Desenvolvimento de Jogos 3D: Concepção, Design e Programação

Esteban Walter Gonzalez Clua[1], João Ricardo Bittencourt[2]

ICAD – IGames/VisionLab
Departamento de Informática – PUC Rio

Centro de Ciências Exatas e Tecnológicas
Universidade do Vale do Rio dos Sinos (UNISINOS)

esteban@inf.puc-rio.br, jrbitt@ludensartis.com.br

Abstract: *This paper describes traditional computer games development process. After a brief discussion about history and documentation, the paper will present the steps necessaries to create 3D games, with emphasis at professional and commercial tools. Along the document, important bibliographies for each process will indicated. Free Softwares or not expensive tools will be shown in order to help research institutions for developing projects in Games and Digital Entertainment field. In the last part, the paper will present other applications created with game development technology and will discuss perspectives for this new research field.*

Resumo: *Este documento discute de forma geral o processo de criação de um jogo computadorizado 3D. Inicialmente serão apresentadas as principais etapas na elaboração de um jogo 3D, destacando diversas ferramentas disponíveis no mercado. Procura-se também destacar e indicar as principais bibliografias adequadas para cada uma das etapas do processo de criação e ferramentas gratuitas e livres que facilitem o uso pela comunidade acadêmica viabilizando novas pesquisas na área de Jogos e Entretenimento Digital. No final deste documento serão apresentadas outras aplicações que podem ser desenvolvidas com a mesma base tecnológica dos jogos e as tendências futuras para este segmento.*

1. Introdução

Um jogo 3D é um software especial, pois contém elementos muito variados: módulos de Computação Gráfica, Inteligência Artificial, Redes de Computadores, Multimídia, entre outros. Todos estes módulos devem funcionar em perfeita harmonia, obedecendo a uma característica fundamental de um jogo: deve ser um software em tempo real. Para que isto seja possível é necessário explorar ao máximo o hardware dedicado, as conhecidas placas gráficas aceleradoras 3D. Para este propósito é fundamental que o jogo esteja baseado sobre diversas APIs, tais como o *OpenGL*, *DirectX* e *OpenAL*.

Além disso, enquanto a maioria dos softwares precisam apenas seguir uma série de requisitos e atender bem os propósitos para os quais foram elaborados, uma característica imprescindível para um jogo é que ele deve ser divertido e agradável de se utilizar, uma vez que seu principal objetivo é proporcionar entretenimento para as pessoas. Os jogos computadorizados precisam criar a sensação de imersividade nos

usuários, tal característica obtida pela combinação de aspectos artísticos e tecnológicos [BAT 02]. Assim, tratar de jogos computadorizados representa lidar com uma área extremamente interdisciplinar, aproximando os aspectos computacionais de outras ciências, tais como Educação, Psicologia, Artes Plásticas, Letras, *Design* Gráfico e Música.

Considerando o aspecto computacional tais aplicações requerem a adoção de sofisticadas técnicas que na maioria das vezes representam o "estado da arte" das pesquisas em Ciência da Computação principalmente as pesquisas relacionadas com Análise de Algoritmos (Otimização), Computação Gráfica, Redes de Computadores e Inteligência Artificial. Por esta razão desenvolver jogos computadorizados torna-se uma área fascinante para o desenvolvimento de aplicações técnico-científicas, conforme foi destacado por Battaiola [BAT 00].

Para Laird e Van Lent [LAI 01], os jogos computadorizados podem ser considerados a *killer application* da computação, principalmente na área de Inteligência Artificial, ou seja, uma aplicação modelo justamente pelo fato de possuir problemas significativos que ao serem solucionados irá impactar em outras aplicações.

É importante destacar que no Congresso da Sociedade Brasileira de Computação (2000) o Prof. Dr. André Battaiola publicou na Jornada de Atualização em Informática (JAI) o artigo intitulado *"Jogos por Computador – Histórico, Relevância Tecnológica e Mercadológica,Tendências e Técnicas de Implementação"* [BAT 00]. Tal publicação representa um marco significativo na comunidade acadêmica, pelo fato do reconhecimento das potencialidades técnico-científicas e mercadológicas das pesquisas aplicadas em jogos computadorizados e entretenimento digital. A publicação deste presente artigo após cinco anos a publicação do trabalho de Battaiola ocorre em um contexto diferenciado, pois atualmente a SBC já consolidou uma Comissão Especial de Jogos e Entretenimento, foi criada a Associação Brasileira das Desenvolvedoras de Jogos Eletrônicos (ABRAGAMES) e até mesmo ações governamentais já foram feitas como o JogosBR – Concurso de Jogos promovido pelo Ministério da Cultura, inúmeros editais FINEP/CNPq fomentando a área de entretenimento e o reconhecimento dos jogos computadorizados como obra de audiovisual pelo Ministro da Cultura. Tais ações evidenciam a importância do setor conforme foi apresentado por Battaiola, em 2000.

Dado esse contexto o objetivo principal deste artigo é apresentar o processo de concepção e desenvolvimento de jogos 3D, destacando ferramentas comerciais e alternativas livres e gratuitas que podem ser usadas para o desenvolvimento de aplicações multimídia em geral no âmbito acadêmico e extensível para o setor industrial. Espera-se desta forma fomentar novas pesquisas interdisciplinares na área da computação promovendo o desenvolvimento de tecnologia nacional aplicada ao setor de Entretenimento Digital. Certamente esta publicação não pretende tratar de todos os aspectos de desenvolvimento de jogos 3D, entretanto pretende servir de embasamento inicial para o desenvolvimento de grupos de pesquisas no âmbito das universidades brasileiras e auxiliar na formação tecnológica de novos estúdios de desenvolvimento. É importante destacar que este artigo possui uma natureza técnico-científica diferenciando-se da concepção tradicional dos demais artigos publicados neste evento. Entretanto tratar da área de *games* é um problema cujos aspectos tecnológicos são inerentes oriundos de um processo de pesquisa aplicada bastante comum no setor. Muitas técnicas que serão citadas representam resultados de pesquisa em Computação Gráfica que a posteriori acabam sendo utilizadas em ferramentas para o

desenvolvimento de *games*. Ao tratar de aspectos técnicos não está minimizando os aspectos científicos, mas sim destacando outro aspecto da ciência.

Para isto este artigo está organizado em dez seções. Na seção 2 é apresentado um breve histórico destacando a evolução dos jogos computadorizados até aos jogos 3D. É importante efetuar esta contextualização para compreender o segmento. Na seção 3 serão apresentadas e detalhadas cada uma das etapas consideradas no processo de desenvolvimento de um jogo 3D. Na seção 4 serão apresentados os principais conceitos teóricos dos motores de jogos, conhecidos como *engines*, destacando os *engines* 3D e demais *engines* de apoio (simulação física, Inteligência Artificial). Na seção 5 será relacionado o uso dos *engines* apresentando a ferramenta *3D Game Studio*, uma solução comercial. Em contrapartida, na seção 6, será apresentado o contexto da *engines* livres e gratuitas com destaque para o *Crystal Space* e o *Ogre3D*. Na seção 7 serão destacadas outras aplicações, principalmente de cunho científico que poderão ser desenvolvidas sob a mesma base tecnológica apresentada nas seções anteriores. Na seção 8 será apresentado detalhadamente o contexto mercadológico nacional e internacional e as iniciativas de fomento ao setor. Na seção 9 serão apresentadas as inovações de pesquisas no setor de entretenimento digital, com ênfase na TV Digital Interativa. Por último, na seção 10 são listadas as referências bibliográficas que servirão de base para continuidade dos estudos referentes aos jogos computadorizados e ao entretenimento digital.

2. História dos Jogos Computadorizados

Esta breve contextualização será baseada no trabalho de Xavier [XAV 03].

2.1 Arqueologia

O professor William Higinbothan (Figura 1) que, imaginando novas formas de demonstrações científicas, inventou um processo de entretenimento com um computador e um osciloscópio. A primeira experiência científica com o uso de imagens eletrônicas operadas por jogadores.

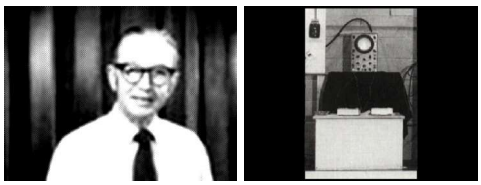


Figura 1: 1958 - Tênis para Dois

Steve Russel (Figura 2), então estudante do MIT, juntamente com amigos, desenvolveu *SpaceWar!*, o primeiro jogo eletrônico propriamente dito. O jogo funcionava como demonstração das capacidades gráficas do processador de imagens de alta resolução.

Figura 2: 1962 - SpaceWar!

Em 1972 (Figura 3), Nolan Bushnell, visionário de um mercado para jogos eletrônicos, popularizou as experiências lúdicas com *Computer Space* e outros jogos públicos operados nas rotas dos pinballs. Fundou com um amigo a Atari, a maior e mais influente empresa de entretenimento eletrônico da época.


Figura 3: 1972 - Computer Space

No mesmo ano Ralf Baer (Figura 4), engenheiro, desenvolveu *Odyssey*, o primeiro console de videogame da história, inicialmente solicitado por militares interessados em treinar soldados em lógica e reflexos rápidos e depois readequado para o uso doméstico.


Figura 4: 1972 - Mgnavox Odyssey

2.2 Arcades

Os arcades são máquinas de jogos de uso público, operados por fichas ou moedas em casas especializadas ou não. Fizeram grande sucesso na década de 70 e 80. O primeiro sucesso dos arcades trata-se do Atari PONG (1972)(Figura 5). Popularizou o conceito de entretenimento eletrônico de forma simplista e solidificou o fenômeno Atari. Objeto do primeiro grande litígio do mercado, PONG foi acusado de plagio do *Odyssey* e ele mesmo foi plagiado por inúmeras outras empresas.

Figura 5: Atari PONG (1972)


Figura 6: Atari *Shark Jaws* (1974)

Em 1974 é lançado o Atari *Shark Jaws* (Figura 6), o primeiro jogo a apresentar personagens independentes animados e uma tentativa explícita de associar o jogo ao fenômeno de bilheteria do cinema *Tubarão*. No ano seguinte é lançado o *Midway Gun Fight*, o primeiro jogo a utilizar um microprocessador, conferindo melhores gráficos e recursos de jogo para os seus participantes.

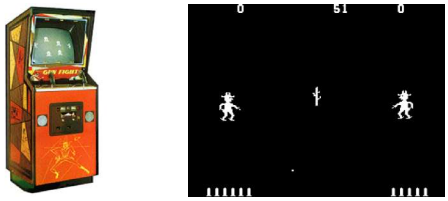


Figura 7: *Midway Gun Fight* (1975)

Em 1976, a Atari lançou o *Night Driver* (Figura 8) utilizando como atrativo visual em sua cabine, volante, marchas e pedais, o jogo se propõe a criar uma atmosfera realista. Na escassez de recursos gráficos, fazia uso do desenho do interior do carro pintado sobre a tela.

Figura 8: *Night Driver* (1976)

Em 1978, o Taito *Space Invaders* (Figura 9) trata-se da primeira grande repercussão dos jogos eletrônicos, que desde então invadiram a sociedade em todos as suas extensões. Para parecer colorido, o jogo japonês fazia uso de *overlays*.

Figura 9: Taito Space Invaders (1978)

Em 1979 com o lançamento do Atari *Asteroids* (Figura 10) é tido com um dos grandes clássico da Atari. Para o desenvolvimento deste jogo foi aproveitado a modelagem vetorial de sucesso como em *SpaceWar!*.

Figura 10: Atari *Asteroids* (1979)

No mesmo ano, a Namco *Galaxian* (Figura 11) foi lançado baseando-se no sucesso militarista de *Space Invaders*. É importante destacar que Space Invaders é o primeiro jogo em cores da história.

Figura 11: Namco *Galaxian* (1980)

Em 1980 é lançado Namco *Pac-Man* (Figura 12). Grande fenômeno de popularidade o jogo que inicialmente visava o público feminino tomou a mídia de assalto e provocou uma crise inflacionária no Japão. Licenciado para a Atari, a versão foi uma das responsáveis pela crise que encerrou o mercado.

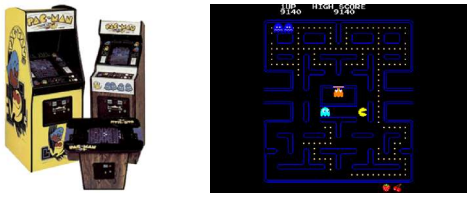


Figura 12: Namco *Pac-Man* (1980)

Nesse ano, *Stern Berzerk* foi o primeiro game a usar com qualidade a sintetização de voz, o jogo foi responsável pela morte de um estudante - a primeira diretamente associada ao uso de jogos eletrônicos - fazendo acalorar o debate sobre a influência dos jogos eletrônicos na sociedade.

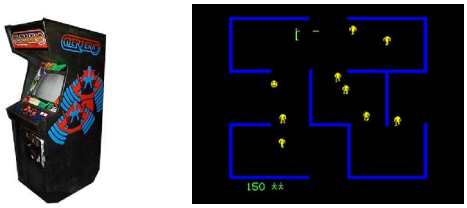


Figura 13: *Stern Berzerk* (1980)

Também em 1980, *Williams Defender* (Figura 14), foi criado praticamente por uma única pessoa, Eugene Jarvis. O jogo amplia o sentido de jogabilidade com vários controles, física realista e um sentido perceptivo original: o universo do jogo existe além do que é visto pelo jogador (conceito de *scrolling*).

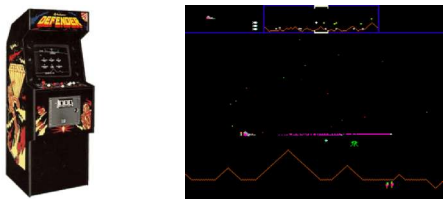


Figura 14: *Williams Defender* (1980)

No ano seguinte a Nintendo lança *Donkey Kong* (Figura 15). A primeira aparição do personagem mundialmente conhecido como Mario e *debut* de seu criador, Shigeru Miyamoto, como o grande designer de jogos de todos os tempos.

Figura 15: *Donkey Kong* (1981)

Em 1991, foi lançado *Capcom Street Fighter II* (Figura 16). *Game* adota o modelo de animação por *sprites*: imagens bidimensionais que se movimentam umas sobre as outras criando ilusão de profundidade e controle dos resultados.

Figura 16: *Street Fighter II* (1991)

A Acclaim, em 1992, lança *Mortal Kombat* (Figura 17)**.** O jogo adota o modelo de animação por captura de movimentos e digitalização subseqüente: imagens de atores são digitalizadas e animadas para a ação. Maior realismo estético em troca de movimentos mais repetitivos.

Figura 17: *Mortal Kombat* (1992)

Figura 18: *Virtua Fighter* (1993)

Em 1993, a SEGA lança *Virtua Fighter* (Figura 18). Esse jogo teve uma grande repercurssão na época por utilização um modelo de animação por objetos tridimensionais. Construções poligonais são animadas levando em conta massa e aceleração em tempo real de processamento. Com a popularização dos recursos gráficos de gerenciamento geométrico e das placas de processamento gráfico, os jogos ditos 3D tornaram-se paradigmáticos desde então, ou seja, começam a ser adotados de forma unânime na produção de jogos.

2.3 Consoles

Os consoles ou videogames são jogos eletrônicos desenvolvidos para uso doméstico. Basicamente funcionam acoplados a aparelhos de televisão. Em 1972, Magnavox Odyssey (Figura 19), após liberação de patentes, Baer conseguiu que a Magnavox produzisse seu produto. Monocromático e sem som, vinha com *overlays*, dados, fichas e dinheiro de brinquedo para aumentar a participação do jogador. Poucas variações de jogo e planos de *marketing* errados não impulsionaram as vendas iniciais que pararam após 100 mil unidades.

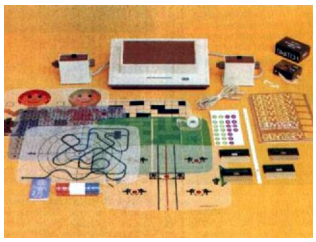


Figura 19: *Magnavox Odyssey* (1972)

Em 1975, Atari *Home Pong* (Figura 20), apoiada financeira e logisticamente pela Sears, a Atari inundou o mercado com a versão particular de *PONG*, seu primeiro sucesso. O excesso do produto e de similares foi responsável pela primeira grande crise do mercado.

Figura 20: Atari *Home Pong* (1975)

Em 1976, *Fairchild Channel F* (Figura 21) foi o primeiro console a multiplicar os jogos por dispositivos de memória externa em cartuchos plásticos. Não permaneceu muito tempo no mercado devido a pouca variabilidade de jogos e principalmente pelo preço final dos mesmos.

