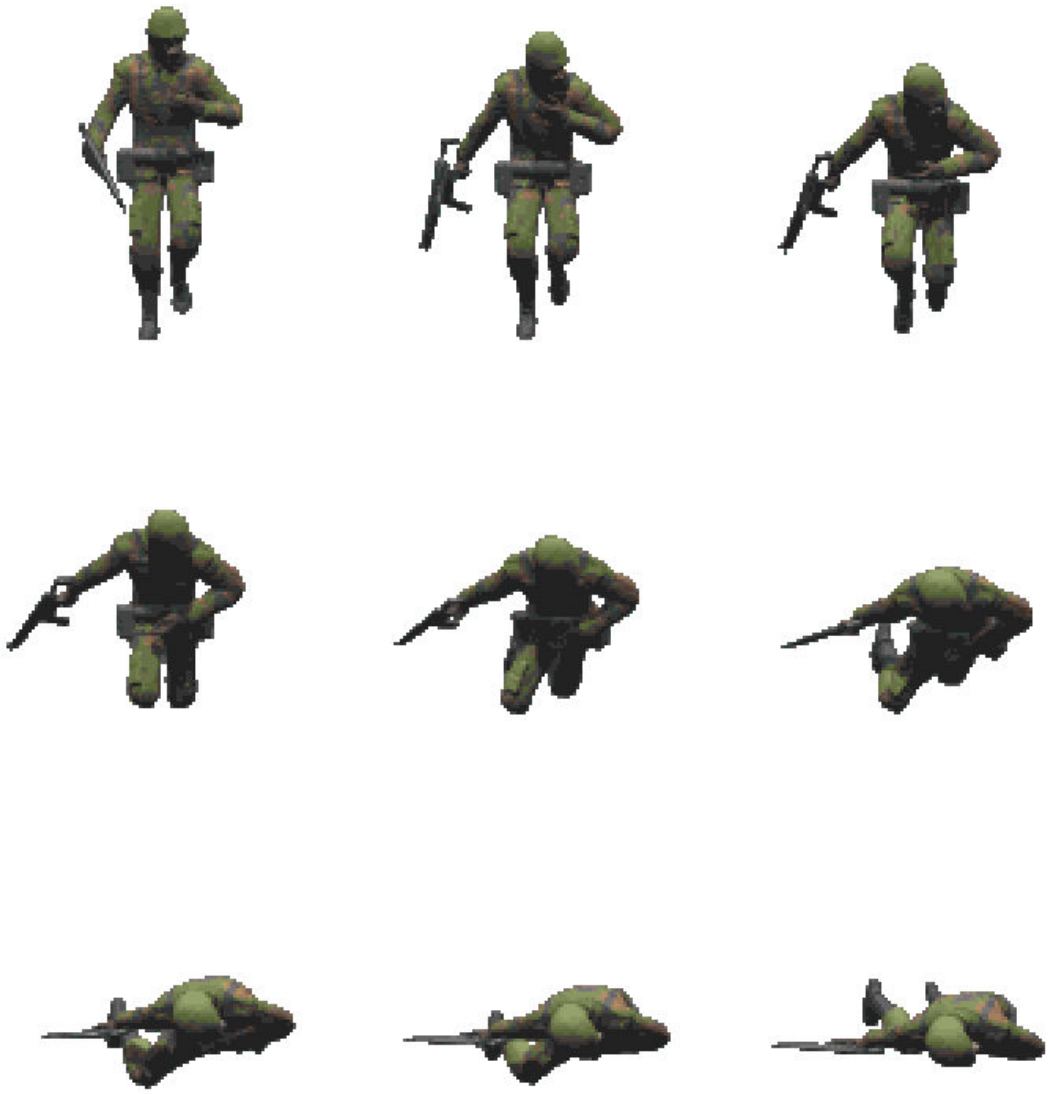


Trop3g.pcx

Trop5g.pcx



Trop7g.pcx



Trop8g.pcx

TropAg.pcx

```
////////////////////////////////////////////////////////////////
// Trooper Actor
// Copyright © 1997 Perceptum Informática Ltda
// All Rights Reserved
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
// Bitmaps
////////////////////////////////////////////////////////////////

BMAP Trop0001, <Trop0g.pcx>,  35,   5,  51, 111;
BMAP Trop0002, <Trop0g.pcx>, 153,   5,  55, 111;
BMAP Trop0003, <Trop0g.pcx>, 275,   5,  51, 110;
BMAP Trop0101, <Trop0g.pcx>,  35, 125,  51, 111;
BMAP Trop0102, <Trop0g.pcx>, 152, 125,  57, 111;
BMAP Trop0103, <Trop0g.pcx>, 275, 125,  51, 110;
BMAP Trop0201, <Trop0g.pcx>,  35, 245,  51, 111;
BMAP Trop0202, <Trop0g.pcx>, 154, 245,  53, 110;
BMAP Trop0203, <Trop0g.pcx>, 275, 245,  51, 110;

BMAP Trop1001, <Trop1g.pcx>,  38,  10,  45, 106;
BMAP Trop1002, <Trop1g.pcx>, 157,  10,  47, 104;
BMAP Trop1003, <Trop1g.pcx>, 271,  10,  59, 106;
BMAP Trop1004, <Trop1g.pcx>, 391,  10,  59, 107;
BMAP Trop1005, <Trop1g.pcx>, 518,  10,  45, 107;
BMAP Trop1101, <Trop1g.pcx>,  38, 125,  45, 110;
BMAP Trop1102, <Trop1g.pcx>, 163, 125,  35, 110;
BMAP Trop1103, <Trop1g.pcx>, 278, 125,  45, 109;
BMAP Trop1104, <Trop1g.pcx>, 392, 125,  57, 109;
BMAP Trop1105, <Trop1g.pcx>, 519, 125,  43, 109;
BMAP Trop1201, <Trop1g.pcx>,  38, 250,  45, 106;
BMAP Trop1202, <Trop1g.pcx>, 162, 250,  37, 105;
BMAP Trop1203, <Trop1g.pcx>, 272, 250,  57, 104;
BMAP Trop1204, <Trop1g.pcx>, 391, 250,  59, 106;
BMAP Trop1205, <Trop1g.pcx>, 519, 250,  43, 107;
BMAP Trop1301, <Trop1g.pcx>,  38, 365,  45, 110;
BMAP Trop1302, <Trop1g.pcx>, 163, 365,  35, 111;
BMAP Trop1303, <Trop1g.pcx>, 278, 365,  45, 111;
BMAP Trop1304, <Trop1g.pcx>, 392, 365,  57, 110;
BMAP Trop1305, <Trop1g.pcx>, 518, 365,  45, 109;

BMAP Trop3001, <Trop3g.pcx>,  36,  11,  49, 106;
BMAP Trop3101, <Trop3g.pcx>, 158,  14,  45, 104;
BMAP Trop3201, <Trop3g.pcx>, 280,  16,  41, 103;
BMAP Trop3301, <Trop3g.pcx>,  43, 137,  35, 102;
BMAP Trop3401, <Trop3g.pcx>, 162, 136,  37, 103;
BMAP Trop3501, <Trop3g.pcx>, 282, 136,  37, 103;

BMAP Trop5001, <Trop5g.pcx>,  25,  12,  71, 106;
BMAP Trop5002, <Trop5g.pcx>, 155,  13,  51, 103;
BMAP Trop5003, <Trop5g.pcx>, 269,  13,  63, 104;

BMAP Trop7001, <Trop7g.pcx>,  58,   8,  85, 185;
BMAP Trop7101, <Trop7g.pcx>, 258,   8,  85, 185;
BMAP Trop7201, <Trop7g.pcx>, 448,   8, 105, 186;
BMAP Trop7301, <Trop7g.pcx>,  53, 208,  95, 187;
BMAP Trop7401, <Trop7g.pcx>, 226, 208, 149, 187;
BMAP Trop7501, <Trop7g.pcx>, 444, 208, 113, 187;
BMAP Trop7601, <Trop7g.pcx>,  51, 408,  99, 187;
BMAP Trop7701, <Trop7g.pcx>, 250, 408, 101, 186;
```

```
BMAP Trop7801, <Trop7g.pcx>, 456, 408,  89, 185;

BMAP Trop8001, <Trop8g.pcx>,  33,   9,  55, 106;
BMAP Trop8101, <Trop8g.pcx>, 141,  17,  79,  98;
BMAP Trop8201, <Trop8g.pcx>, 255,  27,  91,  86;
BMAP Trop8301, <Trop8g.pcx>,  10, 169, 101,  63;
BMAP Trop8401, <Trop8g.pcx>, 133, 177,  95,  57;
BMAP Trop8501, <Trop8g.pcx>, 256, 187,  89,  47;
BMAP Trop8601, <Trop8g.pcx>,  19, 317,  83,  36;
BMAP Trop8701, <Trop8g.pcx>, 136, 320,  89,  32;
BMAP Trop9001, <Trop8g.pcx>, 253, 318,  95,  28;

BMAP TropA001, <TropAg.pcx>,  36,  21,  49,  94;
BMAP TropA002, <TropAg.pcx>, 161,  21,  39,  95;
BMAP TropA003, <TropAg.pcx>, 276,  21,  49,  95;
BMAP TropA004, <TropAg.pcx>, 388,  21,  65,  94;
BMAP TropA005, <TropAg.pcx>, 516,  21,  49,  93;
BMAP TropA101, <TropAg.pcx>,  33, 144,  55,  92;
BMAP TropA102, <TropAg.pcx>, 157, 144,  47,  90;
BMAP TropA103, <TropAg.pcx>, 271, 144,  59,  92;
BMAP TropA104, <TropAg.pcx>, 388, 144,  65,  93;
BMAP TropA105, <TropAg.pcx>, 515, 144,  51,  93;
BMAP TropA201, <TropAg.pcx>,  36, 261,  49,  94;
BMAP TropA202, <TropAg.pcx>, 161, 261,  39,  94;
BMAP TropA203, <TropAg.pcx>, 276, 261,  49,  93;
