UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO DE TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Uso de Inteligência Artificial na animação de Agentes Autônomos Inteligentes em jogos

Orientador
Raul Sidnei Wazlawick

Membros da Banca
Mário Antônio Ribeiro Dantas
Aldo von Wangenheim

Estudante
Brian Schmitz Tani

Ciências da Computação

TRABALHO DE CONCLUSÃO DE CURSO

Florianópolis - 2004

Uso de Inteligência Artificial na animação de Agentes Autônomos Inteligentes em jogos

Brian Schmitz Tani

Este Trabalho de Conclusão de Curso foi aprovado em sua forma final pelo Curso de Ciência da Computação da Universidade Federal de Santa Catarina.

_____

Prof. Raul Sidnei Wazlawick, Dr.

_____

Prof. José Mazzuco Júnior, Dr.

Banca Examinadora

_____

Prof. Aldo von Wangenheim, Dr.

_____

Prof. Mário Antônio Ribeiro Dantas, Dr.

Resumo

Esta pesquisa visa estudar diversas técnicas de animação de agentes inteligentes, buscando conciliar os estudos com o ambiente propício para jogos em terceira dimensão. Além de verificar quais técnicas estão sendo usadas na área de inteligência artificial para atingir tal objetivo. Sendo que, ao final, um modelo será implementado.

Palavras chave: Agentes Autônomos, Agentes Autônomos Inteligentes, Animação de Agentes.

Abstract

This research aims to study several techniques for animating intelligent agents, conciliating those studies with an environment suited for 3d games. Furthermore, we will verify which techniques are being used in the artificial intelligence area to achieve this goal.

Keywords: Autonomous Agent, Intelligent Autonomous Agent, Agent Animation.

Sumário

p.
1. INTRODUÇÃO 7
1.1 Justificativa 8
1.1.1 Social 8
1.1.2 Econômica 9
1.1.3 Científica 9
1.1.4 Pessoal 10
1.2 Objetivos 11
1.2.1 Objetivo geral 11
1.2.2 Objetivos específicos 11
2. METODOLOGIA 12
2.1 Conceitos 12
2.1.1 Método científico 12
2.1.2 Pesquisa 12
2.2 Prototipagem 12
2.3.1 Tipos de prototipagem 13
2.2.2 Etapas da prototipagem 14
2.3 Etapas do projeto 15
3. FUNDAMENTAÇÃO TEÓRICA 16
3.1 Sobre RPG 16
3.2 Sobre motores tridimensionais 17
3.3 Sobre inteligência artificial 18
3.4 Sobre agentes 20
3.4.1 Agentes autônomos 21
3.5 Sobre a animação de agentes 25
3.5.1 Modelos tridimensionais 28
3.6 Sobre a inteligência artificial usada em jogos 31
3.6.1 Navegação 31
3.6.2 Máquina de estados finitos 32
3.6.3 Aprendizado 33
4. DESENVOLVIMENTO DO PROJETO 35
4.1 Requisitos 35
4.2 Implementação 36
4.2.1 Motor gráfico 36
4.2.2 Modelo de Animação 39
4.2.3 Estrutura de um Jogo 40
4.2.4 Inteligência Artificial 43
4.2.5 Ferramentas utilizadas 46
4.3 Resultados 46
5. CONCLUSÕES 50
6. FONTES BIBLIOGRÁFICAS 52
7. ANEXOS 56
7.1 Anexo I – Artigo 56
7.2 Anexo II – Código-Fonte do Projeto 73
7.2.1 – AI_States.h 73
7.2.2 – AI_Controller.cpp 73
7.2.3 – AI_Controller.h 82
7.2.4 – LevelManager.h 85

7.2.5 – LevelManager.cpp 87
7.2.6 – RPG_Creature.h 94
7.2.7 – RPG_Creature.cpp 95

1. INTRODUÇÃO

O mercado brasileiro de jogos está crescendo, se destacando entre vários países, porém ainda é pequeno se comparado com países como Estados Unidos ou Japão, líderes na indústria de jogos. Este crescimento no interesse por jogos proporciona um impulso na indústria nacional. Esta se vê motivada a produzir mais títulos e criar mais empresas. Gera-se mais capital e isso fortalece o, já imenso, negócio multibilionário de jogos.

No âmbito de desenvolvimento de jogos o Brasil ainda está engatinhando, com somente algumas empresas se destacando como Greenland Studio em São Paulo que produziu o jogo Big Brother Brasil, Continuum em Curitiba-PR que produziu o famoso Outlive, e a IGNIS Games, também no Paraná, que está produzindo um jogo centrado no folclore brasileiro e chamado Erynis.

Na UFSC já foram feitos diversos trabalhos de conclusão de curso na área de jogos. Existe também, pelo menos, uma empresa de jogos em uma incubadora aqui em Florianópolis. E isso tende a aumentar à medida que mais projetos apareçam.

Como será visto adiante o potencial mundial para o mercado de jogos é enorme e isso indica que o mercado brasileiro poderia se beneficiar com jogos mais baratos produzidos aqui mesmo, tanto para consoles quanto para PCs, sendo o último mais viável.

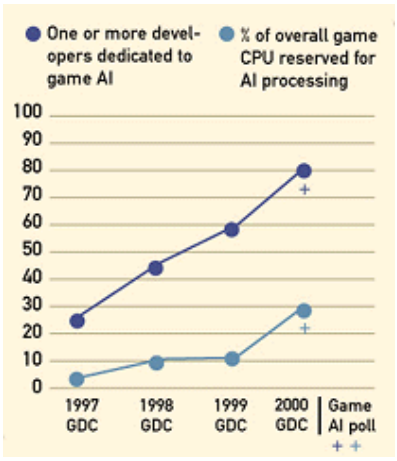


Gráfico que mostra o crescimento no uso de IA ao longo dos anos em jogos.

Em um estudo feito pela empresa Gamasutra (2000) foi visto que o tempo de processamento usado para funções relacionadas a IA[1] vem crescendo em

[1] IA – Inteligência Artificial

porcentagem, e o número programadores associados à programação usando especificamente conceitos IA está aumentando. Disto pode se concluir que a importância dada à inteligência artificial vem crescendo ao longo dos anos e é visto no gráfico apresentado no estudo da referida empresa. E não é para menos, a inteligência artificial propicia a criação de agentes autônomos mais inteligentes, no sentido de fazer uma melhor interação com o jogador, seja “sentindo” emoções e reagindo de acordo como nos jogos de simulação de relacionamentos (The Sims – Maxis 2000), seja combatendo com estratégias próprias num jogo de ação. Em ambos os casos, um alto grau de complexidade em técnicas avançadas de IA melhoram esta interação.