Figura 21: *Fairchild Channel F* (1976)

Entretanto o ano de 1977 é um marco na história dos videogames. Ocorreu o lançamento do Atari VGS 2600 (Figura 22), o maior fenômeno de popularidade de sua época, cerca de 25 milhões de unidades vendidas em 5 anos. O *VGS 2600* aprimorou a idéia original de reprogramabilidade do *Chanell F* para bases mais baratas. O carro-chefe da popularização mundial da marca Atari no mundo também extenuado por uma ludoteca de quase dois milhares de jogos desenvolvidos por diversas *softhouses*, incluindo a dissidente Activision.

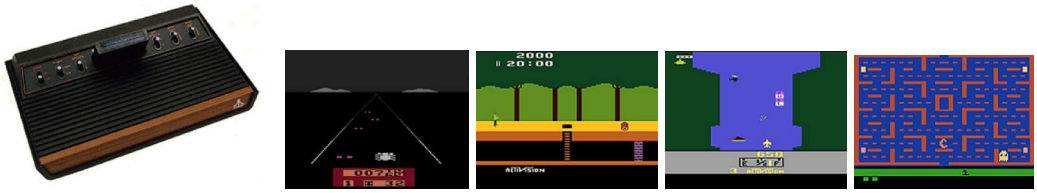


Figura 22: Atari VGS 2600 (1977)

O lançamento do Mattel *Intellivision* (Figura 23) em 1980 representava o concorrente tecnicamente mais poderoso da Atari. O console trazia atrativos como um sintetizador de voz acoplável e um conversor de modo a fazer com que jogos do *VGS 2600* pudessem ser usados nele.

Figura 23: Mattel *Intellivision* (1980)

2.3.1 O *crash* de 1984

O ano de 1984 representou a segunda e derradeira grande crise do mercado de jogos eletrônicos. A Atari se dissolve e arrasta consigo todo o mercado ocidental de consoles. Os principais motivos de tal crise são:

1. **Defasagem tecnológica -** Apesar de em declínio desde fim da Era de Ouro em 1980, os jogos para arcade haviam alcançado um incrível grau de qualidade gráfica com uso, inclusive, da tecnologia de vídeo digital. Por outro lado os consoles ainda engatinhavam com tecnologias obsoletas;
2. **Crise de conteúdo -** Com incontáveis *softhouses* desenvolvendo jogos para a Atari, muitos não correspondiam em qualidade. Por exemplo, um jogo da *Mystique* provocou revolta na sociedade americana por apresentar como temática uma proposta pornográfica e violenta contra a minoria indígena;
3. **Início da informática doméstica -** Visando o grande público e a partir da redução dos preços dos dispositivos eletrônicos a *Commodore* lança VIC-20, o primeiro microcomputador colorido a superar a barreira de U$ 300,00.

Neste mesmo período no Japão, duas empresas despertam novamente o interesse da juventude: a Nintendo e a SEGA. A Nintendo, uma ancestral fábrica de cartas investe em eletrônicos em meados do século XX e tenta se estabelecer nos EUA com o sucesso de *Donkey Kong*. Apostando em uma produção iconoclasta própria, na figura

emblemática de Miyamoto e em contratos de exclusividade com *softhouses* a empresa rapidamente se torna a mais respeitada do novo mercado. A SEGA, uma poderosa concorrente da Nintendo avança sobre o mercado oriental com voracidade e tecnologia de ponta. O grande embate entre as duas empresas reaquece o mercado e garantirá a grande batalha pela subexistência dos consoles e do mercado durante a próxima década.

2.3.2. Geração 8 bits

Em 1985 é lançado o Nintendo *NES*. O *NES* (Figura 24) foi o grande atrativo da renovação pois além de muito colorido, rapidamente já contava com um acervo de dezenas de jogos. No ano seguinte a SEGA lança o *Master System* (Figura 25). Sucesso apenas na Europa o Master System possuía jogos melhores mas em número extremamente reduzido, além de pistola e óculos tridimensionais. A falta de CRPGs (*Computer Role-Playing Games*) foi crucial para a hegemonia do NES sobre ele.

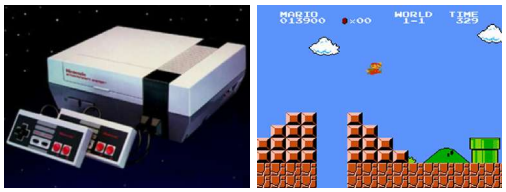


Figura 24: *NES* (1985)

Figura 25: *Master System* (1986)

2.3.3 - Geração 16bits

Em 1988, a SEGA lança o *Genesis* (Figura 26) tentando sobrepujar a concorrente. A SEGA investe em uma nova arquitetura para o nível da apresentação dos consoles, diminuindo o *gap* em relação aos arcades. Novamente o mercado dos jogos respira nos EUA com a ajuda do mascote Sonic. Jogos de CRPG invadem o Japão e acertam plenamente o interesse do público.

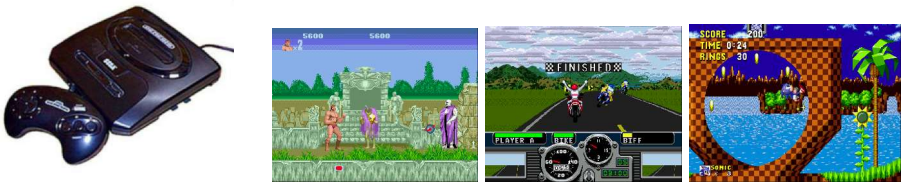


Figura 26: *Genesis* (1988)

Dois anos depois, a Nintendo lança o *SNES* (Figura 27). A Nintendo apostou em novidades e na revalorização das personalidades da casa. Mario garante uma franquia milionária para a empresa que se solidifica para outras novidades.

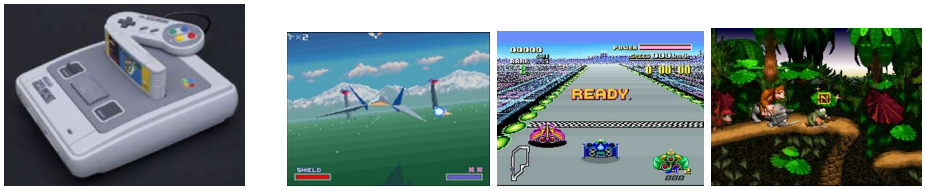


Figura 27: *SNES* (1990)

2.3.4. Geração 32bits

Em 1993 é lançado o *3DO* (Figura 28). Um dos maiores fracassos do mercado dos jogos eletrônicos, o console foi concebido por Trip Hawkins, fundador da *Electronic Arts*, para ser o ícone de desenvolvimento da nova geração. Tudo não passou de uma idealização em torno de um aparelho caro.

Figura 28: *3DO* (1993)

Em 1994, a Sony entra no mercado dos jogos eletrônicos com o lançamento do *PlayStation* (Figura 29). Inicialmente proposto como o *upgrade* do *SNES*, a Sony resolveu ela mesma produzir o aparelho que celebrou o ápice da revolução multimídia para os consoles. Sucesso de crítica e vendas o *PlayStation* ainda é fabricado em nova e diminuta versão.

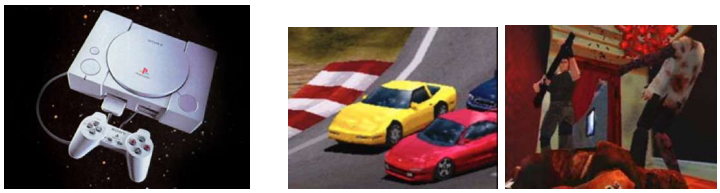


Figura 29: *Playstation* (1994)

Neste mesmo ano a SEGA lança o *Saturn* (Figura 30) para ser concorrente direto do *PlayStation.* O aparelho não teve sucesso por permanecer vinculado a uma arquitetura complexa de muitos processadores e as dificuldades da programação em *Assembler*.

Figura 30: SEGA *Saturn* (1994)

2.3.5. Geração 64bits

Em 1993 é lançado o Atari *Jaguar* (Figura 31). Representante indevido da geração 64bits (por usar dois processadores de 32bits e não um de 64bits) representava o último recurso da Atari antes de ser fortemente pela crítica. Poucos jogos, tentativa infrutífera de revitalização com um leitor de CD e controles gigantes. O público ficou mais crítico.

Figura 31: Atari *Jaguar* (1993)

O Nintendo 64 (Figura 32) é lançado em 1996 como uma rápida incursão da Nintendo garantida pela franquia *Pokemón* e por jogos de esporte. Acabou não conquistando o grande público com gráficos refinados porém de texturas repetitivas em cartuchos muito caros. Além de tudo, perdia em originalidade: revisitações constantes a sucessos de seus títulos antecessores.

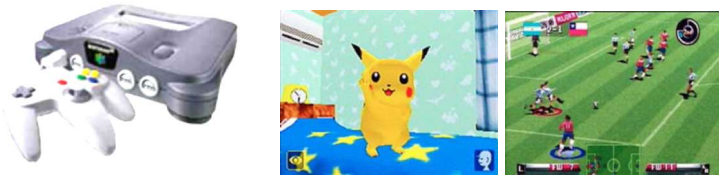


Figura 32: Nintendo 64 (1996)

2.3.6. Geração 128bits

Em 1998, é lançado o SEGA *DreamCast* (Figura 33). Este console teve uma curta existência no mercado - apenas dois anos. O aparelho apesar de excepcional do ponto de vista tecnológico não garantiu público suficiente para arcar com as dívidas da SEGA, que desde então decidiu focar-se unicamente em softwares.

Figura 33: *DreamCast* (1998)

Figura 34: *PSX2* (2000)

Em 2000, a Sony lança o *PSX2* (Figura 34). Adota a tecnologia de DVD, o console manteve um público fiel por garantir que jogos do *PlayStation* pudessem permanecer sendo usados e de forma melhorada.

Em 2001, a Nintendo lança o *GameCube* (Figura 35) e a Nintendo entra nas mídias digitais. O aparelho garantiu a empresa à vitória final do século XX: um mercado mundial fiel ao console e novos protótipos em desenvolvimento.

Figura 35: *GameCube* (2001)

Neste mesmo ano, a Microsoft entra no mercado dos consoles com o lançamento do *X-Box* (Figura 36). O aparelho possui adota a tecnologia DirectX facilitando a adaptação de grandes sucessos dos PCs para o console. Ainda que as vendas não tenham refletido as expectativas, a Microsoft recentemente lançou seu novo modelo, o *X-Box 360*.

Figura 36: *X-Box* (2001)

2.5. Computadores

Jogos de computador surgem como experimentos lingüisticos e de comando homem-máquina. As primeiras empresas desenvolvedoras de jogos para PCs são formadas por jovens entusiastas em garagens, vendendo a produção em lojas de eletrônica e informática. As temáticas iniciais tratavam-se dos *Adventures*, jogos de exploração ambiental através de uma leitura interativa (Figura 37) e os CRPGs - jogos de representação de um universo próprio onde é possível enriquecer experiência e outros atributos dos personagens. Diferem do RPG convencional por ater-se ao mecanismo reativo do jogador diante de escolhas pré-estabelecidas (Figura 38).

Figura 37: *Mistery House*

Figura 38: Zork

Para os computadores pessoais a década de 80 foi extremamente significativa. Em 1983 é lançado *Planetfall* (Figura 39), o primeiro jogo a ser populado por personagens atrativos psicologicamente e apresentar um enredo complexo para a participação do jogador.

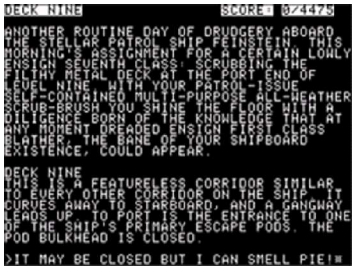


Figura 39: *Planetfall* (1983)

Em 1984 foi lançado o *Flight Simulator* (Figura 40). Depois foi relançado pela Microsoft e o jogo tornou-se o referencial para jogos de simulação operacional realista.

Figura 40: Flight Simulator (1984)

Em 1985 é lançado *Where in the World is Carmen Sandiego?* (Figura 41) Um misto de administração de tempo com aula de geografia. O jogo foi um dos precursores da temática *edutainment* - pressuposto educacional para o divertimento do jogador.

Figura 41: *Whre in the World is Carmem Sandiego?* (1985)

Em 1987 é lançado o *Tetris*, o mais famoso jogo de todos os tempos tem uma história insólita que envolve pirataria e espionagem durante a Guerra Fria. Primeiro jogo a sair da Cortina de Ferro. *Tetris* é sem dúvida o grande bastião da simplicidade para o divertimento (Figura 42). No mesmo ano é lançado *Test Drive* (Figura 43). Foi o precursor de jogos de corrida usando bólidos famosos, o jogo da *Accolade* imerge o jogador no cenário através do ponto de vista do motorista.

Figura 42: *Tetris* (1987)

Figura 43: *Test Drive* (1987)

Em 1989, *Prince of Persia* (Figura 44) usa animações realistas sobre um cenário aparentemente simplista. O jogo é o melhor exemplo de avaliação de recursos vitais para a garantia de um bom projeto de jogo. Também em 89, *SimCity* (Figura 45) é lançado. É usado inclusive em escolas para compreensão de civismo e responsabilidade social o jogo de administração de cidades é um dos mais renomados por ter feito sucesso sem se ater na valorização da violência.

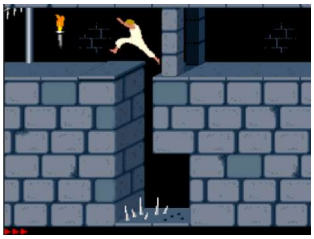


Figura 44: *Prince of Persia* (1989)

Figura 45: *SimCity* (1989)

2.5.1. Quebra dimensional – Do 2D para o 3D

Com as novas tecnologias de criação de universos tridimensionais e a redução dos preços de processadores mais potentes, os jogos para computador tornaram-se altamente imersivos. O primeiro sucesso desta tecnologia para os PCs é *Wolfenstein 3D* (1992) (Figura 46). A partir de experiências prévias com imagens distorcidas sobre sólidos para a falsa ilusão de profundidade, o jogo apresenta pela primeira vez de forma atrativa um elemento que se joga sobre o cenário - uma arma - garantindo ao jogador a experiência de participar visualmente na ação (jogos em primeira pessoa conhecidos como FPS – *First Person Shooters*).

Figura 46: *Wolfenstein 3D* (1992)

Em 1994 é lançado *Doom* (Figura 47) pela id Software. Como atualização da proposta de *Wolfenstein 3D*, melhorias no som (agora também espacial) e na construção ambiental (diversos níveis arquitetônicos possíveis), *Doom* marcou o ápice do desenvolvimento para jogos que seriam doravante conhecidos como FPS.

Figura 47: *Doom* (1994)

Em 1996 é lançado *Quake* (Figura 48). Totalmente revolucionário para sua época, o jogo garantia, sem a necessidade de uma placa aceleradora 3D, a experiência verdadeiramente tridimensional. Todos os elementos - cenários e personagens - eram providos de volume. Desde então, o uso de ambientes construídos tridimensionalmente seria uma constante.

Figura 48: *Quake* (1996)

3. Etapas do Processo de Elaboração de um Jogo

Como qualquer outro software a produção de um jogo computadorizado, seja esse 2D ou 3D, requer a adoção de um processo de desenvolvimento. No caso das aplicações de entretenimento digital, tais com os jogos, é necessário tratar dos aspectos artísticos neste processo. Simplificadamente o processo de desenvolvimento de um jogo 3D envolve as seguintes etapas [ROL 04]:

1. Confecção do *Design Bible*;
2. Produção de áudio e imagens 2D;
3. Modelagem 3D;
4. Desenvolvimento dos artefatos computacionais. Basicamente trata-se da escolha ou desenvolvimento do *engine;*
5. Integração dos aspectos artísticos com os aspectos computacionais.

Nas próximas subseções serão descritos os três primeiros passos no processo de desenvolvimento de um jogo. O passo 4 é descrito nas seções 3,4 e 5.

3.1. *Design Bible*

Assim como não é possível criar um filme sem antes ter um roteiro bem elaborado, também é impossível desenvolver um jogo sem antes ter um documento com todas as suas especificações.

Costuma chamar-se a este documento de *Design Bible*, que pode ser visto como uma espécie de manual de instruções para os futuros desenvolvedores do jogo [ROL 04]. De fato, tão importante é este documento, que o processo de desenvolvimento não pode começar sem que esse não esteja pronto. O *Design Bible* deve conter os seguintes elementos descritos abaixo:

Roteiro

Cada vez mais se assemelham a roteiros de filmes. Este é um item fundamental para o processo de criação e será o elemento crucial para convencer os investidores da potencialidade do produto. É nesse item que o jogo deve mostrar seu diferencial em relação aos outros. Chamam-se aos roteiros de jogos de roteiros interativos, pois diferentemente que os roteiros de filmes, devem ter espaço para interferência do usuário no desencadeamento da história. Ao elaborar o roteiro deve-se considerar qual o estilo do jogo que será desenvolvido.