BMAP TropA204, <TropAg.pcx>, 388, 261,  65,  93;
BMAP TropA205, <TropAg.pcx>, 516, 261,  49,  93;
BMAP TropA301, <TropAg.pcx>,  37, 384,  47,  92;
BMAP TropA302, <TropAg.pcx>, 162, 384,  37,  91;
BMAP TropA303, <TropAg.pcx>, 272, 384,  57,  90;
BMAP TropA304, <TropAg.pcx>, 389, 384,  63,  92;
BMAP TropA305, <TropAg.pcx>, 516, 384,  49,  93;

////////////////////////////////////////////////////////////////
// Sounds
////////////////////////////////////////////////////////////////
SOUND Trop01Snd, <trop01.wav>; // "Ok Charlie Delta Tango..."
SOUND Trop02Snd, <trop02.wav>; // "Over there..."
SOUND Trop03Snd, <trop03.wav>; // "Get down!"
SOUND Trop04Snd, <trop04.wav>; // MP5 firirng
SOUND Trop05Snd, <trop05.wav>; // "Uh!"
SOUND Trop06Snd, <trop06.wav>; // "Help me over here!"
SOUND Trop07Snd, <trop07.wav>; // "This is a restricted area!"
SOUND Trop08Snd, <trop08.wav>; // death
SOUND Trop09Snd, <trop09.wav>; // "X-ray Tango! Come in, over"
SOUND Trop10Snd, <trop10.wav>; // "He's over there"
SOUND Trop11Snd, <trop11.wav>; // "Son of a..."
SOUND Trop12Snd, <trop12.wav>; // "Hey, scumbag!"

////////////////////////////////////////////////////////////////
// Animated Textures
////////////////////////////////////////////////////////////////
TEXTURE Trop0Tex {
        SCALE_XY        23,23;
        SIDES           4;
        CYCLES          4;
        BMAPS   Trop0001, Trop0101, Trop0201, Trop0101,
                Trop0002, Trop0102, Trop0202, Trop0102,
                Trop0003, Trop0103, Trop0203, Trop0103,
                Trop0002, Trop0102, Trop0202, Trop0102;
        DELAY   8, 8, 8, 8;
```

```
        RANDOM  0.7;
        MIRROR  0,0,0,1;
        FLAGS   CLIP;
}
TEXTURE Trop0aTex {
        SCALE_XY        23,23;
        SIDES           4;
        CYCLES          3;
        BMAPS   Trop0001, Trop0101, Trop0201,
                Trop0002, Trop0102, Trop0202,
                Trop0003, Trop0103, Trop0203,
                Trop0002, Trop0102, Trop0202;
        DELAY   4, 4, 4;
        MIRROR  0,0,0,1;
        RADIANCE 0.7;
        FLAGS   CLIP;
}
TEXTURE Trop1Tex {
        SCALE_XY        23,23;
        SIDES           8;
        CYCLES          4;
        BMAPS   Trop1001, Trop1101, Trop1201, Trop1301,
                Trop1002, Trop1102, Trop1202, Trop1302,
                Trop1003, Trop1103, Trop1203, Trop1303,
                Trop1004, Trop1104, Trop1204, Trop1304,
                Trop1005, Trop1105, Trop1205, Trop1305,
                Trop1204, Trop1304, Trop1004, Trop1104,
                Trop1203, Trop1303, Trop1003, Trop1103,
                Trop1202, Trop1302, Trop1002, Trop1102;
        DELAY   4, 5, 4, 5;
        MIRROR  0,0,0,0,0,1,1,1;
        FLAGS   CLIP;
}
TEXTURE Trop3Tex {
        SCALE_XY        23,23;
        SIDES           1;
        CYCLES          9;
        BMAPS   Trop3301, Trop3401,
                Trop3301, Trop3401,
                Trop3301, Trop3401,
                Trop3301, Trop3401,
                Trop3501;
   SOUND   Trop04Snd;
   SDIST   300;
   SVOL    0.4;
   SVDIST  25;
   DELAY   2, 2, 2, 2, 2, 2, 2, 3, 4;
   SCYCLES 0, 1, 0, 0, 0, 0, 0, 0, 0;
   FLAGS   CLIP;
   AMBIENT 0.2;
}
TEXTURE Trop3aTex {
        SCALE_XY        23,23;
        SIDES           1;
        CYCLES          3;
        BMAPS   Trop3001, Trop3101, Trop3201;

        DELAY   4, 3, 3;
        FLAGS   CLIP;
}
TEXTURE Trop5Tex {
```

```
        SCALE_XY        23,23;
        SIDES           4;
        CYCLES          1;
        BMAPS   Trop5001, Trop5002, Trop5003, Trop5002;
        MIRROR  0, 0, 0, 1;
        FLAGS   CLIP;
}
TEXTURE Trop5aTex {
        SCALE_XY        23,23;
        SIDES           4;
        CYCLES          1;
        BMAPS   Trop5001, Trop5002, Trop5003, Trop5002;
        MIRROR  1, 1, 1, 0;