Todavia é necessário tomar-se cuidado, pois quanto maior a complexidade, maior o grau de realismo, porém maior ainda é o custo de processamento. A alternativa é usar linguagem de script que tem um processamento mais leve, no entanto limitam os níveis de interação com o jogador. No entanto com o aumento da capacidade de processamento dos computadores e o aumento no interesse por níveis mais elevados de complexidade das respostas, a primeira opção é cada vez mais procurada.

1.1 Justificativa

1.1.1 Social

Diversão é algo importante, e o entretenimento sob forma de videogame vem crescendo de forma assombrosa no mercado brasileiro. Desta forma, o desenvolvimento de jogos melhores tem potencial para ser absorvido rapidamente pelo mercado.

1.1.2 Econômica

Segundo a previsão feita pela DFC Intelligence[2], o mercado mundial de videogame deverá crescer de 32% a 45% até 2007, chegando a um mercado de U$28.4bi até U$30.1bi, não incluindo consumo de acessórios, aluguéis e jogos usados. Este pode acumular mais U$5 bi anualmente. Razão mais que suficiente para o Brasil investir na indústria de jogos.

Mas não é tudo, como relata a Revista Electronic Gaming Monthly Brasil do mês de Abril de 2003, a Nintendo do Brasil perdeu sua representante, a Gradiente, e por isso não pode mais produzir consoles aqui. Além disso, a Sony está impedida de se estabelecer no Brasil por problemas legais com a própria Gradiente e desta forma não pode produzir seu Playstation 2 aqui no Brasil. E a Microsoft, que produz o Xbox, ainda não decidiu como vai encarar o mercado brasileiro. Mesmo assim, com todos estes impedimentos o Brasil é sétimo em número de usuários na rede “Xbox Live!”. Mesmo tendo de comprar consoles importados e pagar taxas mensais em dólares.

1.1.3 Científica

O processo de criação de jogos envolve diversas áreas do conhecimento, desde o campo da psicologia até o desenvolvimento e pesquisa de tecnologia de ponta. Atualmente o processo de criação dos jogos de ponta se assemelha à produção de um filme mesclado ao desenvolvimento de software de alto nível.

No organograma de um uma equipe de desenvolvimento, como é o caso da Lucas Arts, existe o diretor geral, o qual tem como função supervisionar o processo de criação do jogo que ele esta sendo responsável. Existem ainda, como na

[2] DFC Intelligence: Empresa especializada em fazer previsões de Mercado para a industria de jogos e entretenimento. http://www.dfcint.com/

produção de um filme, a direção de arte, sonoplastia, escritores e produtores. Porém, além dessas pessoas, existe o departamento de programação, que lida com o desenvolvimento do software que vai se tornar o jogo. Uma das áreas que este departamento cobre é o desenvolvimento da inteligência artificial usada nos jogos, o que envolve muita pesquisa e meses de programação. Outras áreas envolvem programação de scripts (e desenvolvimento de SDKs [3]), desenvolvimento de melhores motores gráficos, ou aperfeiçoamento de um existente; criação e planejamento de fases ou níveis[4].

Todos os conceitos de engenharia de software estão presentes neste processo, e alguns aspectos provêm grandes desafios. Um deles é o desenvolvimento de agentes autônomos que sejam inteligentes e propiciem uma melhor experiência para o jogador.

1.1.4 Pessoal

O convívio com diversos tipos de jogos criou um grande interesse nesta maravilhosa forma de entretenimento. Além do lado lúdico dos jogos têm-se o interesse científico de experimentar das técnicas de produção de jogos, mais especificamente jogos usando motores gráficos tridimensionais.

Com a indústria de jogos cada vez mais avançada, jogos demoram mais e demandam mais conhecimento e tecnologia para serem produzidos. Isto é constatado pela crescente complexidade das tecnologias recentemente usadas, e pela grande especialização que vem sendo cada vez mais necessária para se integrar ao mercado de jogos. Antigamente os jogos eram produzidos por

[3] SDK – Software Development Kit, ou, kit de desenvolvimento de software. Geralmente distribuídos por empresas para que empresas independentes possam produzir software para a sua plataforma, seja ele um hardware ou software (jogos para um console, por exemplo).
[4] Fases ou níveis – são áreas nos jogos as quais o jogador tem de passar e cumprir certas tarefas. São conhecidas assim pois antigamente essas áreas eram acessadas em ordem, como níveis ou fases de um processo.

programadores entusiastas que se divertiam produzindo jogos sozinhos ou com pequenas empresas informais. Agora, como é o exemplo da LucasArts entertainment, e muitas outras, o processo de criação de jogos se assemelha a um misto de produção de filmes de hollywood e desenvolvimento de software de ponta, com diferentes departamentos. Desta forma é vital preparar-se para o mercado tendo contato com essas tecnologias.

1.2 Objetivos

1.2.1 Objetivo geral

Desenvolver um agente autônomo inteligente animado utilizando um modelo 3d[5] para jogos tipo RPG[6] com motores gráficos em três dimensões.

1.2.2 Objetivos específicos

- Estudar o estado da arte no uso de inteligência artificial em jogos 3d, além do processo de desenvolvimento de um agente inteligente.
- Averiguar as possibilidades de implementação estudando os requisitos fundamentais no desenvolvimento de um agente inteligente. Isto é, adquirir um motor gráfico que atenda as especificações, criar ou adquirir estruturas fundamentais que possibilitem o desenvolvimento propriamente dito do agente inteligente, e determinar se é possível implementar tal agente dentro desses modelos.
- Modelar um comportamento para o agente usando técnicas apropriadas e dentro das limitações das ferramentas.
- Implementar um modelo de inteligência artificial na estrutura escolhida.

[5] 3d – três dimensões.
[6] RPG – Role Playing Game, ou Jogo de Interpretação de Papéis.