Game Design

Entende-se por game design a conceituação artística do jogo. Hoje em dia, dada a complexidade das histórias e dos cenários elaborados é importante que esta parte do documento seja escrita por um artista. Dentro deste item deverão ser expostos quais as principais características dos cenários, esboços de personagens, descrição das texturas fundamentais, mapas e descrições das fases (também denominado de *level design*). O livro [CRA 04] descreve detalhadamente como é a elaboração deste tópico. Na obra de Salen [SAL 04] também são descritos aspectos de *game* design do pontos de vista da concepção e contextualizado no aspecto sociológico. Veja Figura 49.

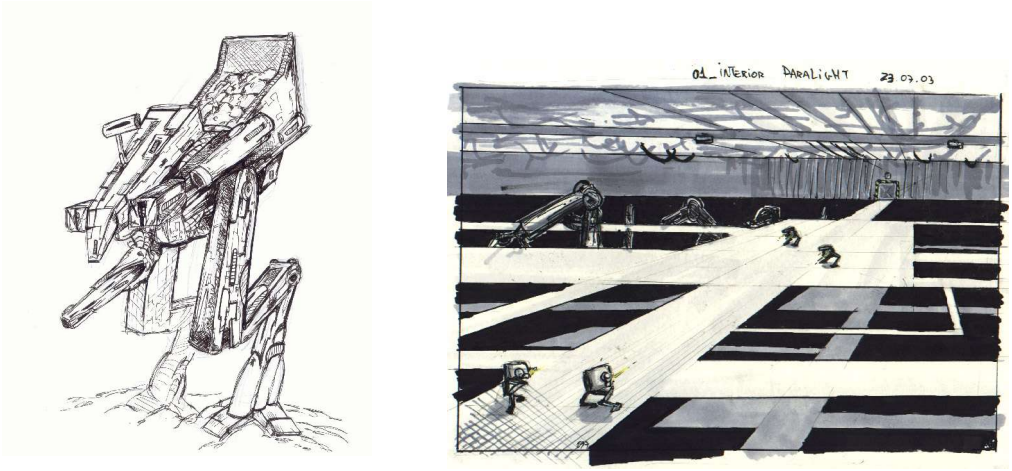


Figura 49 – Exemplo da conceituação artística de um personagem e de um cenário. (extraído do jogo METALmorphosis ©)

Game Play

Nesta parte do documento deve descrever-se como será a jogabilidade. Por jogabilidade entendem-se as regras do jogo e o balanceamento das regras (*game balancing*). Nesta descrição deve ficar claro que o jogo é divertido e irá proporcionar desafios interessantes. Esta parte do documento é muito importante para guiar os programadores principalmente na etapa de *scripting*.

Interface Gráfica

Pode-se dividir a interface em *ingame* e *outgame*. A primeira consiste na instrumentação disponível durante o jogo e é responsável pela entrada de dados do jogador para a aplicação. A interface *outgame* é a forma de apresentar a introdução do jogo, sua configuração, instruções, carregar um jogo salvo anteriormente, entre outras operações de suporte. Costuma-se dizer que a melhor interface é aquela que passa desapercebida para o jogador, permitindo que o mesmo possa focar-se no desenrolar da história e das ações. Veja Figura 50.

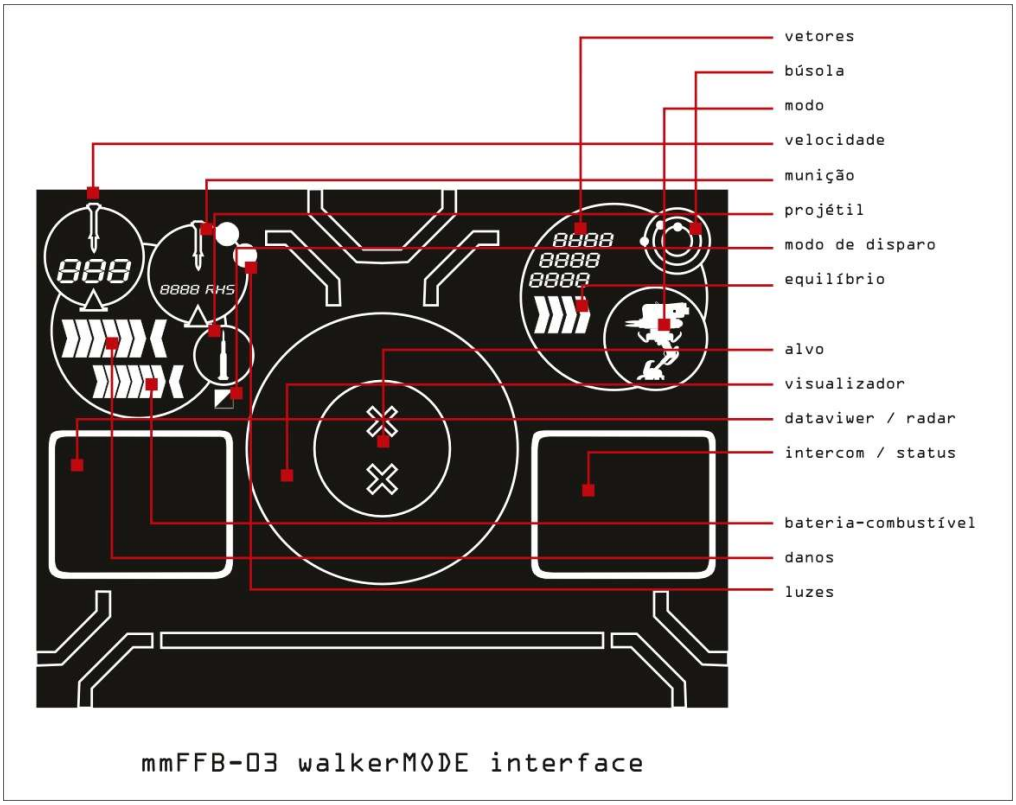


Figura 50 – Exemplo do design de uma interface *ingame* (extraído do jogo METALmorphosis ©).

Terminada a etapa de conceituação, o desenvolvimento de um *game* divide-se em dois caminhos distintos: o de criação artística e o de programação, havendo entretanto uma grande interseção entre ambas. A criação artística pode ser compreendida na elaboração dos *resources* do jogo, ou seja, os elementos que serão usados para sua montagem: modelos 3D, texturas, terrenos, sons, músicas e arquivos de configuração.

3.2. Produção de Áudio e Imagens 2D

Antes de abordar os *engines* propriamente ditos é importante destacar outros aspectos tecnológicos que são utilizados na produção de um jogo – áudio e imagens 2D. Para produção de áudio existem duas ferramentas comerciais bastante arrojadas e bem utilizadas profissionalmente – Cubase e o SoundForge [SON 05]. O Cubase é usado basicamente para mixagem de canais MIDI e o SoundForge é usado para produção de trilhas e efeitos sonoros. O SoundForge é comumente usado no cinema, na televisão e na produção de vinhetas para o rádio. O Audacity [AUD 05] é uma ferramenta livre que permite a criação de áudio, inclusive combinando diferentes canais de som, importando e exportando arquivos no formato WAV, MP3 e Ogg Vorbis. Entretanto não possui todos os recursos oferecidos pelo SoundForge.

No aspecto de áudio, a *OpenAL* desenvolvida pela *Loki Software*, trata-se de uma alternativa de implementação de som bastante interessante, pois a *OpenAL* permite adicionar som 3D, uma biblioteca estruturada na forma de uma máquina de estados que oferece um conjunto de primitivas de mais alto nível para manipulação de som. Para incrementar a imersividade é fundamental adicionar a percepção sonora no *game*, entretanto um mundo virtual pode possuir muitas fontes de som dificultando alguma pré-mixagem [TSI 04] e limitando a quantidade de sons ao máximo de canais de som disponíveis no sistema (em geral de 16 a 64 canais). Assim, o som 3D trata-se de um processo de "renderização" de áudio, ou seja, a produção de áudio em tempo de execução [TSI 04].

Os jogos 3D não são construídos somente com modelos tridimensionais, na produção de um jogo também é necessário compor imagens bidimensionais. Em geral, tais imagens serão usadas como texturas, mas também serão usadas para compor a interface gráfica *ingame* e *outgame,*tais como, botões, janelas, barras de energia e outros componentes gráficos.

Para produzir estas imagens existe uma série de ferramentas para editoração gráfica cujo Adobe Photoshop é um dos softwares mais tradicionais para desempenhar tal atividade. Outra ferramenta alternativa muito completa, mutilplataforma e livre é o GIMP (*GNU Image Manipulation Program*) [GIM 05]. Veja Figura 51.

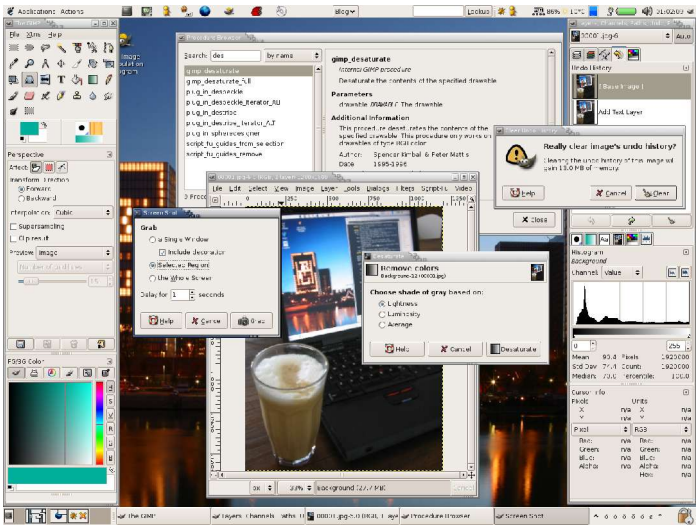


Figura 51: Interface do GIMP que ilustra as barras flutuantes e a barra de ferramenta para criação e edição de imagens bidimensionais.

O GIMP basicamente fornece as mesmas funcionalidades do Photoshop para efetuar tratamento de imagens. Oferece uma série de pincéis, permite trabalhar com inúmeras camadas, converte e salva em diferentes formatos de imagens (BMP,PNG,JPG,TIG,TGA,PCX,GIF, entre outros), uso de máscaras, oferece uma grande quantidade de filtros (*gaussian*, detecção de bordas, distorções, efeitos de luz, entre outros), suporte a macros e permite adicionar novas funcionalidades (filtros e formatos de arquivos) no programa através de *plug-ins* desenvolvidos em C usando *libgimp*. Também permite criar novas extensões usando uma linguagem de *scripts* Script-Fu, baseada em Scheme que equivale as macros. Além disso, permite salvar uma imagem como matrizes em arquivos de código em C, com extensão .c ou .h, que podem ser integrados em sua aplicação. Tal funcionalidade é bastante interessante, por exemplo, a *OpenGL* manipula diretamente matrizes com dados, desta forma criando gráficos no GIMP e depois exportando para C facilita a integração em aplicações que usam *OpenGL*.

As texturas são usadas para representar os materiais que compõem os modelos 3D. Tratam-se de um *bitmap* diferenciado que será replicado sob uma superfície com o objetivo de representar algum material, tais como metal, madeira, concreto, plástico, ou qualquer outro. Muitas texturas são produzidas fotografando-se o material e depois manipulando a imagem digitalmente. Tal procedimento aumenta o realismo das imagens compostas pela textura. Atualmente com a popularização das câmeras digitais facilitou-se o trabalho de obtenção de materiais para criação dessas texturas mais realísticas.

A seguir será descrito como criar uma *plug-in* para produzir uma textura de madeira usando GIMP que poderá ser usada nos modelos 3D. O GIMP possui algumas texturas prontas. O GIMP permite criar macros usando *Script-Fu*, uma linguagem interpretada assemelhando-se a sintaxe do LISP. Os arquivos podem ser criados em qualquer editor de texto. Devem ser salvos no diretório **scripts**, localizado no diretório **. gimp-X** (sendo X a versão do GIMP) localizado no diretório do usuário, no caso dos sistemas Unix no diretório **home**. Abaixo está descrito a sintaxe de um *script* para criar textura de madeira.

```
(define (script-fu-wood-texture inW inH)
      (let*
            (
                  (theImg (car (gimp-image-new inW inH RGB)))

                  (theLayer (car (gimp-layer-new theImg inW inH
                              RGBA-IMAGE  "layer1"  100  NORMAL-MODE)))
            )
            (gimp-image-add-layer theImg theLayer 0)
            (gimp-image-set-active-layer theImg theLayer)
            (gimp-palette-set-background '(248 200 124))
            (gimp-edit-fill theLayer 1)
            (plug-in-solid-noise 1 theImg theLayer 1 1 10 7 0 0)
            (plug-in-solid-noise 1 theImg theLayer 0 0 10 7 5 5)

            (set! theCopy (car (gimp-layer-copy theLayer 0)))
            (gimp-image-add-layer theImg theCopy 0)
            (gimp-image-set-active-layer theImg theCopy)
            (plug-in-mblur TRUE theImg theCopy 0 0 20 270)
            (gimp-desaturate theCopy)
            (gimp-layer-set-opacity theCopy 38)
            (gimp-layer-set-mode theCopy 10)
            (plug-in-edge 1 theImg theCopy 9.0 0 1)
```

```
            (gimp-display-new theImg)
      ))
;end define

(script-fu-register "script-fu-wood-texture"
      _"<Toolbox>/Xtns/Script-Fu/Patterns/Textura de madeira..."
      "Cria uma textura de madeira para ser usada em modelos 3D.\
Requer as dimensoes da imagem."
      "Joao Ricardo Bittencourt"
      "Sob GNU GPL license"
      "April 25, 2005"
      "RGBA"
      SF-ADJUSTMENT     _"Largura"       '(64 1 1000 1 10 0 1)
      SF-ADJUSTMENT     _"Altura"       '(64 1 1000 1 10 0 1)
 )
```

Após o script ser registrado no GIMP a nova funcionalidade poderá ser usada em outros plug-ins. Além do editor gráfico esta capacidade de personalização torna o GIMP uma opção bastante interessante de ser usada no desenvolvimento de jogos, pois se o artista e o desenvolvedor desejam um determinado visual gráfico ao invés de se preocuparem na criação de uma ferramenta de edição gráfica, procuram criar um *script* que possa ser integrado na arquitetura do GIMP. Assim, o foco é para o efeito e não para estrutura de software complementar.

3.3. Modelagem 3D

A equipe de modelagem 3D será responsável por criar os objetos geométricos das fases. A geometria de um jogo pode ser dividida em dois tipos: modelagem estrutural e modelagem de elementos dinâmicos. Esta diferenciação existe pelo fato de que os modelos estruturais, por não sofrerem alteração de posição, sofrerão um pré-processamento, de maneira a otimizar o processo de renderização, como será apresentado na sessão 3. A modelagem estrutural consistirá basicamente na criação do cenário em si, o terreno e alguns outros elementos estáticos.

Para esta etapa os principais softwares utilizados são o Discreet 3DS MAX [DIS 05], MAYA [ALI 05], Avid Softimage [AVI 05] e Ligthwave [NEW 05], pois fornecem recursos avançados tornando-os ótimos para este processo. Tais recursos estão descritos abaixo:

- ***Ferramentas de modelagem baseadas em polígonos:*** Toda a modelagem deverá ser feita por polígonos. Assim sendo, é importante que haja uma fácil e intuitiva forma de manipulá-los;
- ***Ferramentas intuitivas para texturização:*** Grande parte da riqueza de uma modelagem está na boa aplicação de texturas sobre os modelos. É comum, por exemplo, ter que aplicar um mapeamento de texturas em polígonos individuais;
- ***Boas ferramentas para otimização de polígonos:*** É comum durante o processo de modelagem criar objetos com mais polígonos do que se pode suportar no jogo. Assim sendo, é importante que um pacote de modelagem forneça recursos para reduzir o número de polígonos de objetos, minimizando a sua perda de qualidade;

- ***Boa interface de visualização:*** Para o artista é importante que a medida que um objeto seja construído, possa acompanhar este processo em tempo real, sabendo a priori como o mesmo será visto no jogo. Veja Figura 52.