        FLAGS   CLIP;
}
TEXTURE Trop7Tex {
        SCALE_XY        39,39;
        SIDES           1;
        CYCLES          9;
        BMAPS   Trop7001, Trop7101, Trop7201, Trop7301,
                Trop7401, Trop7501, Trop7601, Trop7701,
                Trop7801;
        DELAY   5, 5, 5, 5, 5, 5, 5, 5, 5;
        SOUND   Trop07Snd;
        SDIST   300;
        SVDIST  25;
        SCYCLES 0, 1, 0, 0, 0, 0, 0, 0, 0;
        FLAGS   CLIP;
}
TEXTURE Trop7aTex {
        SCALE_XY        39,39;
        SIDES           1;
        CYCLES          9;
        BMAPS   Trop7001, Trop7101, Trop7201, Trop7301,
                Trop7401, Trop7501, Trop7601, Trop7701,
                Trop7801;
        DELAY   5, 5, 5, 5, 5, 5, 5, 5, 5;
        MIRROR  1;
        SOUND   Trop07Snd;
        SDIST   300;
        SVDIST  25;
        SCYCLES 0, 1, 0, 0, 0, 0, 0, 0, 0;
        FLAGS   CLIP;
}
TEXTURE Trop8Tex {
        SCALE_XY        23,23;
        SIDES           1;
        CYCLES          8;
        BMAPS   Trop8001, Trop8101, Trop8201, Trop8301,
        Trop8401, Trop8501, Trop8601, Trop8701;
        DELAY   5, 4, 3, 3, 3, 3, 3, 3;
        SOUND    Trop08Snd;
        SCYCLES 0,1,0,0,0,0,0,0;
        SVOL 0.3;
        SDIST 200;
        SVDIST 25;
        FLAGS   CLIP;
}
TEXTURE Trop9Tex {
        SCALE_XY        23,23;
        SIDES           1;
```

```
        CYCLES          1;
        BMAPS   Trop9001;
        FLAGS   CLIP;
}
TEXTURE TropATex {
        SCALE_XY        23,23;
        SIDES           8;
        CYCLES          4;
        BMAPS   TropA001, TropA101, TropA201, TropA301,
                TropA002, TropA102, TropA202, TropA302,
                TropA003, TropA103, TropA203, TropA303,
                TropA004, TropA104, TropA204, TropA304,
                TropA005, TropA105, TropA205, TropA305,
                TropA204, TropA304, TropA004, TropA104,
                TropA203, TropA303, TropA003, TropA103,
                TropA202, TropA302, TropA002, TropA102;
        DELAY   4, 5, 4, 5;
        MIRROR  0,0,0,0,0,1,1,1;
        FLAGS   CLIP;
}

////////////////////////////////////////////////////////////////
// Actions
////////////////////////////////////////////////////////////////

ACTION TropTurn {
   RULE waitTime = 64 * RANDOM + 16;
   IF_BELOW RANDOM, 0.07;
      BRANCH TropListen;
   IF_ABOVE RANDOM, 0.95;
      RULE rightTurnTrop = (rightTurnTrop-1)*(rightTurnTrop-1);
   IF_MAX rightTurnTrop;
      BRANCH TurnRight;
   BRANCH TurnLeft;
}

ACTION TropBackoff {
        SET MY.EACH_TICK,       NULL;
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,   NULL;
        SET MY.IF_HIT,    TropHit;
        SET MY.CAREFULLY,   1;

        SET MY.SKILL4, 1;    // skill4 = state
        SET MY.TEXTURE, Trop1Tex;
        SET MY.SPEED, 0.3;
        SET MY.TARGET, REPEL;
        WAITT 64;
        SET MY.TARGET, FOLLOW;
        WAITT 2;
        IF_ABOVE MY.SKILL1, 5;
            END;
        BRANCH TropWait;
}

ACTION TropWait {
        SET MY.EACH_TICK,       NULL;
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,   CycleTropAttack;
```

```
        SET MY.IF_HIT,    TropHit;
        SET MY.CAREFULLY,   0;
        SET MY.GROUND,    0;

        SET MY.SKILL4, 1;    // skill4 = state
        SET MY.TEXTURE, Trop0Tex;
        SET MY.SPEED, 0.0;
        SET MY.TARGET, NULL;
}
ACTION TropWander {
        SET MY.EACH_TICK,       NULL;
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,   CycleTropAttack;
        SET MY.IF_HIT,    TropHit;
        SET MY.CAREFULLY,   1;
        SET MY.BERKELEY,   0;

   SET MY.SKILL4, 1;    // skill4 = state
        SET MY.TEXTURE, Trop1Tex;
        SET MY.SPEED, 0.3;
        SET MY.TARGET, BULLET;
}
ACTION TropSearch {
        SET MY.EACH_TICK,       NULL;