2. METODOLOGIA

2.1 Conceitos

2.1.1 Método científico

Segundo Cervo e Berviam (1996), e respaldado na obra de Falshin (1993) e Salomon (1978), o método científico é basicamente um método de investigação que consiste em um conjunto de processos. Estes processos foram utilizados por vários pesquisadores através dos tempos e reunidos de forma a criar um procedimento sistemático e ordenado de pesquisa científica.

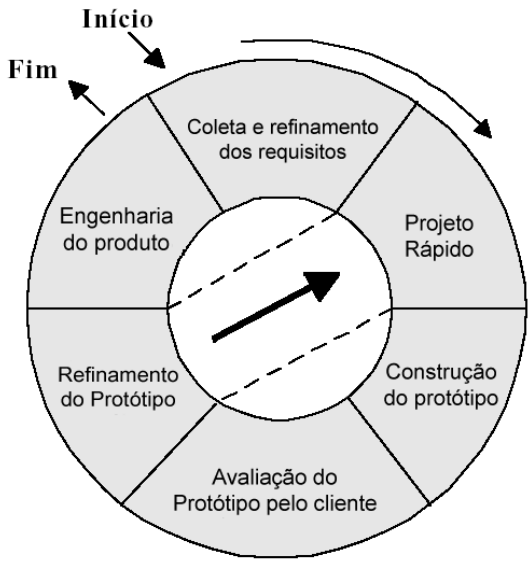
2.1.2 Pesquisa

Nestas mesmas obras a pesquisa é definida como uma atividade que envolve o intelecto humano na busca de soluções para problemas através de procedimentos científicos. Assim parte de uma dúvida ou problema inicial formulado e fundamentando-se no uso do método científico tenta alcançar uma solução.

2.2 Prototipagem

No processo de prototipagem desenvolvem-se implementações parciais do software com aspectos pouco entendidos do produto final, esta implementação é testada e o feedback[7] é usado para melhorar os aspectos falhos ou reavaliar requisitos iniciais. Desta forma, o protótipo passa por um processo de refinamento, e depois de um projeto rápido, um outro protótipo é construído. Este é reavaliado e o processo se repete, até o produto final ser alcançado. A figura a seguir ilustra tal processo.

[7] Feedback – é a informação sobre o resultado de um experimento, etc.; resposta.



Modelo do processo de prototipagem

Para minimizar o número de protótipos desenvolvidos, um estudo aprofundado dos requisitos é necessário.

2.3.1 Tipos de prototipagem

Existem basicamente dois tipos de prototipagem, a prototipagem descartável e a prototipagem evolucionária.

Prototipagem Descartável: Nesta abordagem os protótipos são feitos rapidamente, visando uma comunicação rápida com o cliente. Davis (1995) apud Huang (1998) diz que geralmente se usa a prototipagem descartável quando não se tem total conhecimento sobre alguns aspectos críticos.

Prototipagem Evolucionária: Nesta abordagem os protótipos são reutilizados e “evoluem” até uma versão satisfatória, quando poderá ser convertido em produto final. De acordo com Davis (1995) apud Huang (1998), protótipos evolucionários devem ser construídos quando se tem total conhecimento sobre aspectos críticos do produto, porém pouco conhecimento com determinadas funcionalidades.

2.2.2 Etapas da prototipagem

Coleta e refinamento dos requisitos: Nesta etapa é feita a coleta dos requisitos através de entrevistas com o cliente e pesquisas sobre a tecnologia. Após a coleta dos requisitos, eles são analisados e refinados até atingirem uma condição satisfatória.

Projeto rápido: Após a coleta e refinamento dos requisitos passa-se a etapa do projeto rápido, que consiste em esboços rápidos da interface e aspectos do produto final. Normalmente isto é feito através de esboços simples feitos em papel, ou programas que permitam desenho rápido. Nesta etapa pode-se utilizar o conceito de storyboards[8] ou storylines[9] para definir rapidamente como sistema irá evoluir.

Construção do protótipo: Com o projeto feito, inicia-se a etapa da construção do protótipo. Geralmente este protótipo é construído rapidamente, visando aspectos gerais do projeto rápido e dos requisitos coletados.

Avaliação do protótipo pelo cliente: Com um protótipo em mãos o cliente faz a avaliação, visando determinar quais aspectos podem ser melhorados e quais não serão mais necessários.

Refinamento do protótipo: Com esta avaliação em mãos, dependendo do tipo de prototipagem abordada, é feito ou um refinamento do protótipo construído, com o objetivo de consertar os aspectos tidos como falhos na avaliação e acrescentar novas funcionalidades visando outros requisitos não vistos, ou é construído um novo protótipo com estes mesmos objetivos em mente.

Engenharia do produto: Depois de vários ciclos de projeto-protótipo-avaliação, eventualmente chega-se a um produto estável que atende todos os requisitos que o cliente desejava. O protótipo, então, é convertido na forma do produto final.

---

[8] Storyboard – uma seqüência de figuras, etc. descrevendo o plano de um filme, comercial, hipermídia, jogo, etc.
[9] Storylines - a narrativa ou enredo de uma novela, teatro ou jogo.

2.3 Etapas do projeto

Durante a pesquisa determinaram-se metas para permitir um melhor entendimento do assunto. Estas metas visavam atingir um estado de entendimento suficiente sobre este assunto para que fosse possível começar a produção de protótipos.

Seguindo o método de prototipagem era necessário conhecer o suficiente do assunto para interagir com o cliente e conseguir uma descrição rápida do projeto. Neste caso se tratava em conseguir um motor gráfico gratuito em que fosse possível implementar funções de animação e que fosse adaptável o suficiente para implementar conceitos de Inteligência Artificial. Além disto, era necessária a leitura de bibliografia específica, com informações menos teóricas, para assim descobrir quais perguntas fazer. E quais soluções outros desenvolvedores já propuseram.

Testes seriam feitos para assentar tal conhecimento. Neste momento rascunhos iniciais do modelo deveriam ser testados no papel, seguindo o exemplo de projeto rápido.

Finalmente seria iniciado o método de prototipagem propriamente dito, com protótipos rápidos e simples visando atacar diversas partes do desenvolvimento. Estes protótipos serviriam como meios para familiarizar-se com as ferramentas e técnicas a fim de determinar o melhor curso do projeto.

3. FUNDAMENTAÇÃO TEÓRICA

3.1 Sobre RPG

Como o agente autônomo será usado em um jogo tipo RPG, convém explicar do que se trata.