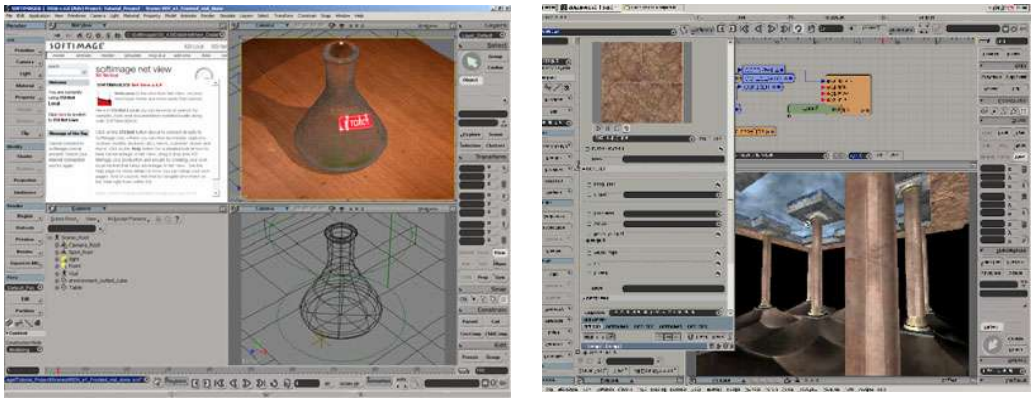


Figura 52 – As interfaces *"what you see is what you play"* permite que o artista possa ver em tempo real, na interface do software de modelagem, como seus modelos ficarão na renderização final do jogo. Imagens extraídas do *Avid Softimage XSI ®.*

Neste processo, uma virtude importante que os artistas devem ter é a de serem capazes de modelar objetos com o menor número de polígonos possível (Figura 53). Otimizando-se a modelagem, será possível que o cenário possa ser mais extenso e que mais objetos possam ser inseridos no mesmo.

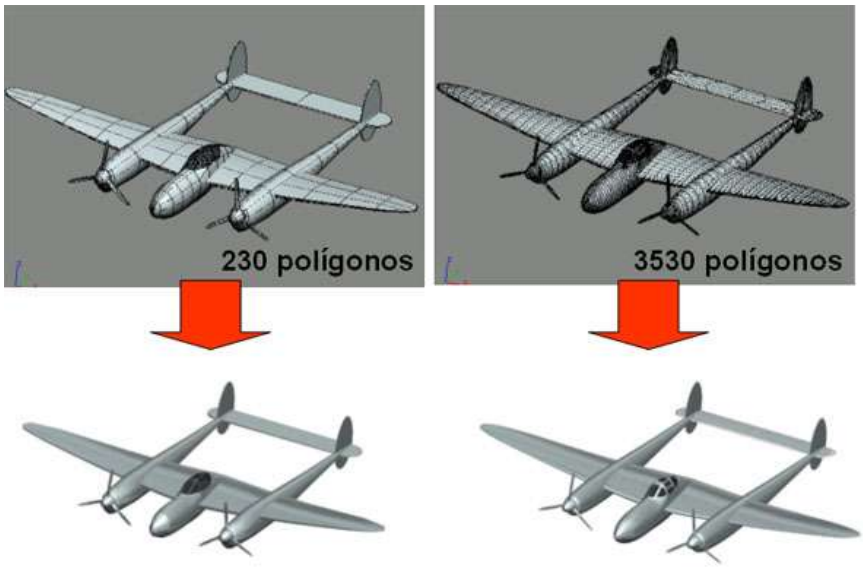


Figura 53 – Um mesmo modelo está representado em duas resoluções diferentes de polígonos. Perceba-se que o resultado final é bastante semelhante, embora o avião da direita possua 15 vezes mais polígonos que o da esquerda.

Além de tais ferramentas, uma alternativa livre e gratuita é o *Blender3D* [BLE 05]. Trata-se de um software gráfico completo que oferece funcionalidades de modelagem, animação, renderização, pós-produção e criação 3D. Sua principal desvantagem é sua interface gráfica que não é fácil de ser utilizada, sendo pouco intuitiva (Figura 54). O Blender3D suporta importação e exportação de diferentes

formatos incluindo 3DS, Cal3D, MDL, OBJ, VRML, DirectX, entre outros. Através da criação de scripts em Python é possível desenvolver novos *plug-ins* estendendo as funcionalidade básicas da ferramenta. O script *MakeHuman* (Figura 55) é um exemplo de *plug-in* livre desenvolvido para o Blender3D visando a modelagem de personagens humanóides. Inclusive o Blender3D pode ser usado como engine de jogos computadorizados, oferecendo tratamento de colisão, suporte a áudio e a *OpenGL*, e permite que a lógica do jogo seja programada em *Python*.

Figura 54: Algumas *screenshots* do Blender3D que ilustram a capacidade de editor para modelagem e animação 3D.

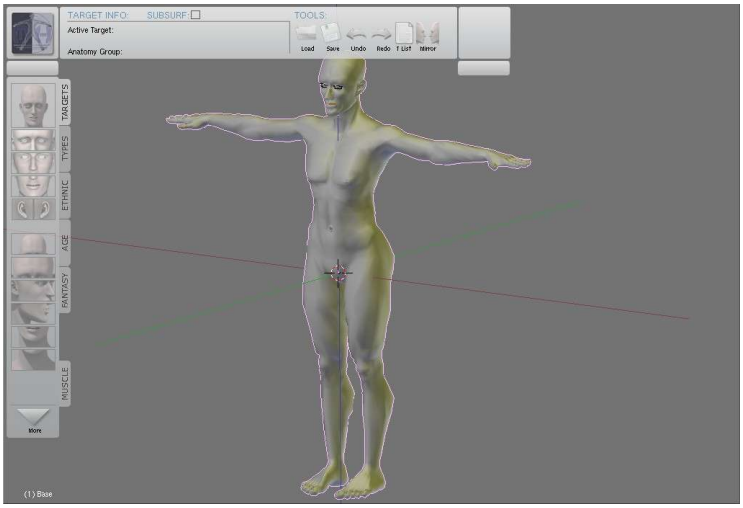
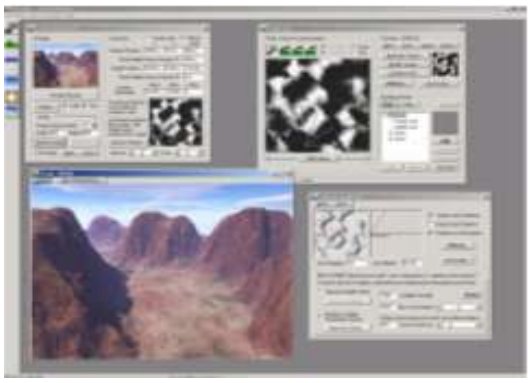


Figura 55: *Screenshot* do *MakeHuman*, script desenvolvido para o Blender3D para modelagem de personagens humanóides.

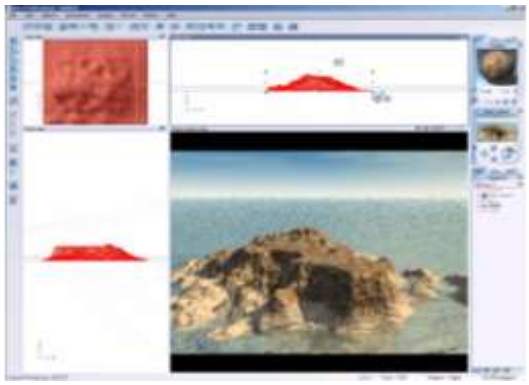
3.3.1. Terrenos

Os terrenos também são conhecidos como *height maps* e consistem em imagens com diversas tonalidades de cinzas. Os *pixels* escuros corresponderão a áreas baixas do terreno e os claros a partes mais altas. Assim sendo, elaborar um terreno resume-se a criar um mapa de altura adequado para o cenário. Este mapa de altura pode ser pintado manualmente, usando algum software de edição de imagem. Entretanto, para relevos mais complexos existem inúmeras ferramentas capazes de gerar mapas mais detalhados, bem como editá-los de forma mais intuitiva. Além de gerar os mapas de altura, estes softwares são capazes de gerar suas texturas correspondentes. Algumas das ferramentas mais utilizadas são o Corel Bryce [DAS 05], VueD'Espirit [EON 05], Terragen [TER 05] e Mojo Word Generator [PAN 05]. Veja Figura 56.

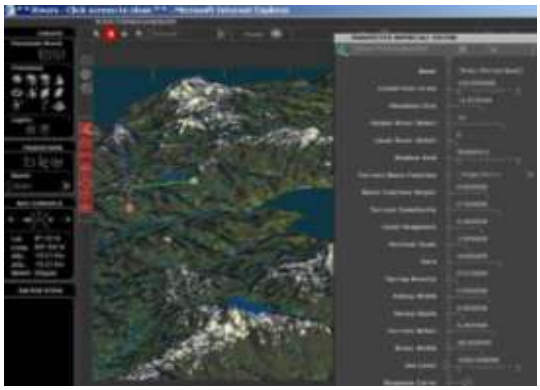


Corel Bryce

Terragen



VueD'Espirit



Mojo World Generator

Figura 56 – Softwares utilizados para criar *height maps* e suas correspondentes texturas.

Posteriormente, estes mapas de altura serão projetados sobre malhas regulares de polígonos. Dependendo do engine, isto será feito na ferramenta de modelagem 3D ou no editor específico do engine. Veja Figura 57.

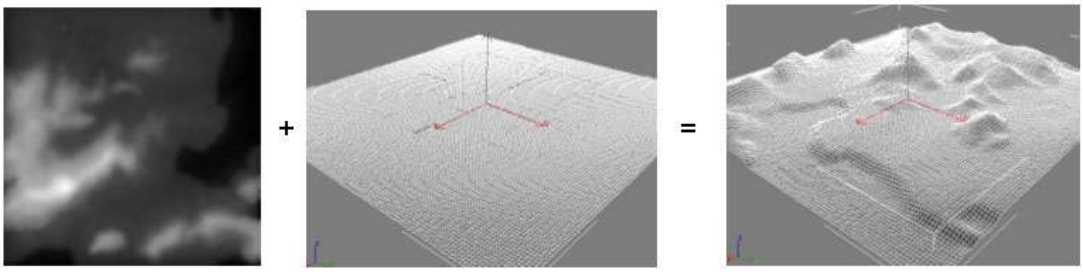


Figura 57 – Um mapa de altura é aplicado sobre uma malha de polígonos regulares, formando um terreno 3D. Este processo pode ser efetuado no engine ou no software de modelagem 3D.

4. Engines

Um motor de um carro é responsável por fazê-lo andar. Ao dar a ignição do veículo, o motorista coloca o motor em funcionamento e começa a mover-se com ele, sem precisar saber como funciona todo o processo mecânico. A transferência do movimento dos eixos para as rodas, a sincronização das explosões dos pistões, a injeção de combustível na câmara de combustão, tudo fica a cargo do motor. Um engine para jogos basicamente

segue o mesmo princípio de funcionamento. Dentro do conceito de engenharia de software trata-se da parte do projeto que executa certas funcionalidades para um programa. Dentro da área de jogos, um *engine* se encarregará por entender-se com o hardware gráfico, irá controlar os modelos para serem renderizados, tratará das entradas de dados do jogador, tratará de todo o processamento de baixo nível e outras coisas que o desenvolvedor de jogos normalmente não deseja fazer ou não tem tempo para se preocupar. Existem inúmeras definições para um *engine*. Entretanto, estas definições convergem em algumas características:

- Permitir que o desenvolvedor possa criar diversos jogos diferentes, usando um mesmo *engine*. É comum, entretanto, que os *engines* sejam catalogados de acordo com os tipos de jogos para os quais eles foram concebidos [EBE 00];
- Poder reaproveitar com facilidade o código desenvolvido em projetos anteriores;
- Abstrair a manipulação de APIs (embora, em muitos casos, o desenvolvedor irá usar as próprias APIs dentro do ambiente do engine, para implementar funcionalidades específicas);
- Possibilitar uma fácil integração entre código e modelagem 3D. Para esta finalidade é comum que os engines apresentem editores de cenas.

Desenvolver um *engine* é uma área complexa, repleta de desafios. Entretanto, o objetivo deste documento não é ensinar a implementar um *engine*, mas sim entendê-los e saber utilizá-los.

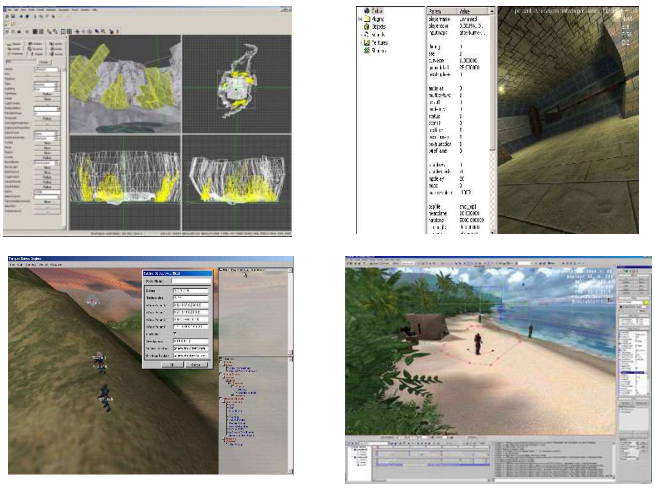


Figura 58 – Exemplo de *level editors* pertencentes a diversos *engines* comerciais.

Normalmente, um *engine* é composto por diversas ferramentas, cada uma responsável por alguma etapa do processo de criação de um jogo. Os componentes mais comuns [ZER 04] de se encontrar em *engines* são os seguintes:

- ***Engine Core:*** Consiste no "coração do *engine*". Este será um programa que executará a aplicação do jogo, manipulará a fase e os objetos, renderizará as cenas, etc. Fazendo-se uma grossa analogia, pode-se dizer que o *engine* core é o sistema operacional do jogo;

- ***Engine SDK:*** É o código fonte do *engine core*. Através dele pode-se alterar o funcionamento do *engine*. Normalmente este componente é o mais protegido e para conseguí-lo, no caso de *engines* comerciais, será necessário comprar o pacote que a empresa oferece. É possível criar jogos sem o SDK de um *engine*, entretanto, os tipos de aplicações possíveis de serem desenvolvidos serão muito mais restritos.
- ***Level Editors:*** Através deste componente será possível unificar modelagens feitas em diversos programas, associá-los à programação, inserir códigos em *scripts*, etc. Em muitos casos, dentro destes editores é possível também criar modelos 3D. Veja Figura 58.
- ***Conversores/Exportadores:*** Os *resources* serão normalmente feitos em diversos softwares comerciais, como se apresentou no capítulo 2. Mais ainda: numa equipe de desenvolvimento grande cada artista poderá criar os elementos no programa de sua preferência. Assim sendo, os *engines* deverão fornecer instrumentos para importar estes modelos para o formato específico do *engine*. Estes conversores poderão ser *plug-ins* instalados nos programas de modelagem 3D ou podem estar incluídos no *level editor;*
- ***Builders:*** Como será discutido algumas operações pré-processamentos sobre os objetos (tais como o BSP, *lightmaps*, portais, PVS, etc.) precisam feitas. Desta maneira, um *engine* fornecerá as ferramentas para realizar estes pré-processamentos. É comum que estejam dentro do *level editor;*
- ***Linguagens Script:*** Grande parte do desenvolvimento da lógica do jogo e da Inteligência Artificial dos elementos dinâmicos será implementada sobre *scripts* e não diretamente sobre o *engine* core. Assim, cada *engine* possuirá sua linguagem de programação script, sendo comum usar linguagens comuns, tais como *JavaScript*, *Python* e LUA.

Pode-se afirmar que um *engine* na realidade é composto por diversos "*sub-engines*", sendo cada um responsável por tratar um tipo de problema envolvido em jogos. Os principais componentes são de renderização, de física, de som e de Inteligência Artificial. Nas próximas subseções cada um desses componentes serão detalhados.

4.1.Engine de Renderização (ou engine 3D)

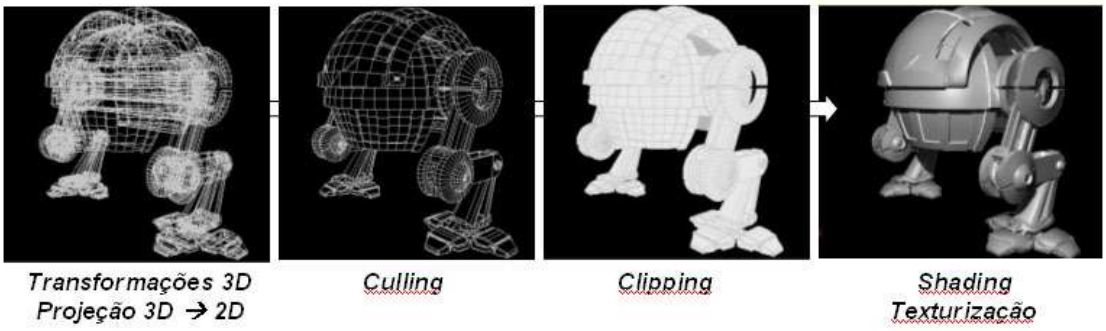


Figura 59 – Etapas mais importantes do *pipeline gráfico*.