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,      CycleTropAttack;
        SET MY.IF_HIT,    TropHit;
        SET MY.CAREFULLY,   1;
        SET MY.BERKELEY,   0;

        SET MY.TEXTURE, TropATex;
        SET MY.SPEED, 0.3;
        SET MY.TARGET, BULLET;
}
ACTION TropHide {
        SET MY.EACH_TICK,       NULL;
        SET MY.IF_NEAR,         TropEscape;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,   NULL;
        SET MY.IF_HIT,    TropHit;
        SET MY.CAREFULLY,   1;
        SET MY.BERKELEY,   0;

        SET MY.SKILL4, 4;    // skill4 = state
        RULE MY.ANGLE = PLAYER_ANGLE + (RANDOM-0.5) * 2.4;
        SET MY.TEXTURE, Trop1Tex;
        SET MY.DIST,  20;
        SET MY.SPEED, 0.6;
        SET MY.TARGET, BULLET;
        WAITT 384;
        IF_BELOW MY.SKILL1, 9;
           BRANCH TropWander;
}

ACTION TropListen {
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,      CycleTropAttack;
        SET MY.IF_HIT,          TropHit;
```

```
        SET MY.CAREFULLY,  0;
        SET MY.BERKELEY,   0;

        SET MY.SKILL4,     0;    // skill4 = state
        SET MY.TEXTURE, Trop0Tex;
        SET MY.SPEED, 0.0;
        SET MY.TARGET, NULL;
        WAITT 120;
        IF_BELOW MY.SKILL1, 9;
           BRANCH TropWander;
}

ACTION TropWarning {
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,      TropBackoff;
        SET MY.IF_HIT,          TropHit;

        SET MY.SKILL4, 7;    // skill4 = state
        SET MY.TEXTURE, Trop7Tex;
   IF_BELOW RANDOM, 0.5;
         SET MY.TEXTURE, Trop7aTex;
   SET TROP_TEX, MY.TEXTURE;
   RANDOMIZE randomTrop, 1;
   IF_BELOW randomTrop, 0.2;
         GOTO overThere;
   IF_BELOW randomTrop, 0.4;
         GOTO getDown;
   SET TROP_TEX.SOUND, Trop07Snd;
   GOTO cont;
getDown:
   SET TROP_TEX.SOUND, Trop03Snd;
   GOTO cont;
overThere:
   SET TROP_TEX.SOUND, Trop02Snd;
   GOTO cont;
cont:
   SET MY.SPEED, 0.0;
   WAITT 32;
   IF_ABOVE MY.SKILL1, 5;
   END;
   RULE MY.ANGLE = MY.ANGLE + 2;
   IF_BELOW RANDOM, 0.5;
        RULE MY.ANGLE = MY.ANGLE - 2;
   BRANCH TropBackoff;
}

ACTION TropLookFor {
   IF_EQUAL MY.SKILL4, 10;
      END;
   IF_ABOVE MY.DISTANCE, 100;
      END;
   IF_EQUAL MY.SKILL4, 10;
      END;
   IF_EQUAL MY.TARGET, FOLLOW;
      END;
   IF_ABOVE MY.SKILL1, 5;
      END;
   SET MY.IF_NEAR,         NULL;
   SET MY.IF_FAR,          NULL;
   SET MY.EACH_CYCLE,      TropSearch;
```

```
   SET MY.IF_HIT,          TropHit;
   SET MY.CAREFULLY,       0;
   SET MY.BERKELEY,        0;
   IF_BELOW MY.DISTANCE,  40;
         SET MY.EACH_CYCLE, TropFollowAttack;

   SET MY.SKILL4, 10;    // skill4 = state
   SET MY.TEXTURE, TropATex;
   SET MY.SPEED, 0.0;

   SET MY.TARGET, BULLET;
}
ACTION TropFollow {
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,      CycleTropAttack;
        SET MY.IF_HIT,    TropHit;
        SET MY.CAREFULLY,   1;
        SET MY.BERKELEY,   0;

        SET MY.SKILL4, 1;    // skill4 = state
        SET MY.TEXTURE, Trop1Tex;
        SET MY.SPEED, 0.4;
        SET MY.TARGET, FOLLOW;
}
ACTION TropFollowAttack {
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,      CycleTropShoot;
        SET MY.IF_HIT,          TropHit;
        SET MY.CAREFULLY,       1;
        SET MY.BERKELEY,        0;

        SET MY.SKILL4, 2;    // skill4 = state
        SET MY.TEXTURE,         TropATex;
        SET MY.SPEED,           0.4;
        SET MY.TARGET,          FOLLOW;
}
ACTION TropFollowWarning {
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,      CycleTropWarning;
        SET MY.IF_HIT,          TropHit;
        SET MY.CAREFULLY,       1;
        SET MY.BERKELEY,        0;

   SET MY.SKILL4, 1;    // skill4 = state
        SET MY.TEXTURE,         Trop1Tex;
        SET MY.SPEED,           0.4;
        SET MY.TARGET,          FOLLOW;
}
ACTION TropAim {