RPG é a sigla em inglês para Jogo de Interpretação de Papéis. Ele surgiu na década de 1970 como forma de um jogo puramente de interpretação, como um teatro, porém com regras em sem um script. Na adaptação para o computador ou videogame os RPGs (ou JIPs) tomam inicialmente uma cara mais restrita, sendo que o jogador controla um personagem, mas segue um script e conversando ou lutando com NPCs[10], ganhando experiência e aprimorando suas habilidades.

De modo mais abrangente existem jogos de RPG para o computador, e recentemente para console, que são baseados em comunidades, onde se retira o fator script e libertam um personagem controlado pelo jogador em mundo totalmente interativo, onde se pode interagir com até milhares de outras pessoas. Este nicho do mercado de jogos de RPG recebe o nome de MMORPG[11].

Neste aspecto o agente autônomo inteligente tomaria o papel dos NPCs, podendo ser tanto os monstros como os inimigos que o PC[12] lutará (ou PCs), propiciando um divertimento maior dependendo do nível de inteligência atribuído ao agente.

---

[10] Non-Player Character: Personagem não jogador, aquele controlado pelo computador ou mestre de jogo. Também conhecido no âmbito de jogos de computador como Agente Autônomo.
[11] Massively Multiplayer Online RPG: RPG Massivamente Multi-Usuário Online onde se pode interagir com milhares de outras pessoas em uma comunidade na Internet.
[12] Player Character: Personagem Jogador, aquele controlado pelo jogador.

3.2 Sobre motores tridimensionais

No desenvolvimento de um agente autônomo inteligente é necessário conhecer um pouco sobre o que é um motor gráfico. Isto porque, para que um agente autônomo seja propriamente visualizado é necessário que este seja animado em um motor gráfico. No caso desta pesquisa, o intuito é estudar as técnicas de animação para um motor gráfico tridimensional.

Um motor tridimensional é uma biblioteca de funções para gráficos 3d. Muitos motores tridimensionais estão disponíveis na Internet, alguns são gratuitos, outros custam de $100 até $300,000 para uso comercial. A maioria dos motores usa a linguagem C++ para a programação, poucos – normalmente para iniciantes – usam programação usando a linguagem Basic. Programar um jogo em C++ ou Basic garante grande flexibilidade, porém requer muito tempo investido e muitos problemas a serem tratados antes que qualquer coisa se mova na primeira fase de seu jogo.

Um motor gráfico utiliza-se de um Renderer[13] que se encarrega de desenhar os objetos tridimensionais na tela, usufruindo-se das funções de bibliotecas adequadas para placas específicas. Software, que usa funções genéricas, porém mais lentas; DirectX, que é melhor nas placas de vídeo mais atuais, mas perde em placas mais antigas para o OpenGL.

Além disso, o motor gráfico pode ser aliado a um motor de física, que calcula o movimento, a rotação, etc. de todos os objetos de acordo com um conceito de física como forças (gravidade, torque, forças centrífugas, deformações, etc.). Apesar de não fazer parte propriamente dita do motor gráfico, um bom motor de física, garante um toque a mais de realidade ao jogo.

---

[13] Renderer: Renderizador desenha os objetos tridimensionais na tela.

Também existe o sistema de partículas que provê o controle e tratamento de efeitos especiais no motor gráfico, como explosões e jatos de água.

Para aumentar a eficiência do motor gráfico usam-se sistemas de armazenamento e busca inteligentes e avançados como árvores BSP (usado em Quake) e algoritmo dos Portais (usado em Unreal Tournament), que aumentam a velocidade de amostragem diminuindo o número de objetos a serem desenhados. Outra maneira de aumentar a eficiência é usando um sistema LOD [14] que automaticamente muda o nível de detalhe de um objeto dependendo da distância que ele está do observador. Isto causa objetos serem simplificados se estiverem longe e acelera a taxa de amostragem.

Além disso, é responsabilidade de um motor gráfico implementar uma hierarquia de objetos. Isto é de suma importância para a animação de agentes, pois com essa hierarquia é possível criar um "esqueleto" virtual do agente facilitando sua animação. O processo de animação de tais esqueletos é dita "kinematics" [15] (ALIAS|WAVEFRONT), ou "skeletal animation".

3.3 Sobre inteligência artificial

Como visto em Niederberger (2002) a inteligência artificial pode ser definida como a área da ciência da computação voltada a atribuir um comportamento inteligente a máquinas, seja este comportamento humano ou não. E desta forma pode ser usada em diversas áreas do conhecimento, desde economia até jogos. É possível classificar a Inteligência Artificial neste aspecto em quatro grandes abordagens:

---

[14] Level of Detail: Nível de Detalhe, usado para automaticamente reduzir a complexidade do objeto a ser desenhado dependendo quão longe ele está do observador, melhorando a velocidade de amostragem.

[15] Kinematics – Mistura do termo kinectic, que quer dizer físico e cinematics, que quer dizer movimento. Este assunto será abordado mais tarde.

- **Agir Humanamente**: Esta abordagem visa dar uma característica humana à máquina. E para testar isso se usa o teste de Turing, onde um participante humano tenta identificar somente através de conversas em bate-papos quais dos outros participantes é uma máquina.
- **Pensar Humanamente**: Pode ser coberto pelo campo da ciência cognitiva, associando modelos da IA com técnicas experimentais do campo da psicologia. Nesta abordagem o objetivo é imitar o mais completamente possível o processo cognitivo humano, sendo a solução deixada em segundo plano, comparada ao processo que levou a ela.
- **Pensar Racionalmente**: Usar ferramentas lógicas para resolver problemas. No entanto esta abordagem requer um modelo acurado da realidade, o que a torna, por muitas vezes, impraticável. E mais, fazer funcionar algo em teoria é diferente de fazê-lo na prática.
- **Agir Racionalmente**: Agir no intuito de alcançar um objetivo próprio. Que leva a criação de agentes racionais, que agem de forma a inferir sobre que maneira deverá agir dada uma circunstância, além de tomar decisões baseadas em dados incompletos, ou não baseadas em inferências (ex. Reflexos).

Em todas estas abordagens várias áreas da inteligência artificial são estudadas. E mesmo essas áreas não são mutuamente exclusivas, elas se sobrepõem e tentam atacar um problema de diversos ângulos. No entanto, os jogos modernos têm características próprias como visto em Nareyek (2002):

- **Tempo Real** – pouco tempo de processamento disponível. Raciocínios complexos requerem maior processamento, porém garantem respostas mais acuradas.