O *engine* 3D será responsável basicamente pelo *pipeline gráfico* (Figura 59), que é o processo de gerar imagens 2D partindo de modelos 3D (figura 8). Este processo é dividido em diversas etapas [LAM 03], sendo as mais importantes:

- ***Transformações 3D:*** Nesta etapa aplica-se o movimento aos modelos 3D. Este movimento consiste no seguinte: a cada passo do jogo uma matriz irá acumulando o resultado de todos os movimentos que o objeto sofreu ao longo de seu histórico. Antes de visualizar a cena, será aplicada esta matriz sobre cada vértice que o compõem, posicionando-o no local que lhe corresponde naquele instante;
- ***Projeção 3D → 2D:*** Os vértices que compõem o objeto são coordenadas 3D, porém a imagem do modelo deverá ser desenhada numa superfície bidimensional (tela do computador). Nesta etapa, os vértices do modelo serão projetados sobre o plano de projeção da câmera. É comum encontrar esta etapa do *pipeline* junto com a etapa de transformações 3D, pois em última instância realizar esta projeção consiste numa aplicação de matriz de transformação também;
- ***Culling:*** Existem inúmeras formas de otimizar o processamento gráfico de um jogo. Uma destas consiste nos métodos de culling (Cull em inglês significa "refugo, escolher, selecionar de dentro de um grupo"). Assim, o que as técnicas de *culling* terão de fazer é saber escolher polígonos adequadamente, de forma que numa determinada situação, estejam presentes apenas aqueles que realmente importam para a visualização a partir do ponto em que a câmera se encontra. Pode-se pensar também da seguinte forma: quais dos polígonos de uma cena deverão ser enviados para o *pipeline* da placa gráfica? Obviamente não se deseja enviar algum que não terá nenhuma influência na visualização, mas até que ponto é simples realizar esta escolha, de forma rápida. Existem muitos algoritmos que farão este tipo de escolha. Em muitos casos a eficiência deste procedimento estará atrelada ao tipo de agrupamento e ordem de polígonos (um terreno possui uma distribuição de polígonos completamente diferente de que um personagem ou do que um labirinto). O *culling* pode ser feito em qualquer estágio do *pipeline* gráfico. Entretanto, é preferível eliminar os polígonos que não interessam o quanto antes para evitar processamento desnecessário sob polígonos que futuramente poderão ser eliminados. Vale a pena ressaltar que um método de *culling* não anula outro: podem-se ter os efeitos somados em muitos casos;
- ***Clipping:*** Ao projetar polígonos sobre o plano de projeção da câmera, alguns polígonos cairão totalmente dentro da área da tela e outros cairão parcialmente dentro, ou seja, apenas uma parte do polígono estará na tela de projeção. Para estes polígonos é necessário realizar o *clipping* (recorte), que consiste em criar novas arestas e vértices;
- ***Rasterização (iluminação e texturização):*** Finalmente, a última etapa do processo de renderização consiste em preencher os polígonos adequadamente, aplicando o material com o qual estão definidos. Inicialmente poderia-se pensar em fazer este processo através de um cálculo de iluminação para cada um dos pixels (*per pixel*) do interior de um polígono. Entretanto isto seria demasiadamente caro em termos computacionais. Para

viabilizar este processo realiza-se uma interpolação (rasterização) entre a cor de cada um dos vértices que compõem o polígono. Desta forma, o cálculo de iluminação é feito apenas para cada vértice (*per vertex*) visível da malha. Esta rasterização também poderia demandar bastante tempo de processamento, pois apesar de ser algo simples de ser feito, existem pixels numa tela. Entretanto, este processo é realizado por um hardware específico para esta tarefa, que é a placa gráfica, ou GPU – *Graphic Processor Unit*.

Existem uma série de efeitos de iluminação (*blur*, *bump-mapping*, especularidade) que não podem ser aplicados realizando-se uma rasterização como foi explicada anteriormente. Isto porque para estes efeitos é necessário realizar um cálculo de iluminação *per pixel* e não *per vertex*. Para solucionar este tipo de problema, as placas gráficas mais modernas possuem a capacidade de suportar *pixel shaders* (Figura 60), que são pequenos programas que são executados para cada pixel que será plotado na tela. Para maiores informações sobre esta tecnologia, ver [STL 04].

Figura 60 – Imagens geradas em tempo real utilizando *pixel shaders*.

4.2. Engine de Física

Grande parte da interatividade de um jogo se deve ao funcionamento de algumas leis da física sobre o mundo virtual criado. Assim, ao andar sobre um labirinto e bater numa parede o jogador não pode atravessá-la; ao dar um pulo, o jogador deve colidir com o chão e não continuar caindo para sempre; ao acelerar um carro, sua velocidade deverá ir crescendo gradualmente e não abruptamente.

Os cálculos de física básicos num jogo são:

- ***Colisão:*** Objetos 3D devem colidir com outros. Esta colisão não é trivial de ser realizada para um mundo virtual como é para o mundo real, já que em última instância corresponderia em verificar se cada um dos polígonos de um determinado objeto possui interseção com cada um dos polígonos do restante da cena. Existem inúmeras formas de otimizar estes cálculos (ver [EBE 03]), sendo a mais comum a técnica de *bounding-boxes*, que consiste em englobar cada objeto por uma caixa e calcular a colisão para a caixa e não para a malha completa do objeto;
- ***Resultante de forças:*** Os objetos se movimentam num mundo real devido a aplicação de diversas forças sobre o mesmo. Num ambiente virtual será necessário simular a aplicação de forças de diversas naturezas sobre os objetos, calculando a resultante a cada instante para verificar como será o seu movimento. Para mais detalhes, ver [BER 03].

A *Open Dynamics Engine* (ODE) é uma biblioteca livre para simulação de dinâmica de corpos rígidos, incluindo veículos, ambientes de realidade virtual e

criaturas. É desenvolvida em C++ e pode ser facilmente integrada com uma engine, assim permitindo ser reusada facilmente. Para utilizar basta incluir a biblioteca no *engine* do jogo e utilizar as funções oferecidas por essa. Também é possível usar partes do código para desenvolver uma *engine* personalizada. É possível especificar um mundo com propriedades físicas com seus devidos objetos e a cada ciclo de atualização do jogo todas as forças físicas são aplicadas sob tais objetos considerando o tratamento de colisões. A própria Fly3D [FLY 05], um engine nacional adota a ODE. Outro exemplo de uso da ODE é o *plug-in* para o 3D Studio chamado Ornatrix que permite renderizar cabelos. Todo o modelo físico é baseado nesta biblioteca.

Outra alternativa é o projeto de código-aberto denominado ODF (*Open Dynamics Framework*) [ODF 05] cujo objetivo é criar simulações físicas em *game engine*. Entretanto o ODF executa somente a plataforma Windows, pois utiliza o DirectX.

4.3. Engine de Som

Este componente do engine permitirá o controle sobre os arquivos de som da biblioteca de recursos do jogo. Normalmente utilizará alguma API adequada para manipular este tipo de arquivos, tal como o DirectSound ou o OpenAL. Além de permitir abrir e tocar estes arquivos, os engines de som permitirão um controle de som posicional, permitindo que objetos da cena emitam sons e estes se comportem conforme o posicionamento do objeto na cena.

4.4. Engine de Inteligência Artificial

Entende-se por Inteligência Artificial (IA) para jogos, os programas que descreverão o comportamento de entidades não controladas pelo jogador, tipicamente os NPCs (*Non-Player Characters*).

Na maioria dos casos, o comportamento inteligente desses agentes computacionais é implementado através de máquinas de estados. Os algoritmos de máquinas de estados procuram resolver problemas formalizando diversos possíveis estados em que um elemento pode se encontrar (no caso de uma televisão, por exemplo, poderia-se ter estes estados: Desligada, Acesa e *Stand-by*). A transição de um estado para outro estará atrelado a algum evento que dispare esta mudança (no exemplo anterior, ao apertar a tecla <*on*> a TV muda do estado de desligada para *Stand-by*).

Desta maneira, a inteligência de um personagem de um jogo pode ser descrita por diversos estados em que o mesmo pode se encontrar (no caso de um jogo de ação, poderíamos descrever estes estados como espera, perseguição, ataque, fuga, morte, por exemplo). Este personagem deverá fazer um monitoramento constante sobre acontecimentos que possam disparar a mudança de um estado para outro. Usando o exemplo de um jogo de ação, suponha-se que o fato do jogador aproximar-se mais do que 30m do NPC faça com que o mesmo passe do estado de espera para o estado de perseguição. Veja a figura 61.

Entretanto é perceptível que ocorreram grandes inovações do ponto de vista gráfico, mas aspectos inteligentes das entidades computacionais não foram sofisticados com a mesma velocidade. Muitas vezes um jogo sofisticado graficamente apresenta uma IA mediana. Isto ocorre principalmente pelo fato do custo computacional que os algoritmos inteligentes e de processamento gráfico possuem. Conforme Sewald [SEW

02], técnicas de IA tais como Redes Neuronais Artificiais e Algoritmos Genéticos são pouco utilizadas.

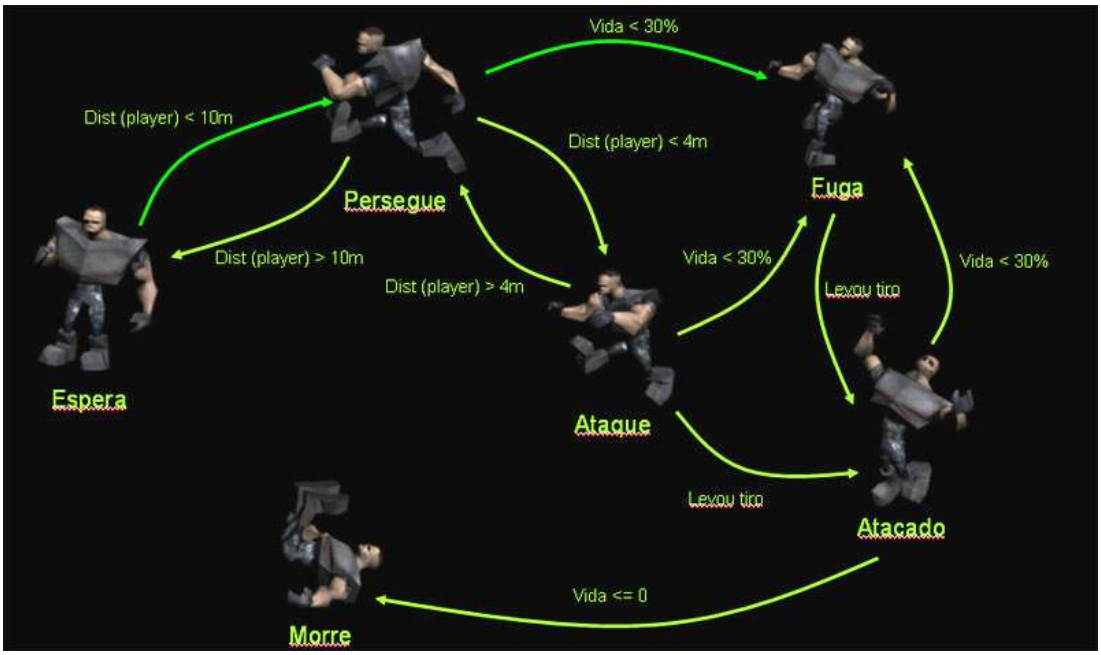


Figura 61 – Exemplo de uma máquina de estados típica para um jogo de ação.

Um exemplo de *toolkit* para o desenvolvimento de agentes inteligentes é o DirectIA [DIR 05]. Trata-se de uma solução comercial genérica que permite desenvolver agentes adaptativos que aprendem através do contato com o ambiente e com usuário. Oferece uma série de módulos de Emoção Artificial, planejamento, comunicação e Aprendizado de Máquinas.

É importante destacar que ferramentas de suporte para criação da IA de *games* ainda é uma área deficitária.

4.5. A Escolha do Engine

Existe uma grande quantidade de *engines*. Em Isacovic [ISA 05] há uma excelente lista de referências sobre os principais e mais completos. Como escolher o *engine* adequado para desenvolver um jogo? Existem diversos fatores que devem ser considerados:

- ***Orçamento disponível:*** Existem *engines* que custam U$ 100,00 e outros que ultrapassam U$ 1 milhão. Os mais caros, além de possuírem todos recursos que um *engine* é capaz de ter, normalmente são programas que possuem uma excelente equipe de apoio, que irá acompanhar a empresa desenvolvedora do início ao fim da sua produção. Este acompanhamento consistirá em desenvolver *plug-ins* específicos, adaptar funcionalidades do *engine* para as necessidades específicas e até dar treinamento para os programadores;
- ***Tipo de jogo a ser desenvolvido:*** Apesar de existirem *engines* que são capazes de construir tipos bastante diferente de jogos, a maioria terá uma série de características que favorecem para o desenvolvimento de um tipo específico de jogo. Por exemplo, o *Quake Engine* permite desenvolver jogos no estilo FPS (*First Person Shooter*).

- ***Milestones:*** O tempo que se possui para produzir um jogo influencia muito na escolha do *engine*. Alguns *engines* permitem que um jogo seja feito em poucos dias, mas implicará numa série de soluções padrões, que o tornam semelhante a muitos outros jogos produzidos com aquele mesmo *engine*;
- ***Plataforma:*** Um jogo pode ser desenvolvido para diversas plataformas, tais como PC, *X-Box*, *GameCube*, *PlayStation*, *MacOS*, entre outras. É fundamental que o *engine* escolhido suporte a plataforma para a qual se está desenvolvendo. Os *engines* mais caros normalmente são capazes de produzir jogos para diversas plataformas;
- ***Documentação oferecida:*** O desenvolvedor de jogos deverá, antes de adotar um engine, verificar como é a documentação do mesmo. Sem uma documentação eficiente, dificilmente o programador será capaz de utilizar adequadamente os recursos oferecidos pelo *engine*;
- ***Ferramentas disponíveis:*** Cada *engine* possuirá conversores e exportadores, que permitirão que sejam utilizados outros programas para programar seu SDK ou para modelar os objetos 3D. É fundamental que o desenvolvedor analise quais as ferramentas com as quais se pode produzir os para o *engine* em questão (se a equipe de modeladores conhece apenas o MAYA, é conveniente que o *engine* a ser adotado suporte o MAYA para os modelos 3D).

Outro ponto que precisa ser destacado é que percebe-se na maioria dos desenvolvedores a preocupação em centralizar todo o processo de criação do *engine* dentro das organizações. Tal atitude acaba ocasionando uma grande quantidade de reescrita de código [ROL 04]. Certamente que este comportamento é indesejado porque representa um maior custo de desenvolvimento e preocupação com características que já foram amplamente implementadas. Ao invés das soluções serem recriadas constantemente, seria mais interessante efetuar um reaproveitamento de componentes já previamente desenvolvidos. Para Rollings & Morris [ROL 04], à medida que a complexidade dos jogos incrementa não será possível que o time interno de desenvolvedores produza toda a codificação de todos os componentes dentro de um prazo hábil. Portanto, a partir desta constatação é possível afirmar que gradativamente a componentização começará a tornar-se mais comum entre os desenvolvedores. Além do setor industrial percebe-se esta cultura no meio acadêmico. Perde-se tempo e incrementa-se custos em conseqüência do não reaproveitamento de soluções existentes.

5. Introdução ao 3D Game Studio

O *3D Game Studio* (Figura 62) [3DG 05] é um *engine* de alto nível de abstração, isto é, possibilita que o desenvolvimento de um *game* seja realizado sem a necessidade de grandes conhecimentos de programação ou conhecimento de algoritmos específicos. Por um lado a produção se torna simples e rápida, mas por outro lado as possibilidades são limitadas e as interfaces gráficas dos jogos terão ligeira semelhança entre si. O custo do engine, a versão extra, é U$89,00, mas permite produzir jogos sem nenhum marca d ´água, entretanto não oferece suporte aos jogos *multiplayers*, simulação física e *shaders*, por exemplo. A versão profissional custa U$899,00, mas sem nenhuma restrição. Executável somente na plataforma Microsoft Windows. Uma característica bastante

importante do engine é o fato de ser extensível usando uma DLL (*Dynamic Link Library*)

Figura 62 – Alguns níveis de jogos feitos no 3D Game Studio

O engine é composto basicamente por 2 módulos:

- ***Model Editor:*** responsável sobretudo pela criação e animação de modelos dinâmicos e de terrenos;
- ***Level Editor:*** para elaborar uma fase por completo.

Para se criar um jogo no *3D Game Studio* é necessário, além de conhecer estas duas ferramentas, conhecer também a linguagem *script* (WDL), que permite criar a lógica do jogo e a IA (Inteligência Artificial) de alguns elementos.