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,      TropShoot;
        SET MY.IF_HIT,          TropHit;
        SET MY.CAREFULLY,       0;
        SET MY.BERKELEY,        0;

        SET MY.SKILL4, 3;    // skill4 = state
        SET MY.TEXTURE,         Trop3aTex;
```

```
        SET MY.SPEED, 0.0;
        SET MY.TARGET, FOLLOW;
        CALL TropShout;
}
ACTION TropShoot {
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.EACH_CYCLE,      CycleTropShoot;
        SET MY.IF_HIT,          TropHit;
        SET MY.CAREFULLY,       0;
        SET MY.BERKELEY,        0;

   SET MY.SKILL4, 3;    // skill4 = state
        SET MY.TEXTURE, Trop3Tex;
        SET MY.SPEED, 0.0;
        SET MY.TARGET, FOLLOW;
        SET SHOOT_SECTOR, 6.28;
        SET SHOOT_FAC,    tropShootFactor;
        SET SHOOT_RANGE,  100;
        SET SHOOT_Y,   0.0;
        SHOOT MY;
        IF_EQUAL HIT_DIST, 0;
           GOTO miss;
        SET UP_REGION, MY.REGION;
        IF_EQUAL HERE, UP_REGION.BELOW;
           GOTO miss;
        IF_EQUAL UP_REGION, HERE.BELOW;
           GOTO miss;
        SET PLAYER_RESULT, RESULT;
        BRANCH HitPlayerDelay;
        END;
miss:
        BRANCH TropFollowAttack;
}
ACTION TropEscape {
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          TropHide;
        SET MY.EACH_CYCLE,      CycleTropHide;
        SET MY.IF_HIT,          TropHit;
        SET MY.CAREFULLY,       1;
        SET MY.BERKELEY,        0;

        SET MY.SKILL4, 4;    // skill4 = state
        SET MY.TEXTURE, Trop1Tex;
        SET MY.DIST,  30;
        SET MY.SPEED, 0.4;
        SET MY.TARGET, REPEL;
}
ACTION TropDead {
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.IF_HIT,          NULL;
        SET MY.EACH_CYCLE,      NULL;

        SET MY.SKILL4, 9;    // skill4 = state
        SET MY.TEXTURE, Trop9Tex;
        SET MY.SPEED, 0.0;
        SET MY.TARGET, NULL;
        SET MY.PASSABLE, 1;
        SET MY.CAREFULLY,   0;
        SET MY.BERKELEY,   1;
```

```
}
ACTION TropDie {
   SET MY.SKILL1,     10;   // certify actor is dead
        SET MY.IF_NEAR,         NULL;
        SET MY.IF_FAR,          NULL;
        SET MY.IF_HIT,          NULL;
        SET MY.EACH_CYCLE,      TropDead;

        IF_NEQUAL MY.FLAG4, 0;
           CALL TropDropKey;
        SET MY.SKILL4, 8;    // skill4 = state
        SET MY.TEXTURE, Trop8Tex;
        SET MY.SPEED, 0.0;
        SET MY.TARGET, NULL;
}
ACTION TropImplode {
   RULE distX = (MY.X - EXPLOSION_CENTER.X)*(MY.X -
EXPLOSION_CENTER.X)+
      (MY.Y - EXPLOSION_CENTER.Y)*(MY.Y - EXPLOSION_CENTER.Y);
   SQRT distX, distX;
   RULE distZ = MY.HEIGHT - EXPLOSION_CENTER.HEIGHT;
   IF_ABOVE distX, 10;
      BRANCH BeamReact;
   SET MY.PASSABLE, 1;
   SET MY.CAREFULLY,   0;
   SET MY.IF_NEAR,         NULL;
   SET MY.IF_FAR,          NULL;
   SET MY.IF_HIT,          NULL;
   SET MY.EACH_CYCLE,      VanishStop;

   SET MY.SKILL4, 8;    // skill4 = state
   SET MY.TEXTURE, Trop0aTex;
   SET MY.SPEED, 0.0;
   SET MY.TARGET, NULL;
   IF_NEQUAL MY.FLAG4, 0;
       CALL TropDropKey;
}

ACTION TropHit {
   IF_EQUAL SHOOT_FAC, 0;
      GOTO obstacle;
   IF_EQUAL HIT, MY;
      GOTO hit;
   IF_EQUAL EXPLOSION_ON, 0;
      GOTO obstacle;
   RULE distX = (MY.X - EXPLOSION_CENTER.X)*(MY.X -
EXPLOSION_CENTER.X)+
      (MY.Y - EXPLOSION_CENTER.Y)*(MY.Y - EXPLOSION_CENTER.Y);
   SQRT distX, distX;
   IF_ABOVE distX, 25;
      GOTO obstacle;
hit:
   IF_NEQUAL EXPLOSION_ON, 2;
      GOTO cont;
   IF_EQUAL MY.VISIBLE, 1;
      BRANCH TropImplode;
cont:
   ADD MY.SKILL1, SHOOT_FAC;
   IF_ABOVE MY.SKILL1, 5;
       GOTO die;
   IF_BELOW RANDOM, 0.05;
```

```
        GOTO die;
    SET TROP_TEX,  MY.TEXTURE;
    SET MY.IF_NEAR, NULL;
    SET MY.IF_FAR, NULL;
    SET MY.IF_HIT, NULL;
    IF_ABOVE RANDOM, 0.7;