- **Dinamismo** – Jogos provêm um ambiente altamente dinâmico. Isto significa que os personagens devem ser capazes de lidar com diversos tipos de situações, e muitas vezes situações não previsíveis.
- **Conhecimento Incompleto** – Um personagem de jogo não tem conhecimento completo do mundo o qual está inserido. É necessário computar quão longe e o que um determinado personagem consegue enxergar ou ouvir. Além de saber navegar em qualquer ambiente, de forma correta. Ou seja, no caso de um jogo de tiro, em primeira pessoa, não se espera que um personagem atravesse paredes ou que fique preso em um determinado local.
- **Recursos** – O mundo do personagem ou seus recursos podem ser restritos.

Nareyek(2002) explica claramente a importância dessas técnicas e IA nos jogos e como se é vista a personificação desta nestes jogos quando diz:

> "Técnicas de IA podem ser aplicadas em uma variedade de tarefas nos jogos modernos de computador. Um jogo que usa redes probabilísticas para predizer o movimento que o jogador irá fazer, e deste modo aumentar a taxa de amostragem de gráficos, pode estar em um nível alto de IA. Mas apesar da IA nem sempre ser personificada, a noção de inteligência artificial em jogos de computadores é principalmente relacionada a personagens. Estes personagens podem ser vistos como *agentes*, suas propriedades se encaixando perfeitamente no conceito de agente de inteligência artificial".(NAREYEK, 2002).

3.4 Sobre agentes

Mas o que é um agente? A noção de agente na literatura tem várias interpretações, alguns autores até dizem não ser possível dar uma definição que acate a todas as demandas.

Niederberger(2002) define um agente como sendo um sistema, seja físico o em software, com as seguintes propriedades:

- **Autonomia**: agentes devem operar sem intervenção direta de humanos ou outros, e ter algum controle sobre suas ações e estados internos.
- **Habilidade Social**: agentes devem poder se comunicar com outros agentes ou humanos através de algum tipo de linguagem de comunicação de agentes.
- **Reatividade**: um agente deve perceber o ambiente a sua volta e responder de forma adequada e em tempo hábil.
- **Pró-atividade**: um agente não deve ser somente reativo, deve ser tomar iniciativa para que possa atingir seu objetivo.

Se restringirmos a visão do que é agente para os vermos como um softbot, ou agentes de software, Etzioni tem uma definição mais apropriada:

> "Um *softbot* é um agente que interage com um ambiente em software através de comandos sensores e interpretando a resposta do ambiente. Os atuadores dos softbots são comandos que mudam o estado do ambiente exterior. Os sensores dos softbots são comandos que fornecem informações". (ETZIONI, 1994 apud NIEDERBERGER, 2002).

Em suma um agente inteligente pode ser visto como uma entidade autônoma que interage com o ambiente, com outros agentes ou com humanos através de uma linguagem própria, ou reagindo a estímulos e tomando a iniciativa com pró-ações que o favorecem. E além destas propriedades é, ou conceituado ou implementado usando conceitos mais aplicados a humanos.

3.4.1 Agentes autônomos

Segundo Niederberger (2002), e respaldado na tese de Nareyek (2002), um agente ideal é aquele que sempre toma a ação que é esperada para maximizar sua medida de performance, dada as percepções, até o momento, vistas. Um agente é autônomo até o ponto que suas escolhas de ações dependem de sua própria

experiência, em vez de se basear em conhecimentos ditados pelo desenvolvedor (RUSSEL, Stuart J. e NORVIG, Peter, 1996).

Um programa agente consiste principalmente da execução repetida de tarefas como percepção, inferência, seleção e ação.

A tarefa de percepção coleta informações sobre o ambiente, seu estado desde a última ação. Com essas informações e baseado no estado do sistema após a última ação o agente consegue inferir o que tem de ser feito a seguir. Com isso ele terá um número de possíveis ações o qual deverá selecionar uma e então executá-la.

Nós podemos então dividir esses agentes autônomos por ordem de complexidade.

Tipos de agentes:

- **Agente Reativo ou baseado em ação-reação.**

  Estes agentes possuem um modelo bem simples no qual percebem o ambiente, e baseado em uma condição pré-imposta executam uma ação. São muito rápidos e se o sistema não for complexo, tende a ser uma boa solução. No entanto sua aplicação em jogos é muito restrita, e mesmo se introduzirmos o conceito de máquina de estados, como Niederberger sugere, ainda sim sua aplicação seria pequena. Pois em ambos os casos o desenvolvedor deverá prever todas as possibilidades de reações.

  Ex: se (motor_falhando) então {carro.pare}.

- **Agente com base em objetivos ou agentes deliberativos**.

  Quando introduzimos o conceito de busca de objetivos significa que o agente não mais responde somente a percepções do ambiente externo. Ele as analisa tentando avaliar a melhor solução que o leve a atingir seu objetivo. Nareyek diz isso claramente.

> "(...)Os objetivos e um modelo de mundo contendo informações sobre os requisitos da aplicação e as conseqüências de ações são representadas explicitamente. Um sistema interno de planejamento baseado em refinamento usa a informação do modelo de mundo para criar um plano que alcança o objetivo do agente."
> "O problema básico do planejamento é dado pela descrição inicial do mundo, uma descrição parcial do objetivo do mundo, e um conjunto de ações/operadores que mapeiam uma descrição parcial de mundo para uma outra descrição parcial de mundo. A solução é a seqüência de ações que levam da descrição inicial do mundo para a descrição do objetivo do mundo e é chamado de plano. O problema pode ser enriquecido incluindo-se mais aspectos, como problemas temporais e incertezas, ou requerendo a otimização de certa propriedade. (...)."
> "O problema com agentes deliberativos é sua lentidão. Todas as vezes que a situação é diferente da que foi antecipada pelo processo de planejamento, o plano deve ser recomputado. Computar planos pode ser muito demorado, e considerando os requisitos de tempo-real em um ambiente complexo se torna fora de questão" (NAREYEK, 2002)

- **Agente baseado em otimização (Utility-based Agents).**

Agentes baseados em otimização vão além dos agentes baseados em objetivos, quando analisam quais ações devem seguir baseados em quão satisfeitos ficarão com o resultado.