Há diversos tipos de elementos, importantes de serem distinguidos:

- ***Objetos Estáticos***: Corresponderá a toda a geometria que será colocada na BSP e que será pré-processada pelo *engine*. Estes objetos não poderão ter nenhuma transformação durante o jogo. Consistem em geral das paredes, de objetos decorativos, tais como colunas e vigas, acessórios que não sofrerão alteração. Os objetos pré-fabricados e os blocos são os tipos de elementos estáticos mais comuns. Quanto mais objetos puderem ser colocados dentro desta categoria, melhor para a performance do jogo; O formato destes objetos é WMP;
- ***Objetos Dinâmicos:*** Correspondem a modelos que poderão sofrer alterações durante o jogo, tais como NPCs, o *player*, armas, *power-ups*, etc. Estes modelos podem estar animados e podem em geral ser modelados através de

ferramentas terceiras, tais como o 3DS MAX, MAYA, TrueSpace, etc. O formato dos objetos dinâmicos é WMB ou MDL. Apenas estes objetos terão a propriedade *behavior*, que consiste na ação que irá efetuar durante o jogo;

- ***Terrenos:*** Apesar de serem estáticos, possuem um tratamento de otimização especial e portanto correspondem a objetos a parte. O formato de um terreno é HMP e apenas pode ser criado no *Model Editor*;
- ***Sprites:*** Os *sprites* são elementos estáticos, mas constantemente são atualizados para que a sua normal esteja apontando para o *player*. Os *sprites* consistem na geometria mais simples de um jogo, uma vez que são apenas um plano com uma textura. Os *sprites* podem ser do formato PCX, TGA ou BMP.

5.1. *Level Editor*

O *Level Editor* é o principal módulo do *3D Game Studio*. É nele que serão inseridos os elementos estáticos, os dinâmicos, terrenos, luzes, *sprites*, *câmeras*. Também é nele que se fará a associação de programas feitos em WDL com os modelos 3D. Além de tudo isto, é neste módulo que será gerada a BSP, os *light maps*, PVS e demais componentes que permitirão que o jogo seja executado.

Na Figura 63 pode-se ver que a tela do *Level Editor* está dividida em 3 partes:

1 – Área de trabalho: Esta região permite ver a geometria da cena. Estas visões podem ser ortogonais (sem deformação perspectiva) ou perspectivas, dadas pela visão de camera. Normalmente se operará com a visão ortogonal de topo, lado e frente e com a visão perspectiva da camera 3D. Para inserir uma nova vista, basta clicar no menu ***view|Add View***. Para a visão de camera pode-se optar pelo modo de visualização: ***View | Wire Frame*** permite ver a modelagem em formato de arestas e vértices, ***View | Solid*** permite ver a modelagem com os polígonos preenchidos e ***View | Textured*** permite ver a modelagem com as texturas aplicadas. Normalmente este último modo é o mais conveniente de se usar, pois permitirá que se tenha uma idéia mais próxima do resultado final. Com o botão da direita do *mouse* pode realizar-se a operação de *PAN*, que consiste em mover a área que se esta mostrando na região. A operação de *PAN* ocorre simultaneamente em todas as vistas ortogonais, mas separadamente na visão de camera. Para se realizar um *ZOOM* sobre uma região, pode-se usar o *Mouse wheel* ou clicar no ícone de lupa. Sobre a visão de camera é possível também realizar uma operação de *ORBIT*, que consiste em girar a camera ao redor de seu alvo. Isto pode ser feito através do ícone de um olho verde com uma flecha em formato de arco.

2 – *Área de Projeto:* Nesta seção manipula-se os diversos recursos que se encontram presentes no cenário. ***Object*** refere-se a todos os objetos presentes no cenário, ***Views*** relaciona todas as vistas armazenadas (isto é feito clicando-se na pasta com o botão direito do mouse e escolhendo a opção *Add View Settings*, que gravará as configurações das vistas num item mostrado dentro desta pasta), ***Texture*** indica todas as bibliotecas de texturas disponíveis, bem como as imagens de cada biblioteca e ***Resources*** indica todos os elementos que estão sendo usados, tais como sons, *scripts*, mapas, etc, Ao dar um *double-click* sobre um dos recursos, será aberto o arquivo pelo programa "lincado" ao seu tipo.

3 – *Menu e Toolbar*: Nesta área se encotram as funcionalidades e comandos do *Level Editor*.

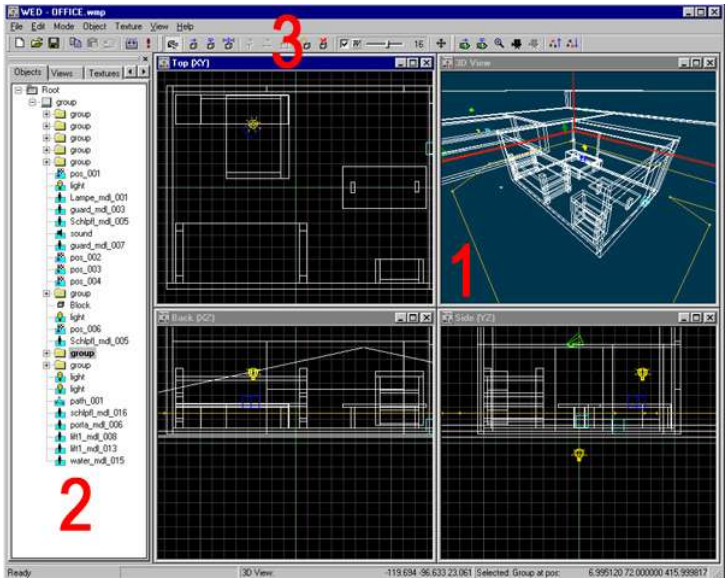


Figura 63: Tela do Level Editor do 3D Game Studio

5.1.1. Modos de Manipulação

Apenas um modo de manipulação pode estar acionado de cada vez. Estes modos podem ser:

- Selecionar (botão com um dedo): permitirá selecionar um determinado objeto ou grupo. Para selecionar basta clicar sobre o objeto ou traçar um retângulo com o *mouse*, fazendo com que todos os objetos que estejam dentro sejam selecionados. Um objeto selecionado aparecerá em vermelho, enquanto os não selecionados aparecem em branco.
- Mover (botão com uma flecha sobre um cubo): Permitirá mover um determinado objeto ou grupo livremente.;
- Rotacionar (botão com um arco sobre um cubo): Permitirá rotacionar um determinado objeto sobre o eixo perpendicular a vista em que se realiza a operação;
- Escalar (botão com setas em cruz sobre um cubo): Irá alterar a escala de um objeto. Ao mover o *mouse* horizontalmente a escala será feita na direção horizontal da vista e ao mover verticalmente a escala será feita na direção vertical. Para que a escala seja igual nas duas direções, basta mover o mouse apertando a tecla **ctrl**.

As transformações podem ser feitas com maior ou menor precisão através do ***snap***. O *snap* consiste em realizar incrementos pré-estabelecidos nas transformações. O valor do *Snap* é estipulado no pequeno *slider* que se encontra na área de botões do *Tool bar*. Caso o valor seja correspondente a 1 as transformações serão contínuas.

5.1.2 Texturas

As texturas são organizadas em bibliotecas, chamadas de ***WADS***, que podem ser manipuladas através do comando ***Textures | Wad Manager***. Para se criar uma nova biblioteca, usa-se o comando ***Add Wad***. A biblioteca construída não possuirá nenhuma textura associada, enquanto o usuário não as inserir. Para inserir texturas deve-se clicar

com o botão direito sobre o espaço correspondente à biblioteca, na área de projeto e escolher o comando ***Add Texture***. Pode-se também criar uma biblioteca que contenha todas as imagens que estão presentes no projeto corrente. Para tanto deve-se escolher o comando ***Build Wad*** no Wad Manager. Para carregar uma biblioteca de texturas para dentro de um determinado projeto se deve usar o comando ***Add Wad*** estando dentro do Wad Manager. Ao realizar isto aparecerá a lista de todas as texturas existentes nesta biblioteca no campo *Textures*, da área de projetos.

Pode-se criar uma lista com as texturas mais usadas num projeto, sendo que estas texturas podem ou não estar em WADs diferentes. Para isto deve-se usar o comando ***Textures*** | ***Texture Bookmarks***.

Para associar uma textura a um determinado objeto, basta selecionar o objeto e em seguida dar um *double click* sobre a textura desejada, na área de projeto. Ao realizar isto, todos os polígonos do objeto receberão esta textura. A textura pode ser manipulada através do painel de propriedades do objeto em questão (Figura 64), tendo em conta que o controle será individual para cada polígono que compõe o objeto. Para escolher o polígono deve-se clicar sobre as setas que estão no canto superior direito da janela. O polígono selecionado irá aparecer em amarelo.

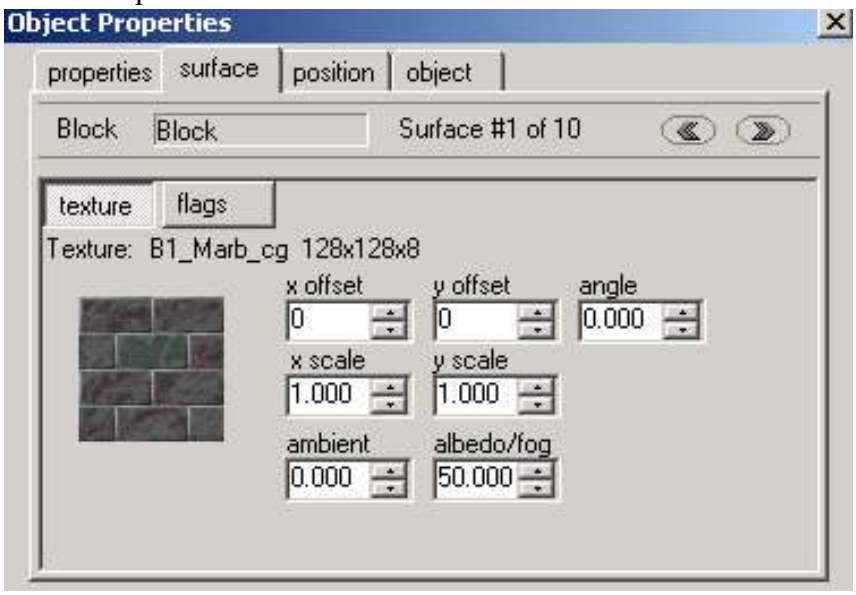


Figura 64 – Painel de propriedades de um objeto

Uma das opções do painel de propriedades se refere justamente à forma como a textura está aplicada. (opção ***surface***). O parâmetro *Offset* permite mover a textura sobre os polígonos, *scale* permite alterar a escala da textura em relação ao objeto e *angle* permite girar a textura sobre os polígonos.

Pode-se aplicar uma textura também apenas para um elemento de um grupo ou apenas para uma das paredes de um bloco. Para tanto deve-se selecionar o elemento ou a parede. Isto é feito selecionando-se o grupo ao qual pertence e depois usar o comando ***Scope Down*** (botão com um grupo de cubos e uma seta para baixo) que irá descendo aos níveis mais baixos da hierarquia. Para voltar ao objeto original, usa-se o comando ***Scope Up***.

5.1.3. Construindo Salas

As salas serão normalmente feitas através de objetos primitivos, ou blocos (cubos, cilindros, pirâmides, etc). Para criar um objeto primitivo usa-se o comando ***Object | Add Primitive*** ou ***Object | Add Cube***. Note-se que os objetos primitivos serão sempre representados por polígonos e deve-se escolher o grau de refinamento dos mesmos. É importante que se escolha sempre o menor número de polígonos possíveis, de forma a otimizar a cena.

Uma vez criado o objeto que servirá de sala, deve-se usar o comando ***Edit | Hollow Block*** para que cada polígono se torne uma parede (composta por 6 polígonos). Isto permitirá que uma parede possa ser vista dos dois lados, sem a necessidade de que o engine crie um estado de normais duplas para cada polígono.

Como na maioria das vezes as salas serão originadas por cubos, já existe um comando capaz de criar salas prontas, ou seja, com o *Hollow Block* já aplicado. (***Object | Add Hollow Cube***)

Para acessar às propriedades de uma determinada primitiva, basta clicar com o botão direito sobre o mesmo e optar por ***properties***. As propriedades que um bloco pode receber são:

- *Passable*: O bloco não terá colisão com os elementos dinâmicos do jogo. (Por exemplo, água de uma piscina);
- *Invisible*: O bloco não será visualizado no jogo, mas poderá ter colisão com os outros elementos.
- *Texture Lock*: A textura permanecerá imóvel quando o objeto sofrer transformações espaciais.

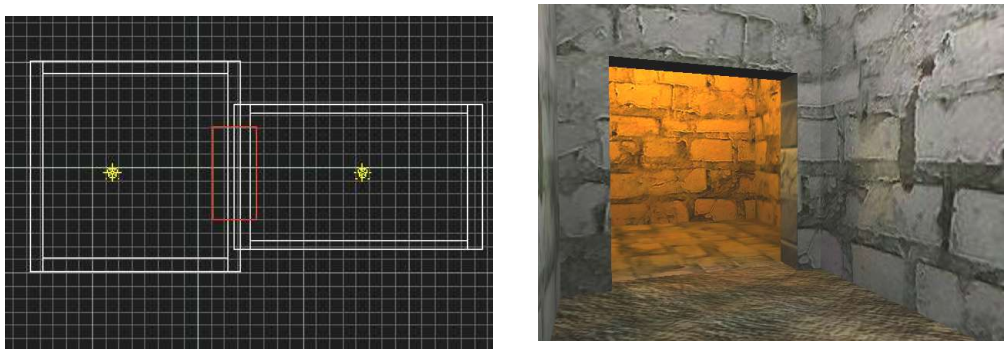


Figura 65 – Dois cubos vistos de cima e com o *Hollow Block* já aplicado. O cubo vermelho é o objeto auxiliar e consiste na parte a ser removida dos dois blocos. Em (b) pode-se ver o resultado da operação.

Após ajustar o tamanho e a posição do bloco que servirá de sala, muito provavelmente se desejará criar buracos que sirvam de passagem de uma sala para outra. Para isto, deve-se utilizar um objeto auxiliar, que poderá ser qualquer uma das primitivas disponíveis. Este objeto deverá ser colocado de tal forma que contenha dentro dele toda a parte da geometria de outros objetos que serão removidos (ver figura 65). Após posicioná-lo adequadamente, deve-se manter este objeto auxiliar selecionado e utilizar o comando ***Edit | CSG Subtract***. Feito isto, o objeto auxiliar ainda estará presente, mas todas as partes das geometrias que estiverem contidas em seu interior

terão desaparecidos. Para completar a operação deve-se deletar ou mover o objeto auxiliar. Esta operação apenas funciona com os objetos estáticos, ou seja os blocos e os objetos pré-fabricados. Não terá nenhum efeito com os objetos dinâmicos, *sprites* e terrenos.

5.2 Objetos Pré-fabricados

Os objetos pré-fabricados são objetos estáticos (WMP) que estão dentro do diretório "..\Gstudio\Prefabs\". Caso o usuário deseje inserir objetos nesta lista, basta colocá-los dentro deste diretório. Em geral estes objetos consistem em vários elementos agrupados, portanto para editá-los deve-se utilizar o comandos de *Scope Down* antes.

5.3 Luzes

As luzes podem ser de dois tipos: estáticas e dinâmicas. Luzes dinâmicas apenas podem ser criadas através de *scripts*, já as estáticas devem ser criadas no *level* editor, através do comando ***Object | Add Light***. Esta luz gerará os *light maps*, que consiste no pré-processamento da iluminação armazenado pelo *engine*. Isto implica que durante o jogo não poderão ser alteradas em hipótese alguma. As propriedades de uma luz são a sua cor e seu alcance (*Range).* Este alcance não implica que a luz termine abruptamente quando chegue ao valor estipulado, mas existirá uma queda gradual, seguindo uma interpolação linear. Apenas as luzes estáticas irão produzir sombras realistas, graças ao pré-processamento do *engine*.

Ao colocar as luzes estáticas, será possível notar que locais fora do alcance da luz ou locais com sobra ficarão totalmente pretos. Isto ocorre quando não há nenhuma luz ambiente. Para inserir luz ambiente deve-se utilizar o comando ***File | Map Properties | Sun***. A luz ambiente não possui um ponto de origem e portanto apenas é necessário dizer a cor do mesmo.

5.4 Compilando e executando uma cena

Para poder executar o jogo é necessário antes compilá-lo, ou seja, gerar a sua BSP, os *lightmaps*, os portais, etc. Para isto, é necessário que a cena esteja fechada, ou seja, o *player* não pode em hipótese alguma ver uma região que não haja nenhum polígono.

Para compilar deve-se usar o comando ***File | Build WMB***. Existem uma série de opções (consultar o manual para uma explicação mais detalhada) mas estão divididas em 2 grupos: Build, que reconstruirá tudo ou *update*, que apenas atualizará alguma mudança. Para compilar a cena por completo deve-se usar a opção *Build Level Map*. Entretanto, em cenários grandes isto poderá demorar muito, portanto caso apenas se tenham inserido objetos dinâmicos desde a última compilação, usa-se *Build Entity Map* ou no caso de apenas se ter alterado algum parâmetro de um objeto dinâmico, basta dar *Update Entities*.