       GOTO sonOfa;
    SET MY.TEXTURE, Trop5Tex;
       PLAY_SOUND Trop05Snd, 0.2, MY;
    GOTO wait;
sonofa:
    SET MY.TEXTURE, Trop5aTex;
    PLAY_SOUND Trop11Snd, 0.2, MY;
wait:
    WAITT 4;
    SET MY.IF_NEAR, TropEscape;
    SET MY.IF_FAR, TropHide;
    SET MY.IF_HIT, TropHit;
    SET MY.TEXTURE, TROP_TEX;
    IF_ABOVE MY.SKILL1, 4;
        BRANCH TropHide;
    IF_NEQUAL MY.SKILL4, 4;
        BRANCH TropFollowAttack;
    END;
die:
    SET MY.SKILL1, 10;
    BRANCH TropDie;
    END;
obstacle:
    BRANCH TropTurn;
}
ACTION TropDropKey
{
     RULE Key2Tng.X = My.X;
     RULE Key2Tng.Y = My.Y;
     SET Key2Tng.GROUND,    1;  // ensures proper region for LOCATE
     RULE Key2Tng.HEIGHT = My.FLOOR_HGT-0.01;

     SET Key2Tng.INVISIBLE, 0;
     LOCATE Key2Tng;
     SET Key2Tng.GROUND,    0;
}

ACTION CycleTropAttack {
    CALL LocateActor;
    IF_EQUAL SHOT_SOUND_ON, 0;
        goto cont;
    CALL TropLookFor;
cont:
    IF_NEQUAL MY.VISIBLE, 0;
        GOTO distance;
    IF_BELOW MY.DISTANCE, 25;
        GOTO distance;
    SET SHOOT_SECTOR, 3;
    SET SHOOT_FAC,    0;
    SET SHOOT_RANGE,  200;
    SHOOT MY;
    IF_EQUAL HIT_DIST, 0;
        END;
    SET UP_REGION, MY.REGION;
    IF_EQUAL HERE, UP_REGION.BELOW;
```

```
      END;
  IF_EQUAL UP_REGION, HERE.BELOW;
      END;
distance:
  IF_NEQUAL MY.FLAG2, 0;  // flag 2 = player's got a gun
      GOTO attack;
  IF_NEQUAL GUN_ON, 0;
      GOTO attack;
  IF_BELOW MY.DISTANCE, 20;
      BRANCH TropFollowWarning;
  END;
attack:
  SET MY.FLAG2, 1;   // flag 2 = player's got a gun
  CALL TropTalk;
  IF_BELOW MY.DISTANCE, 50;
      BRANCH TropFollowAttack;
}
ACTION CycleTropHide {
   CALL LocateActor;
   SET SHOOT_SECTOR, 6.283;
   SET SHOOT_FAC,    0;
   SET SHOOT_RANGE,  200;
   SHOOT MY;
   IF_EQUAL HIT_DIST, 0;
           GOTO stop;
   SET MY.SPEED, 0.6;
   SET MY.TARGET, BULLET;
   END;
stop:
   SET MY.SPEED, 0;
   SET MY.TARGET, NULL;
}

ACTION CycleTropShoot {
   CALL LocateActor;
   IF_ABOVE MY.SKILL1, 9;
       BRANCH TropDie;
   SET SHOOT_SECTOR, 6.28;
   SET SHOOT_FAC,    0;
   SET SHOOT_RANGE,  100;
   SHOOT MY;
   IF_EQUAL MY.VISIBLE, 1;
       GOTO shoot;
   IF_EQUAL HIT_DIST, 0;
       GOTO attack;
   SET UP_REGION, MY.REGION;
   IF_EQUAL HERE, UP_REGION.BELOW;
       GOTO attack;
   IF_EQUAL UP_REGION, HERE.BELOW;
       GOTO attack;
shoot:
   SET SHOT_SOUND_ON, 1;
   SET shotSecCount,  0;
   IF_ABOVE RANDOM, 0.8;
        BRANCH  TropAim;
   BRANCH  TropShoot;
attack:
   BRANCH  TropFollowAttack;
}

ACTION CycleTropWarning {
```

```
   CALL LocateActor;
   IF_NEQUAL GUN_ON, 0;
       BRANCH TropFollowAttack;
   IF_ABOVE MY.DISTANCE, 10;
       GOTO cont;
   SET MY.EACH_CYCLE,      NULL;
   BRANCH TropWarning;
cont:
   IF_ABOVE MY.DISTANCE, 40;
   BRANCH TropWander;
}

ACTION TropTalk {
   RANDOMIZE randomTrop, 1;
   IF_ABOVE randomTrop, 0.9;
       GOTO talk2;
   IF_ABOVE randomTrop, 0.8;
       GOTO talk1;
   END;
talk1:
   PLAY_SOUND Trop01Snd, 0.3, MY;
   END;
talk2:
   PLAY_SOUND Trop09Snd, 0.3, MY;
}

ACTION TropShout {
   RANDOMIZE randomTrop, 1;
   IF_ABOVE randomTrop, 0.8;
       GOTO hey;
   IF_ABOVE randomTrop, 0.35;
       GOTO overThere;
   PLAY_SOUND Trop03Snd, 0.3, MY;
   END;
overThere:
   PLAY_SOUND Trop10Snd, 0.3, MY;
   END;
hey:
   PLAY_SOUND Trop12Snd, 0.4, MY;
}

////////////////////////////////////////////////////////////////
// Actor
////////////////////////////////////////////////////////////////
ACTOR TropAct {
      TEXTURE   Trop1Tex;
      HEIGHT   -0.05;
      DIST      20;
      SPEED     0.3;
      FLAGS     CAREFULLY,FRAGILE;
      IF_HIT    TropHit;
      EACH_TICK TropWait;
}
ACTOR Trop1ct {
      TEXTURE   Trop0Tex;
      HEIGHT   -0.05;
      SPEED     0.0;
      FLAGS     CAREFULLY,FRAGILE;
      IF_HIT    TropHit;
}
```