Um exemplo para este tipo de agente pode ser este: Ao escolher atravessar o rio o agente se depara com as seguintes soluções: nadar, pegar um barco ou cruzar a ponte. Ele está sem dinheiro para pagar o pedágio da ponte, então ele descarta essa solução, e quaisquer das duas outras soluções o fariam atravessar o rio, mas ele escolhe pegar o barco, pois não ficará molhado.

- **Agentes Híbridos**

Estes agentes usam de outros tipos de arquitetura de agentes para formar uma arquitetura híbrida que se espera ser capaz de resolver certos problemas. Um exemplo seria o uso de uma arquitetura que se assemelhasse a um agente baseado em objetivo, porém quando não fosse possível recomputar seu plano, assumia uma resposta reativa. Ainda sim, não é rápido o suficiente para os requisitos dos jogos modernos, podendo criar planos para coisas que já aconteceram.

- **Agente Qualquer-Hora (Anytime Agent)**

Este tipo de agente é chamado assim, pois computa seus planos somente quando possível e somente por quanto tempo for permitido. Deste modo ele sempre terá um plano, mesmo quando tiver pouco tempo. Eles trabalham dinamicamente, aperfeiçoando seus planos à medida que tem tempo disponível, porém ao final do tempo respondem com uma ação baseada no plano que computaram até o momento, mesmo que seja imperfeito. Assim consegue-se uma continuidade entre reação e planejamento. Nareyek diz.

> "Para horizontes de curto tempo de computação, somente planos primitivos (reações) estarão disponíveis, tempos mais longos de computação serão usados para melhorar e otimizar o plano do agente. Quanto mais tempo estiver disponível para as computações do agente, mais inteligente será o comportamento resultante. Adiante, a melhoria iterativa habilita o processo de planejamento a facilmente se adaptar a situações inesperadas ou que mudaram. Esta classe de agentes é muito importante para aplicações de jogos de computador." NAREYEK, 2002.

Com isso podemos dizer que os agentes anytime são os mais adaptados para os ambientes dos jogos que exijam grande verossimilhança com a realidade. Niederberger (2002) comenta que estes ambientes são acessíveis, não-determinístico, e não episódico. Comenta também que para jogo de um jogador, o ambiente pode ser estático, invariável, porém para jogos distribuídos de múltiplos jogadores, pode ser altamente dinâmico, dependendo da implementação. Jogos podem ser ainda, discretos ou contínuos; enquanto jogos do tipo "pula-e-corre" são discretos, um simulador de vôo ou um MMORPG são contínuos.

No entanto antigamente não se havia o poder de processamento requerido para comportamentos mais complexos, então normalmente usava-se de artifícios como aplicar regras diferenciadas aos NPCs. Tornando o jogo um pouco mais difícil, pois o jogador jogava em desigualdade. Porém apesar de ainda usar tais artimanhas,

hoje em dia mais e mais tempo de processamento está sendo dedicado somente às tarefas de computação para IA.

3.5 Sobre a animação de agentes

Vimos informações sobre motores gráficos, vimos técnicas de IA, vimos os tipos de modelos de agentes. Como isso é usado na animação desses agentes?

Os modelos de agentes nos proporcionam somente quais ações um determinado agente pretende fazer. Geralmente essas ações são dadas em alto nível, como “ande ao esmo”, ou “ataque tal lugar”, ou ainda “ataque tal aparte do corpo”. Como isso se traduz em um movimento visível na tela?

Em jogos comerciais como Half-life (Valve), Quake 2 (Id) e Quake 3(Id), são distribuídos SDKs que permitem utilizar uma linguagem de script para planejar o comportamento dos agentes. Esses SDKs entendem comandos específicos como “mova-se 200 unidades de medida” e traduzem em movimentos de objetos 3d que compõem a representação virtual do agente de forma a parecer um caminhar. Deste modo fica transparente ao programador a maneira como é feita a animação.

No entanto é interessante para nós vermos como é exatamente feita esta animação. Estas são as principais técnicas:

Captura de Movimento[16]: Usam-se modelos reais vivos usando um conjunto de sensores que visam captar o movimento desejado. Seja ele o bater das asas de um pássaro ou a maneira como o Ronaldinho chuta a bola. Esta maneira provê o melhor resultado em termos de suavidade e verossimilhança com o movimento

[16] Captura de Movimento – Mais conhecido por Motion Capture.

desejado. Depois de capturados os movimentos são categorizados e escolhidos de forma otimizada para executar uma tarefa.

Movimentos gerados por procedimentos: Estes modelos compõem uma classe de técnicas que visam gerar animações através de procedimentos. Normalmente usam-se bibliotecas de animação como animações pré-gravadas que são selecionadas de acordo com a situação devido ao pouco processamento na animação. Vosinaskis (2001) utiliza uma biblioteca de animação e de técnicas de geração de movimento por procedimento para implementar seu SimHuman, uma plataforma para agente virtuais animados em tempo-real com capacidades de planejamento.

Uma destas técnicas é a kinematics, que utiliza um esqueleto hierárquico virtual. Para movimentar um braço nesta técnica é passando uma mensagem para a junta do ombro, por exemplo, dizendo que gire 30º no eixo y. Esta mensagem é então interpretada e o braço gira de acordo com esses parâmetros. Este modelo pode ser melhorado deformando os polígonos que compõem o membro quando este é movimentado, desta forma não se criaria um “buraco” na animação como visto ao lado.

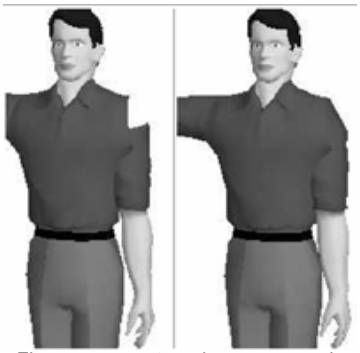


Figura que mostra o buraco causado pela falta de deformação.

Esta técnica tem lá seus defeitos, por um lado implementar movimentos simples se torna mais fácil. Por outro animar orientando-se a objetivos se torna uma tarefa custosa, pois é difícil prever a posição das extremidades.

Uma segunda técnica é a uma tentativa de evitar o problema acima mencionado. Chama-se Kinematics Invertida ou Restrição de Espaço-tempo. Nesta

abordagem o começo e o fim do movimento já estão pré-definidos, e a técnica tem que calcular uma trajetória do ponto inicial ao ponto final sem que haja uma colisão.