Uma vez compilado, o jogo está pronto para ser executado. Para isto deve-se usar o comando ***File | Run Level***. Pode-se optar por executar o jogo em tela cheia ou apenas numa janela. Durante a execução, pode-se entrar no modo de *debug*, que permite ver os polígonos e uma série de informações importantes, tais como o *frame rate*, memória sendo usada, etc. Isto é feito apertando a tecla ***D*** durante a execução do jogo. (para ver os polígonos deve-se teclar novamente a tecla ***D***)

5.5 Objetos Dinâmicos

Os objetos dinâmicos são todos os objetos que não ficarão parados durante o jogo. O comportamento destes objetos é determinado por uma função especial, chamada de *action* e escrita em *WDL*. Esta ação pode estar pronta e disponibilizada numa das bibliotecas do *3D Game Studio* ou pode ser programada pelo desenvolvedor.

Os objetos dinâmicos possuem algumas propriedades especiais, que não estão presentes nos estáticos. Uma das principais é o *behavior*, que consiste na ação que será colocada no objeto em questão. Para associar um comportamento deve-se chamar a janela de propriedades e em seguida clicar no ícone em forma de pasta, no lado esquerdo do campo *Action*. Ao fazer isto irá aparecer uma série de nomes de ações disponíveis. Estas ações estão numa das bibliotecas de *script* do Game Studio ou estão no script na cena em questão. Não é preciso fazer nada para que apareça a relação destes nomes, basta ter criado um script para o projeto. (Isto é feito pelo comando ***File | Map Properties | Main*** e clicar sobre a pasta *new script*)

Na janela de *behaviors* estão presentes campos para as varáveis disponibilizadas pelo script. Estas variáveis permitem que um mesmo *script* possa ser usado por mais de um elemento dinâmico e que gere ações diferenciadas. (ver manual de WDL para maiores detalhes.)

5.6 *Scripts*

Os *scripts* permitirão desenvolver toda a lógica do jogo. A linguagem utilizada é semelhante à linguagem C, porém com algumas facilidades. Todo projeto deve ter associado a si um arquivo do tipo WDL. Este arquivo pode ser criado automaticamente, bastando para isto usar o comando ***File | Map Properties | Main*** e clicar sobre a pasta *new script*. Pode-se também usar um arquivo já existente (de outro jogo, por exemplo).

O arquivo de *script* possui a seguinte estrutura:

- Definições de diretórios a serem disponibilizados (onde se encontram todos os recursos a serem utilizados pelo nível em questão):

```
path "models";// Path to model subdirectory - if any
path "sounds";// Path to sound subdirectory - if any
path "bmaps";// Path to graphics subdirectory - if any
Path "g:\\PROGRAM FILES\\GSTUDIOPRO\\template";
```

- Definições de bibliotecas a serem disponibilizadas:

```
include <movement.wdl>;
include <messages.wdl>;
include <menu.wdl>;
include <particle.wdl>;
include <doors.wdl>;
```

Estas bibliotecas consistem sobretudo em funções e ações prontas. Ao criar um novo *script* o Game Studio já incluirá as bibliotecas que a versão oferece ao usuário. Assim por exemplo, o arquivo *doors.wdl* possui várias ações prontas para abertura de portas, chaves, utilização de códigos para abrir portas, etc. Deve-se consultar o manual de WDL do Game Studio para uma informação detalhada de todas as ações que se encontram dentro destes arquivos.

- Definições de variáveis globais:

```
var video_mode = 6;     // screen size 640x480
var video_depth = 16; // 16 bit colour D3D mode

bmap horizon_map = <sky01.pcx>;
bmap mountain_map = <zionlow.pcx>;
```

As variáveis globais correspondem àquelas variáveis que podem ser usadas por qualquer função ou pela função principal. Os principais tipos de variáveis que o *script* do Game Studio permite definir são: var (para qualquer tipo numérico), bmap (para imagens), string (para textos em ASCII), music (para arquivos de som tipo MIDI) e sound (para arquivos de som do tipo WAV).

Definições de funções e ações:

```
function start_song()
{
           wait(4);
           if (midi_vol > 0) {
                  play_song_once(testsong,80);
        }
}

action IA_personagem
{
         my._entforce = 0.7;
         my._health = 35;
         my.enable_scan = on;
         my.enable_shoot = on;
         my.enable_detect = on;
         my._state = _state_wait;
         estado_espera();
}
```

As funções e ações descreverão a lógica do comportamento dos objetos dinâmicos e é provavelmente aqui que o programador gastará a maior parte do seu tempo. Para maiores detalhes ver o manual de *scripts* do Game Studio.

Função Main:

```
function main()
{
    // set some common flags and variables
    // announce bad texture sizes and bad wdl code
    warn_level = 2;

     tex_share = on; // map entities share their textures

    // play a wav file
     snd_playfile ("trilha1.wav", 100, 100);

     fog_color = 2;

    // now load the level
     load_level(<castelo.WMB>);
```

```
    // fog = 100;

   // load some global variables, like sound volume
    load_status();

   //     client_move();    // for a possible multiplayer game
   // call further functions here...
}
```

A função Main é a primeira coisa a ser executada ao dar *Run* no jogo. Observe-se que é ela quem carrega o arquivo binário do cenário (load_level), e é ela quem chama todas as definições globais do jogo. Esta função será criada automaticamente ao criar o *script*, e em geral poucas coisas deverão ser acrescentadas a ela.

6. Engines Livres

Conforme foi descrito na seção 2 e 3 existem inúmeras soluções livres que podem ser usadas inclusive no desenvolvimento de softwares fechados e comerciais, pois suas licenças permitem tal utilização. Nesta situação pode-se traçar um paralelo com o desenvolvimento da *OpenGL* que até o seu desenvolvimento cada empresa desenvolvia sua própria biblioteca gráfica despendendo tempo e investimentos para criar um conjunto de primitivas gráficas de baixo nível. Quando a *OpenGL* agregou tais primitivas permitiu que fossem desenvolvidos novos algoritmos gráficos mais significativos, de alto nível, pois já era possível usar estas primitivas mais elementares para projetar soluções mais sofisticadas.

Muitas desenvolvedores despendem grande parte de seus projetos recriando um sistema de tratamento de eventos do sistema, por exemplo, que já fora implementado por diversos SKDs e *engines*. Tais desenvolvedores poderiam partir das soluções já existentes, modificando-as e adaptando-as e redistribuindo-as para comunidade.

6.1. Crystal Space

Crystal Space é um projeto *Open Source* desenvolvido por Jorit Tyberghein [CRY 05] [WEN 02]. É distribuída gratuitamente sob a licença LGPL. Desenvolvida usando C++ é compatível com as seguintes plataformas: Linux, Windows e MacOS. No caso do sistema Windows o *Crystal Space* suporta o *DirectX* e em todos os sistemas oferece suporte a biblioteca *OpenGL*. Destaca-se que a *Crystal Space* não é um *engine*, mas trata-se de um *kit* de desenvolvimento para jogos computadorizados 3D. No guia do desenvolvedor [CRY 05] está anunciado: *"Crystal Space é um pacote de componentes e bibliotecas na qual podem ser úteis para criação de jogos computadorizados"*. Entretanto usando a arquitetura da Crystal Space é possível desenvolver os próprios engines.

Apesar de permitir o desenvolvimento de jogos bidimensionais, a maioria de suas funcionalidades estão voltadas para renderização de gráficos 3 em tempo real. Também apresenta suporte a uma série de formatos de som e persistência dos mundos usando o formato XML (*eXtensible Markup Language*). Também permite carregar arquivos no formato 3DS, MDL, MD2, ASE, OBJ e POV. Existe uma farta documentação incluindo tutoriais, guias, referências da API e listas de discussão.

Segundo o site do *Crystal Space* [CRY 05] existem mais de 650 pessoas que colaboram com o projeto.

Wen [WEN 02] destaca que o *Crystal Space* possui um diferencial em relação os *engines* 3D proprietários, tais como *Quake III* e *Unreal* pelo fato do *Crystal Space* ser uma ferramenta de propósito geral. Ou seja, não é específica para jogos, ela consiste de uma API gráfica que pode ser utilizada por qualquer aplicação multimídia. Este *toolkit* é bastante usado pela comunidade pelo fato desta ser bastante estável e pela grande quantidade de funcionalidades 3D [WEN 02]. A sua principal fraqueza trata-se do mecanismo para detecção de colisão [WEN 02].

Uma característica da arquitetura do *Crystal Space* muito importante é o uso do sistema de *plug-ins* [WEN 02], isto é, um executável funcionará com diversos renderizadores, tais com *OpenGL*, *Direct3D* e *Glide*. Assim cria-se uma camada de abstração da interface oferecida pela ferramenta e a forma de implementação que poderá adotar qualquer mecanismo de renderização. Gradativamente novos *plug-ins* são criados e distribuídos separadamente facilitando o processo de desenvolvimento pelo fato dessa alta modularidade. Para citar um exemplo de aplicação do *Crystal Space*, pode-se destacar o jogo *PlaneShift* [PLA 05]. Esse jogo consiste de um MMORPG livre com gráficos 3D, distribuído gratuitamente e desenvolvido com o *Crystal Space*.

Comparando o *3D Game Studio* e com *Crystal Space* certamente é muito mais fácil e intuitivo de utilizar o primeiro do que o segundo, pois oferece uma série de recursos gráficos em uma IDE projetada para criação de jogos. Entretanto o segundo é multiplataforma é permite a criação de novos *plug-ins* para serem adicionados na infra-estrutura geral do motor. Conforme o jargão computacional a *Crystal Space* é uma ferramenta de mais baixo nível, mas em contrapartida permite uma personalização mais ampla. Veja algumas *screenshots* de *games* criados usando a *Crystal Space* na Figura 66. Veja também o exemplo de *shader* na Figura 67.

Figura 66: Algumas imagens do jogo *PlaneShift* desenvolvido com a *Crystal Space*

Figura 67: Exemplo de CG *Shader Crystal Space*

O *download* do código-fonte pode ser feito a partir do *site* da *Crystal Space* (CS). Atualmente a versão estável é a 0.98r004. Não é disponibilizado nenhum arquivo binário, logo o *Crystal Space* precisa ser compilado pelo próprio usuário. O processo de instalação é trivial, pois trata-se descompactar um arquivo zipado. Nesse arquivo está disponibilizado o manual do programador, o guia da API do *Crystal Space* e todo seu código-fonte.

Para usuários iniciantes o processo de compilação pode ser um pouco complicado, entretanto o manual oficial do *Crystal Space* descreve detalhamente todo o processo de instalação para as plataformas Microsoft Windows, GNU/Linux e MacOS. Os usuários do sistema operacional GNU/Linux encontrarão mais facilidade para compilar pelo fato do sistema possuir nativamente uma forte integração com as ferramentas de compilação para C/C++. Para compilar o *Crystal Space* na plataforma Window primeiramente é necessário instalar no sistema o *Cygwin* ou *MigWM*. Caso seja utilizado o Microsoft Visual Studio não é necessário instalar estas ferramentas. Neste trabalho será descrito como proceder a instalação considerando o Cygwin.

Concluída essa etapa inicial deve-se efetuar o *download* do último release da Win32libs que contem os headers pré-compilados para os usuários do *Cygwin*. O processo de instalação segue o formato padrão de um wizard comumente encontrado nas aplicações Windows. Além dessa biblioteca deve-se fazer o download de um porte do DirectX 8.0 para *Cygwin* e da *Open Dynamics Engine* (ODE). Para instalação do DirectX deve-se descompactar o arquivo e compilá-lo através do comando do *make*. A ODE não precisa ser compilada, pois já estão disponíveis os binários para plataforma Windows.

Por último o *Crystal Space* pode ser compilado. O *script* ./configure deve ser executado. No diretório raíz do *engine* basta digitar jam para iniciar a compilação. A tarefa é bastante demorada. Não elimine os arquivos temporários que foram usados na instalação, pois as libs acabam sendo removidas. Para visualizar a execução de um teste digite na linha de comando *walktest*.

6.2. Ogre3D

O *Ogre3D* (*Object-Oriented Graphics Rendering Engine*) trata-se de uma ferramenta construída em C++, orientado a objetos e distribuído livremente e gratuito sob a licença LGPL. Foi desenvolvido como a dissertação de mestrado de Jeff Plummer na Universidade do Arizona/EUA. Destaca-se que o *Ogre3D* da mesma forma que o *Crystal Space* não tratam-se de um *engine* explicitamente, mas constituem-se de um conjunto de componentes que podem ser usados na criação de um *engine*. Tanto o *Crystal Space* quanto *Ogre3D* oferecem ao desenvolvedor possibilidades de personalização e extensão, desta forma potencializando a reusabilidade de código, abstraindo primitivas de baixo nível e permitindo o desenvolvimento de novas técnicas que poderão ser "plugadas" na arquitetura dos motores.

O *Ogre3D* oferece suporte ao DirectX e a OpenGL, *shaders*, suporte a texturas, além de uma série de outras funcionalidade de suporte para criação de jogos 3D. Na Figura 68 são apresentadas algumas telas de aplicações criando com o *Ogre3D*. O processo de instalação é bem mais simples que a *Crystal Space*, não é preciso recompilar basta fazer *download* do instalador e seguir os passos de uma instalação típica do Windows. É importante destacar que a *Ogre3D* é bastante documentada inclusive com a especificação UML dos seus componentes.

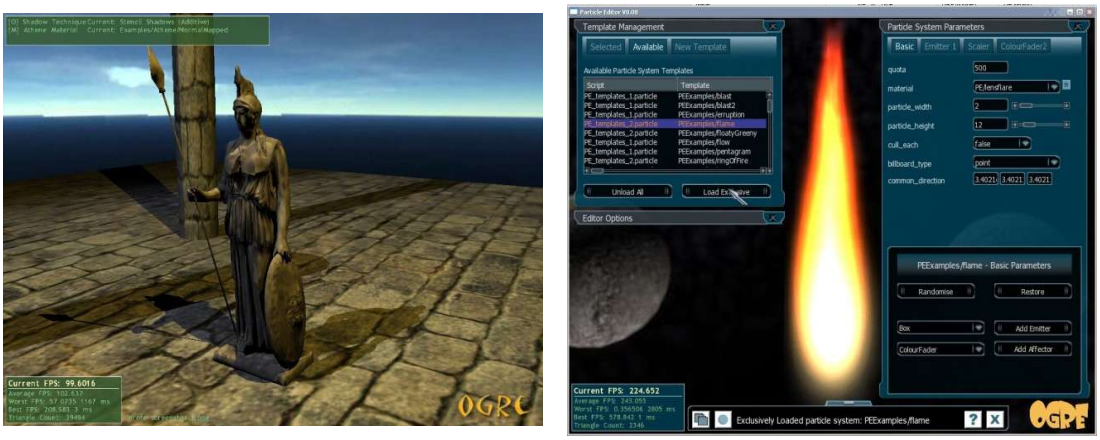


Figura 68: Exemplo de aplicações criadas com Ogre3D.

7. Desenvolvendo Outras Aplicações Usando Ferramentas Relacionadas ao Jogos Computadorizados

O objetivo desta seção é apresentar para comunidade exemplos de aplicações que podem ser desenvolvidas usando a mesma infra-estrutura tecnológica dos *games*. Assim, conforme foi destacado na introdução, muitos resultados aplicados em entretenimento digital podem ser aplicados em outras pesquisas.

Um primeiro grupo de aplicações são os simuladores destacando as aplicações na física que integram *OpenGL* e ODE. Por exemplo, o *OpenSim* (Figura 69) que trata-se de um simulador 3D para robôs autônomos. Outra aplicação é o *Biodesigner* (Figura 70) que trata-se um modelador e visualizador de moléculas.

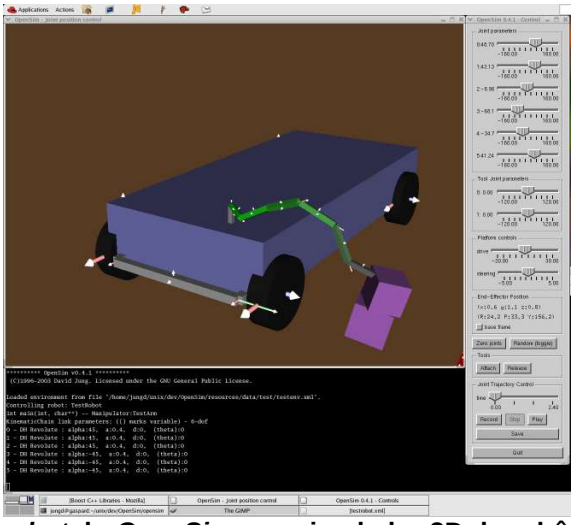


Figura 69: *Screenshot* do *OpenSim*, um simulador 3D de robôs autônomos.