ANEXO 3: *BREVE HISTÓRICO DA PRODUÇÃO BRASILEIRA DE GAMES 3D*

Este breve relato apresenta os oito anos iniciais da produção brasileira de games 3D[25]. O foco é o trabalho das quatro principais empresas dedicadas a esse tipo de desenvolvimento no Brasil: Perceptum, SouthLogic, Espaço e Greenland.

Criada por mim e por Odair Gaspar em outubro de 1995, a Perceptum Software Ltda. nasceu de experiências profissionais anteriores com o desenvolvimento de softwares em tempo real para aplicações de automação industrial.

A Perceptum lançou o primeiro Game 3D brasileiro de ação em primeira pessoa, *Incidente em Varginha*, em setembro de 1998, de forma independente, devido à virtual ausência de publicadoras interessadas em games nacionais na época. O game atingiu sucesso apenas relativo no mercado nacional, mas gerou vendas significativas na Europa e na Ásia sob o título *Alien Anarchy*; foi encartado em revistas especializadas nos Estados Unidos, além de ter sido vendido em alguns países da América do Sul, com o título *Misión Alien*. A distribuição das versões internacionais do game foi realizada pela Midas Interactive Entertainment.

O desdobramento mais insólito de *Incidente em Varginha* teve início com uma consulta do U.S. Army Special Forces Command (Comando das Forças Especiais do Exército Americano), feita por email em 11 de novembro de 1998. A mensagem sondava o possível uso do game no treinamento de soldados das forças especiais[26] como alternativa ao game *Doom*, da ID Software, utilizado na época para treinamento dos SEALS, grupo de elite da marinha americana. Descrevia ainda algumas missões de treinamento hipotéticas, incluindo ataques a centros de radar e a veículos de lançamento de mísseis SCUDs – curiosamente, todas as missões tinham como cenário o Iraque. Respondi pessoalmente ao email, esclarecendo que a Perceptum não era uma empresa americana, mas brasileira, e que um game muito similar ao solicitado estava sendo lançado nos Estados Unidos pela Novalogic: *Delta Force*. Não recebi qualquer retorno,

[25] Existem games brasileiros importantes que não se utilizam de tecnologia 3D em tempo real (o pioneiro *Amazônia*, de Renato Degiovani, e o game de estratégia *Outlive*, da Continuum, entre eles). Devido às especificidades do desenvolvimento de games 3D, tema deste trabalho, o histórico limita-se a esse tipo de produção.

mas uma versão especial de *Delta Force* foi posteriormente desenvolvida para o treinamento das forças de elite do exército americano.

Em 1999, a Perceptum foi contratada pelo Instituto Itaú Cultural para o desenvolvimento da exposição virtual *Imateriais*. Seis grandes ambientes virtuais multiusuários foram criados com a tecnologia desenvolvida para o game *IV2: Sombras da Verdade*, seqüência não publicada de *Incidente em Varginha*. A exposição virtual *Imateriais* contava ainda com comunicação por voz em tempo real, algo que só viria a se popularizar em games online alguns anos depois.

Um demo de *IV2*, otimizado para as novas instruções do processador Pentium III, foi selecionado pela Intel para apresentação no evento *Pentium III Preview Day*, pré-lançamento do novo processador que ocorreu em San Jose, Califórnia (EUA), em 17 de fevereiro de 1999. Todo o trabalho de aperfeiçoamento do código foi realizado dentro do Programa de Desenvolvedores Intel.