Uma terceira técnica é por movimento baseados em física. Esta técnica, diferentemente das outras duas, utiliza conceitos de torque e forças para modelar a deformação causada pelo movimento. Por um lado modelos complexos podem representar muito fielmente a simulação do movimento, por outro exigem muito tempo de processamento. Esta técnica é mais bem usada em simulações precisas de acidentes de trânsito, por exemplo. Os quais exigem uma precisão que os modelos kinamáticos podem não prover.

Isso porque os modelos kinemáticos são baseados em uma estrutura virtual que pode levar a movimentos não naturais. Por exemplo, este modelo aceitaria ordenar que o ombro girasse 50º no eixo y, 120º no eixo x e 270º no eixo z. Nos modelos baseados em física, isso não aconteceria, pois seria calculado que o torque causado pela rotação absurda em um eixo quebraria o braço, ou poria os polígonos que formam o contorno do braço e uma posição muito extrema além do possível.

Isso torna extremamente difícil usar estas simulações para produzir movimentos realistas, que obedeçam as leis da física.

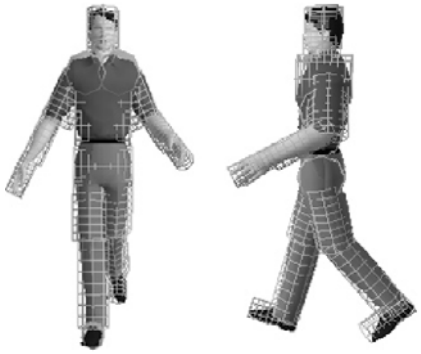
Controle de Movimento: Esta técnica baseia-se na divisão do movimento objetivo em sub-movimentos, e estes sub-movimentos separam-se em outros sub-movimentos para gerar, ao final, o movimento necessário.

Ações dinâmicas são ações executadas por um agente virtual que não são predefinidas (como em uma seqüência de a animação), mas adaptadas para um ambiente sempre mutável. (Vosinaskis, 2001). Com isso é possível criar movimentos dinâmicos baseados em uma composição de outros movimentos ou uma

sobreposição de movimentos parecidos, como dividir andar em andar rápido e andar devagar, sobrepondo outro movimento.

Niederberger (2002) comenta vários estudos usando está técnica desde a animação de peixes (que utilizam somente dois parâmetros para gerar movimentos, velocidade e ângulo), até simulação de humanos correndo.

Para simplificar a detecção de colisão em modelos complexos utilizam-se bounding poligons[17], que são polígonos simplificados para facilitar o processamento da colisão e ainda manter certo aspecto visual agradável.



Modelo humano com polígonos de borda.

3.5.1 Modelos tridimensionais

Normalmente em jogos a tarefa de animação dos modelos é feita pelos designers, sendo que as animações são feitas e armazenadas nos próprios arquivos de modelo ou de outra forma.

Sendo assim pesquisou-se os diferentes tipos de formato de arquivo para modelos tridimensionais.

MD2

Arquivo criado para o jogo Quake 2 e produzido pela Id Software. Este formato consiste em uma representação 3d de um agente usando informação sobre seus vértices. Assim ele é restrito as animações incluídas nele.

Modelo humano carregado do formato MD2.

[17] Bounding poligons – Polígonos de borda.

Estas animações representam diversos estados de um agente. No Quake 2 são: Ocioso, correr, atacar, três tipos de animação de dor, pular, cambalhota, saudar, cair de costas, acenar, apontar, ocioso agachado, andar agachado, atacar agachado, dor agachado, morrer agachado, morrer caindo para trás, morrer caindo para frente e uma animação de dano por explosão.

Devido ao sucesso do jogo Quake 2 e do lançamento de plataformas SDK para fãs produzirem suas próprias modificações, este formato é amplamente divulgado na internet sendo fácil a obtenção de alguns exemplos gratuitos. Sendo assim muitos motores gráficos o incluem como modelo suportável.

MD3

Um modelo mais avançado é o MD3 que armazena a representação 3d em blocos: Cabeça, tronco e pernas. Com cada animação separada o modelo poderia rodar animações independentes em seus blocos. Poder-se-ia correr e atacar ou torcer o torso e cabeça para visualizações.

Neste modelo as animações armazenadas são: Três animações blocos pernas e troco do modelo morrendo; Três animações para os dois blocos deles mortos; Uma animação do torso fazendo um gesto; Duas animações do torso atacando; Uma animação do torso largando algo; Uma animação do torso levantando; Duas animações do torso ocioso; Uma animação das pernas andando agachado; Uma animação das pernas andando; Uma animação das pernas correndo; Uma animação das pernas andando para trás; Uma animação das pernas nadando; Duas animações das pernas pulando; Duas animações das pernas aterrissando; Uma animação das pernas ociosas; Uma animação das pernas ociosas abaixado e uma animação das pernas virando.

O formato de arquivo MD3 foi usado nas primeiras versões do Jogo Quake 3: Arena, Id Software. E claramente oferece mais detalhes na animação para implementar diversos estados.

Este formato, por ser mais detalhado, é um pouco mais difícil de achar exemplos que seu predecessor. Sendo um arquivo mais recente não é tão divulgado no que se diz a suporte em motores gráficos.

MS3D

Esse formato, além das informações dos outros formatos, vértices, triângulos e pontos de textura; possui informações sobre juntas.

Essas juntas dão liberdade para o motor gráfico alterar o estado das animações desses modelos, controlando os membros de um agente humanóide, por exemplo.

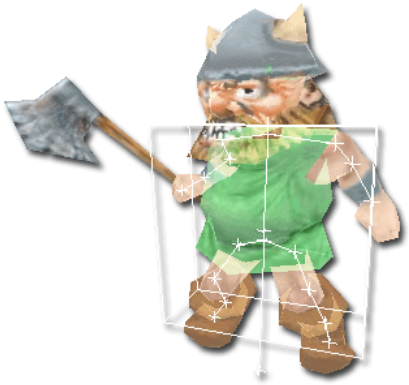
Alguns motores gráficos suportam tal formato.

Obj e Max

Esses dois formatos são formatos para modelos 3d estáticos. Eles não possuem animação inclusa. Sendo assim não são tão interessantes para este projeto.