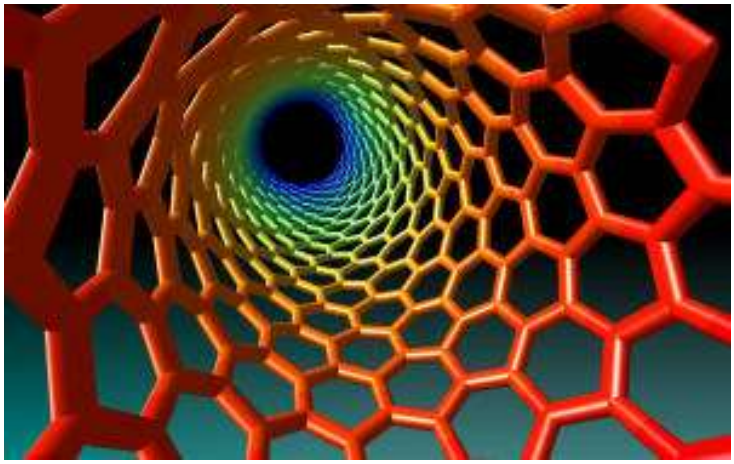


Figura 70: *Screenshot* do *Biodesigner*, representando uma molécula.

Outra tipo de aplicação que potencial de ser investigada na área de interação humano-computador, trata-se de interfaces gráficas 3D. Muito computadores pessoais são equipados com placas aceleradoras 3D que são sub-utilizadas quanto ao gerenciamento da interface gráfica com o usuário. Logo, esta infra-estrutura de hardware pode ser usada para criar ambientes 3D completos. Acredita-se [BEN 99] que interfaces gráficas 3D facilitam a interação entre homem-máquina considerando aspectos cognitivos e perceptivos. Aplicativos de não-entretenimento poderiam explorar as potencialidades gráficas dos jogos, tais como aspectos artísticos, combinação de cores, facilidade de uso para criar interfaces mais intuitivas para os usuários.

O Projeto *Looking Glass* [PRO 05] apoiado pela Sun Microsystems (veja Figura 71) trata-se de um exemplo de interface gráfica 3D. É um projeto livre desenvolvido em Java que permite construir um sistema gráfico para *desktops* baseados em tecnologia 3D. Ao invés das tradicionais janelas e cliques do *mouse* sob esses componentes, o usuário contará como uma coleção de objetos 3D para serem manipulados. No caso da Figura 71 é apresentada uma coleção de CDs que ao invés de ser mostrada na tradicional forma de uma lista de um cadastro de CDs, esses são apresentados para o usuário como uma coleção de CDs tridimensionais.

Figura 71: Exemplo da interface gráfica do Projeto *Looking Glass*

Por último, destaca-se os jogos desenvolvidos com propósitos educacionais. Para um compreensão melhor desta temática recomenda-se a leitura de Bittencourt [BIT 04], Bittencourt & Giraffa [BIT 03] e Clua & Bittencourt [CLU 04]. Para um pleno entendimento deste tópico é necessário abordar alguns conceitos filosóficos, sociológicos e educacionais que simplesmente por uma razão de escopo não serão abordados neste trabalho. Entretanto considera-se que o contexto do século XXI é bastante diferenciado do contexto industrial de algumas décadas passadas. Atualmente vive-se em uma sociedade cibercultural, logo deve-se preocupar com a formação de sujeitos pós-industriais

Compreendendo-se a importância dos jogos computadorizados, com ou sem fins pedagógicos explícitos, no processo de ensino-aprendizagem seria interessante fomentar a produção de novos títulos e com qualidade que poderão ser usados em ambientes de aprendizagem, tais como salas de aula. Entretanto tais jogos não devem adotar o mesmo padrão dos conhecidos jogos didáticos, pois estes não são atrativos para os jovens, pois não criam uma sensação de imersão conforme Clua et al [CLU 02]. Portanto tais técnicas de desenvolvimento de *games* 3D podem ser bem utilizadas para criar jogos que sejam lúdicos, divertidos, atrativos e que possam atender, como *background*, algum ou uma série de objetivos pedagógicos. Pode-se citar o projeto *Games-To-Teach* [GAM 05] conduzido pelo *Massachusetts Institute Technology* que propõe uma série de novos estilos de jogos voltados para uma nova geração de aprendizes. Na Figura 72 é apresentada uma *screenshot* do jogo *Revolution* que aborda a história da Guerra Civil Americana na forma de um MMORPG (*Massive Multiplayer Online Role-Playing Games*).

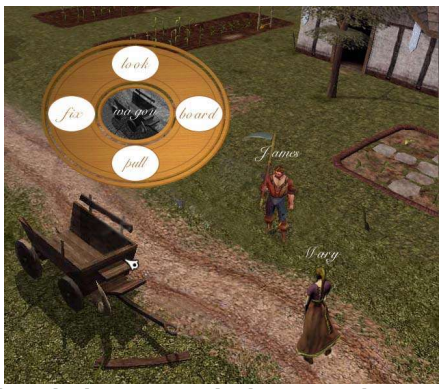


Figura 72: *Screenshot* do jogo *Revolution – Project Game-to-Teach* (MIT)

8. Uma Visão do Mercado Nacional e Internacional sobre Jogos Computadorizados

Embora a produção nacional tenha seu início na década de 80, apenas recentemente o setor de desenvolvimento de jogos no país atingiu massa crítica e passou a ser considerado como segmento industrial e reconhecido como área de investigação técnico-científico.

Num levantamento realizado pela ABRAGAMES em setembro de 2004, haviam no país 37 empresas dedicadas ao desenvolvimento de jogos eletrônicos (destas, 10 surgiram nos últimos 2 anos): 44 Bico Largo, Akan, Amok Entertainment, Atlantis Studios, Bermuda Soft, Bitcrafters, Calibre Games, Continuum Entertainment, Cyprux, Délirus Entertainment, Devworks Game Technology, Eons Games, Espaço Informática, Finalboss, FourX, Green Land Studios, Hoplon Infotainment, Ignis Games, Inflammation Entretenimento, Jynx Playware, LocZ Games, Meantime Mobile Creations, Nology, Nyx Entertainment, Oniria Entertainment, Palmsoft, Perceptum, Preloud, Sevenway Mobile Application, Sioux, South Logic Studios, Staridia Softworks, Sylic Games, Tilt.net, Vortek, Z80 e Ziqx.

Na data do levantamento haviam 25 jogos em desenvolvimento, e foram relacionados 35 jogos nacionais (apenas para PC) lançados nos últimos 2 anos [ABR 05].

Também deve-se destacar a importância da criação de grupos de estudos principalmente pelos estudantes da criação. Deve-se evitar a uma abordagem autoral, ou seja, um único sujeito desenvolvendo o jogo por completo. Idealmente é mais interessante a formação de grupos interdisciplinares, cujos membros dominam áreas diferentes da produção de um jogo. Assim, potencializam-se as chances de desenvolver um protótipo que futuramente pode tornar-se um *game* comercial estabelecendo uma parceria com alguma empresa atuante no mercado. Além de poder participar dos diferentes concursos que estão sendo promovidos na atualidade.

9. Perspectivas e Considerações Finais

A indústria mundial de jogos eletrônicos apresenta atualmente uma impressionante curva de crescimento (apenas em 2004 a indústria do entretenimento digital movimentou cerca de 40 bilhões de dólares, ultrapassando de forma significativa o faturamento do cinema). Hoje os jogos eletrônicos estão presentes em vários dispositivos digitais interativos, como telefones celulares, PDAs, computador (jogos on line, CD, DVD), além dos próprios consoles de videogames. Uma das tendências do mercado de tecnologia para entretenimento é que as residências possuam um conversor digital que funcione como um centro de entretenimento familiar, que habilitará o acesso a serviços como TV digital, filmes e música sob demanda, Internet, jogos (individuais ou multiplayer, inclusive online), gravação de programas, compra de conteúdos digitais individualizados (filmes, jogos, shows, esportes, educação, outros), entre outras aplicações. Soma-se a isto o fato de que os jogos estão sendo considerados uma tecnologia estratégica em muitos países, porque um jogo é essencialmente uma ferramenta de simulação e visualização que pode ser usada para entretenimento, mas também para outros propósitos específicos, tais como treinamento, simuladores de vôo, treinamento militar, visualização médica e científica e educação.

Alia-se a esta visão estratégica, o fato de que o domínio do processo de criação de conteúdo para entretenimento digital será vital para a criação de conteúdo para a TV Digital e TV Interativa. De uma forma geral, os *games* vêm ditando as tendências de modelo de interação. As gerações que jogam jogos individualmente ou *online* têm uma percepção avançada sobre interatividade e uma acuidade gráfica diferenciada. A indústria dos *games* sempre investiu em interfaces muito bem elaboradas, de alta qualidade visual, e também vem estudando com muita propriedade questões comportamentais referentes a interação homem-máquina e usabilidade. Certamente os *games* estabeleceram padrões e continuam a impor patamares de qualidade para a indústria do audiovisual, que devem ser considerados com muito cuidado.

Este panorama despertou o interesse de países fora do restrito círculo de desenvolvedores tradicionais, representado pelos Estados Unidos, Japão, Canadá e países da Europa Ocidental. Atentos a isto, aqueles países resolveram explorar suas vantagens competitivas (alguns tinham tradição tecnológica, outros baixos custos de produção) e traçaram políticas públicas para desenvolver suas indústrias locais. Os melhores exemplos são: Coréia do Sul, Austrália, Índia, China e Cingapura [ABR 05]. Assim, é fundamental que no Brasil surjam e se solidifiquem grupos de pesquisas, projetos, laboratórios e conseqüentemente empresas e modelos de negócios neste setor.

Espera-se que no término do presente trabalho tenha sido possível apresentar uma contextualização da evolução, importância e processo de desenvolvimento dos jogos computadorizados, especificamente os jogos 3D. Assim, despertando o interesse na comunidade acadêmica para o desenvolvimento de pesquisas no segmento de Entretenimento Digital constituindo um grande nicho para efetuar pesquisas aplicadas. Para as outras áreas da Ciência da Computação conforme foi visto, os jogos computadorizados são excelentes aplicações para o desenvolvimento de modelos computacionais. Destaca-se que outras áreas do conhecimento também são beneficiadas com pesquisas nesse setor, tais como a Educação visando o desenvolvimento de jogos aplicados ao processo de ensino-aprendizagem.

Referências Bibliográficas

3D Game Studio. Disponível em: <http://www.3dgamestudio.com> Acesso em: 20 abr. 2005.

ABRAGAMES. Disponível em: <http://www.abragames.org.br/> Acesso em: 15 mai. 2005.

Alias. Disponível em: <http://www.aliaswavefront.com> Acesso em: 30 abr. 2005.

Softimage|XSI. Disponível em: <http://www.softimage.com/products/xsi/v4/> Acesso em: 15 mai. 2005

Audacity. Disponível em: <http://audacity.sourceforge.net/> Acesso em: 15 mai. 2005.

Battaiola, André L. Jogos por Computador – Histórico, Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação In: XIX Jornada de Atualização em Informática. Curitiba:SBC, Julho/2000, v. 2. pp. 83-122.

Battaiola, André L.; Elias, Nassim C.; Domingues, Rodrigo G. et al Desenvolvimento de um Software Educacional com Base em Conceitos de Jogos de Computador In: XIII Simpósio Brasileiro de Informática na Educação. São Leopoldo: SBC, 2002, pp. 282-290.

BenHajji, Farid; Erik, Dybner. 3D Graphical User Interfaces. Estocolmo: Universidade de Estocolmo, Relatório Técnico, 1999, 50 p.

Bergen, Gino Van Den. Collision Detection in Interactive 3D Environments (Morgan Kaufmann Series in Interactive 3D Technology). Morgan Kaufmann. October 2003.

Bittencourt, João R.; Giraffa, Lucia M. A Utilização dos Role-Playing Games Digitais no Processo de Ensino-Aprendizagem. 2003. Relatório Técnico n°031, PPGCC/FACIN/PUCRS, Porto Alegre, 2003, 62 p.

Bittencourt, João R. Um Framework para Criação de Jogos Computadorizados Multiplataforma. Porto Alegre: PUCRS/PPGCC, 2004, 199 p.

Blender3D. Disponível em: <http://www.blender3d.com> Acesso 15 mai. 2005.

Crawford, Chris. Chris Crawford on Game Design. New Riders Publishers, 2004.

Clua, Esteban Walter Gonzalez; Junior, Carlo Luciano de Luca; Nabais, Rodrigo José de Moraes. Importância e Impacto dos Jogos Educativos na Sociedade. In: I Workshop Brasileiro de Jogos e Entretenimento Digital. Proceedings. SBC: Fortaleza, 2002.

Clua, Esteban W.G.; Bittencourt, João R. In: XV Simpósio Brasileiro de Informática na Educação. Manaus: SBC, Novembro/2004.

Crystal Space. Disponível em: <http://crystal.sourceforge.net/> Acesso em: 22 abr. 2005.

DAZ Production. Disponível em: <http://www.daz3d.com/> Acesso em: 15 mai. 2005.

DirectIA. Disponível em: <http://www.directia.com> Acesso em: 15 mai. 2005.

Discreet. Disponível em: <http://www.discreet.com/> Acesso em: 25 abr. 2005.

Eberly David H., 3D Game Engine Design : A Practical Approach to Real-Time Computer Graphics. Morgan Kaufmann, September, 2000.

Eberly David H. Game Physics (Interactive 3d Technology Series). Morgan Kaufmann; Bk&CD-Rom edition. December, 2003.

E-On Software. Disponível em: <www.e-onsoftware.com> Acesso em: 10 mai. 2005.

Fly 3D. Disponível em: <http://www.fly3d.com.br> Acesso: 15 abr. 2005.

Games to Teach Project. Desenvolvido pelo MIT. Disponível em: <http://cms.mit.edu/games/education/> Acesso em: 28 abr. 2005.

GIMP. Disponível em: <http://www.gimp.org/> Acesso em: 15 mai. 2005.

Isacovic, Karsten. 3D Engine List. Disponível em: <http://cg.cs.tu-berlin.de/~ki/engines.html> Acesso em: 10 mai. 2005.

Laird, John. Van Lent, Michael. Human-level AI's Killer Application: Interactive Computer Games In: AI Magazine, v.22, n.2, 2001, pp. 15-25.

LaMothe, André. Tricks of the 3D Game Programming Gurus-Advanced 3D Graphics and Rasterization, Pearson Education, June 2003.

NewTek. Disponível em: <http://www.newtek.com> Acesso em: 10 mai. 2005.

ODF Rocket. Disponível em: <http://www.physicstools.org/> Acesso em: 15 mai. 2005.

Ogre3D. Disponível em: <http://www.ogre3d.org> Acesso em: 17 mai. 2005.

Pandromeda. Disponível em: <http://www.pandromeda.com> Acesso em: 10 mai. 2005.

PlaneShift. Disponível em: <http://www.planeshift.it>. Acesso em: 25 abr. 2005.

Project Looking Glass. Disponível em: <https://lg3d.dev.java.net> Acesso em: 15 mai. 2005.

Rollings, Andrew and Morris, Dave. Game Architecture and Design: A New Edition. New Riders Publishers, 2004.

Salen, Katie; Zimmerman, Eric. Rules of play : game design fundamentals. Cambridge: MIT, 2004. 672 p.

Schwab, Brian. AI Game Engine Programming (Game Development Series). Charles River Media, September, 2004.

Sewald, Leonardo. SCORE: Uma proposta para projeto e implementação do comportamento de agentes cognitivos aplicado a jogos computadorizados interativos. Porto Alegre: PUCRS/PPGCC, 2002, 75 p.

Sony MediaSoftware. Disponível em: <http://mediasoftware.sonypictures.com/> Acesso em: 16 mai. 2005.

St-Laurent, Sebastien. Shaders for Game Programmers and Artists. Muska & Lipman/Premier-Trade; 1st edition.May, 2004.

Terragen. Disponível em: <www.planetside.co.uk/terragen/> Acesso em: 10 mai. 2005.

Tsingos, Nicolas; Gallo, Emmanuel; Drettakis, George In: ACM Transactions on Graphics, n. 3. vol. 23, July/2004

Wen, Howard – Crystal. Crystal Space: An Open Source 3D Graphics Engine. In: Linux Journal v.2002 issue 97, Maio/2002. Disponível em: <http://www.linuxjournal.com/article.php?sid=5514>. Acesso em: 04 mar. 2005.

Xavier, Guilherme. Eletroludens. Trabalho de conclusão de curso feito no departamento de artes da PUC-Rio, 2003

Zerbst, Stefan and Düvel, Oliver. 3D Game Engine Programming. Thomson Course Technology, Premier press, 2004.