Outra empresa brasileira participante do mesmo programa foi a Jack in the Box Computing, hoje conhecida como SouthLogic Studios. Fundada por Christian Lykawka em julho de 1996, a empresa lançou um game 2D, *Guimo*, em 1997. Em 1998, Christian ingressou na incubadora de novas empresas da Universidade Federal do Rio Grande do Sul e iniciou o desenvolvimento de um *engine* 3D proprietário, o *Aspen Engine*. Na mesma época, Adriano Ledur, Ricardo Coimbra da Rocha e Gustavo Goedert uniram-se a Christian. *Aquarius*, um protótipo de game desenvolvido com o *Aspen Engine*, foi também selecionado pela Intel e exibido no *Pentium III Preview Day*, em 1999. Impressionado com o trabalho da equipe, um dos engenheiros de software da Intel, Chris Kastensmidt, então consultor internacional do Programa de Desenvolvedores, deixou a empresa americana no mesmo ano para unir-se à SouthLogic Studios. Sua participação como diretor comercial, especialmente na negociação junto a empresas do mercado norte-americano, seria importantíssima no futuro da empresa gaúcha.

Em maio de 2001, a Perceptum finalizou o desenvolvimento de *Micro Scooter Challenge*, lançado no mercado internacional no mês seguinte. O game, uma simulação de corrida de patinetes, foi criado a pedido da Incagold, publicadora baseada em Londres, e distribuído em 14 países por diferentes empresas, com versões em Inglês e Alemão. A distribuição na Europa foi particularmente bem sucedida, e *Micro Scooter Challenge* ficou durante semanas entre os títulos mais vendidos na Alemanha.

[26] É importante ressaltar que os soldados inimigos em *Incidente em Varginha* retratavam especificamente forças de elite do exército americano.

Entretanto, o contrato com a Perceptum não foi honrado, desenvolvimento e *royaties* sobre as vendas não foram pagos e, como resultado, a empresa brasileira passou dois anos à beira da falência. Outro título que estava sendo desenvolvido pela Perceptum para a Incagold, *Beach Volleyball*, já em estágio de demo, teve seu desenvolvimento suspenso por falta de pagamento em 2001.

Em julho de 2001, outra empresa de games brasileira lançou seu primeiro título. Tratava-se de um game de tiro em primeira pessoa, *Hades 2*, da Espaço Informática. Criada em 1999, em Porto Alegre, pelos irmãos Augusto e Gustavo Bülow, a Espaço destacou-se desde o início pela dedicação e seriedade no processo de desenvolvimento. Soluções engenhosas e de baixo custo foram utilizadas para criar *Hades 2*, que empregava uma versão mais recente do mesmo engine de *Incidente em Varginha*. Os sprites de alguns personagens, por exemplo, foram gerados por Maurício Lucas a partir de imagens digitalizadas dos próprios membros da equipe, usando fantasias. *Hades 2* foi comercializado em quatro países: Brasil, Argentina, Inglaterra e Polônia.

A Perceptum produziu o advergame *Super Mini Racing*, um simulador de corrida com carros em miniatura, em meados de 2001. Trata-se do game 3D brasileiro com maior número de cópias distribuídas. Mais de meio milhão de exemplares foram encartados gratuitamente nos jornais *O Estado de S. Paulo* e *Jornal da Tarde*, nas edições de 12 de outubro de 2001. Em novembro do mesmo ano, a Espaço produziu o game natalino *Christmas Magic*, distribuído pela Incagold no Brasil, EUA, Inglaterra, Polônia e Coréia do Sul.

Outra produtora brasileira de games estreou com um título 3D em agosto de 2002. *Vampiromania* foi produzido pela Green Land Studios. Formada por Marcelo Duarte Oliveira, Leonardo A. Branco, Wagner Gomes Carvalho, Victor C. Strella e Vitor Gomes Rodrigues, a Green Land Studios foi aberta em outubro de 2001, em Santos, São Paulo. O game *Vampiromania* estava ligado à produção de *O Beijo do Vampiro*, da Rede Globo, e aparecia em diversas cenas, sendo jogado por personagens da novela. Os principais títulos 3D da Green Land foram resultados dessa parceria com a Rede Globo: *Big Brother Brasil 3D On-line*, de fevereiro de 2003; *Sandy e Júnior, Aventura Virtual*, de setembro de 2003; e *Aquária – O Jogo*, de dezembro de 2003.

Trophy Hunter 2003, game de simulação de caça desenvolvido pela SouthLogic Studios, foi publicado por Atari e Infogrames em setembro de 2002, no Canadá e nos Estados Unidos. O game resultou de diversas negociações com publicadores dos Estados Unidos, centradas na capacidade do Aspen Engine de simular ambientes

abertos e florestas. A boa receptividade desse título gerou dois outros produtos similares da SouthLogic, *Deer Hunter 2004* e *Deer Hunter 2005*, ambos publicados pela Atari Americana.

A Espaço Informática lançou, em maio de 2003, o game de simulação de tênis *Matchball Tennis*, distribuído pela Brasoft / BVH no Brasil, Alemanha, Áustria e Suíça. Um detalhe interessante do desenvolvimento de *Matchball Tennis* foi o processo utilizado para a animação dos personagens. Para evitar os altos custos da captura de movimentos, a equipe da Espaço utilizou técnica baseada na rotoscopia: gravou em vídeo a atuação de um tenista, em dois ângulos, para depois animar manualmente seu personagens, quadro a quadro, com base no material gravado. Apesar de trabalhoso, o processo resultou em qualidade similar à da captura profissional de movimentos, mas a custo irrisório.

Pferderennstall, da Espaço, é um game de estratégia em tempo real sobre o gerenciamento de um haras de cavalos de corrida. Publicado em janeiro de 2005 no mercado alemão, *Pferderennstall* foi um sucesso de vendas e teve como desdobramento a versão *Hipodromme Tycoon*, publicada em na França em novembro de 2005, e *Thoroughbred Tycoon*, seqüência publicada em 2006.

Finalmente, em 2006, a Perceptum publicou *Super Mini Racing Ice*, seqüência de seu primeiro advergame, em conjunto com a Canal Kids. A SouthLogic, por sua vez, concluiu *Kelloggs Football Tiger*, publicado no mercado norte-americano pela Microtime.