Formato de Arquivo Direct-X

Formato proprietário da Microsoft. Este formato contém uma informação sobre o esqueleto virtual do modelo, além de um número não determinado de possíveis animações. Ele permite



Modelo de um anão guerreiro carregado do formato de arquivo do DirectX.

que o motor gráfico torne-o fácil de manejar, porém é difícil encontrar exemplos gratuitos.

3.6 Sobre a inteligência artificial usada em jogos

Existem diversas técnicas usadas para conferir certo grau de inteligência em agentes de jogos. E segundo Paul Tozur (2002a), da Ion Storm Austin, a inteligência artificial é mais bem usada em parceria com bons projetos de jogos (game design), uma simbiose entre os dois por assim dizer. Sendo assim não se trata somente de que técnica usar, mas sim como usa-la de acordo com o projeto do jogo. Algumas técnicas de inteligência artificial podem ser muito boas em jogos de estratégia baseados em turnos, mas a mesma técnica poderia ser inviável em jogos em tempo real.

Veremos algumas técnicas usadas em jogos que podem ser pertinentes no desenvolvimento de agentes para jogos de RPG.

3.6.1 Navegação

Para um agente ser autônomo é necessário que possa se locomover pelo mundo virtual. O Agente também deve obedecer aos limites do mundo, respeitando obstáculos e achando seu caminho. Ele pode mover-se a esmo identificando obstáculos e desviando deles de maneira incidental, podem percorrem pontos determinados ou podem seguir um caminho calculado, esta técnica chama-se busca de caminhos ou *pathfinding*.

Dan Higgins (2002) da Stainless Steel Studios, Inc conta que *Pathfinding* pode ser usado para diferentes ações além de somente achar um caminho para movimentar um agente. Com ele é possível descobrir se um ponto é acessível, e

assim poder escolher algum outro meio de acesso ou como arranjar defesas em um jogo de estratégia.

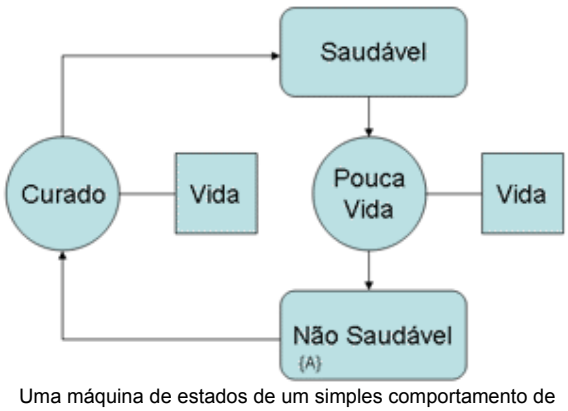
Uma maneira de usar a técnica *pathfinding* é através de uma máquina A*. A* é um algoritmo para achar a melhor solução de um problema, percorrendo nós. Neste caso, os nós são unidades adjacentes em um mapa e o algoritmo tenta alcançar um ponto neste mapa vasculhando os nós.

No caso de uso de pontos determinados no mapa, também chamados de *waypoints*, a técnica é usada desta maneira. Em um mapa pontos são escolhidos como *waypoints*, e o agente navega através de caminhos pré-calculados entre esses pontos. Esses pontos podem ainda conter informações pertinentes a quais outros pontos ele pode enxergar dentro do mapa, salvando assim o cálculo de campo de visão. Como Paul Tozour (2002b) conta em outro artigo, o algoritmo A* pode ser usado neste modelo também, considerando os *waypoints* como nós.

3.6.2 Máquina de estados finitos

Uma das técnicas usadas na modelagem de agentes é o uso de uma maquina de estados finitos, ou FSM (*Finite State Machine*) em inglês. Essa técnica pode ser usada para modelar tanto simples comportamentos, quanto os mais complexos, sendo que uma implementação genérica de uma máquina de estados finitos com uma linguagem para definir os estados, ações, condições e transições é essencial para re-usabilidade do código.

Ao lado vemos um exemplo de uma máquina de estados proposto por



Uma máquina de estados de um simples comportamento de uma criatura.

proposto Phil Carlisle(2002) usada para descrever um modelo simples de comportamento de um agente. Ela possui dois estados (Saudável e Não Saudável), quando a variável “Vida” cai abaixo de um determinado valor, isso causa a condição “Pouca Vida” a ser atingida, fazendo com o que o agente mude de estado para Não Saudável. Neste estado existe uma ação, determinada pelo símbolo {A}, que pode ser buscar comida ou algo assim. A ingestão de comida pode fazer com o que a variável vida se recupere, assim a condição “Curado” é acionada e ele volta ao estado de “Saudável”. Como.

3.6.3 Aprendizado

Em respeito a aprendizado existem várias técnicas usadas em jogos. Uma delas é o aprendizado por monitoração, ou GOCap (Game Observation Capture). Nesta técnica um ser humano controla o agente e este observa o comportamento, identificando padrões e armazenando comportamentos. Depois com o agente propriamente treinado ele pode ser usado.

Outra técnica é a do algoritmo genético. Em que dado um conjunto de regras e valores aleatórios, e através de várias gerações, um agente com o comportamento adequado evolui. Assim em vez de testar um agente e testando se ele se comporta adequadamente, usa-se a metáfora da seleção natural para escolher os agentes mais adaptados.

Outra técnica ainda é o uso de modelos estatísticos N-gram para prever o comportamento de jogadores. Esse N-gram constitui de um diagrama com N nodos e os nodos 1 a N-1 são eventos do passado. Assim analisando esses eventos pode-se prever estatisticamente o comportamento do jogador e reagir de acordo.

O reconhecimento de padrões também é usado como técnica de aprendizado, e embora funcione de maneira diferente da anterior, também é usada para prever possíveis respostas em uma determinada seqüência.

Redes neurais também podem ser usadas para dar a capacidade ao agente de aprender. Alex J. Champandard (2002), da Artificial Intelligence Depot, mostra dois exemplos de tal. Um com reconhecimento de padrões para assistir na tomada de decisão de componentes de IA de alto nível. E o outro é o uso das redes neurais para mapear controladores de robôs fisicamente, só que em mundos virtuais tridimensionais. Um agente aprenderia a controlar seus atuadores no mundo virtual.

4. DESENVOLVIMENTO DO PROJETO

Para implementar um agente era necessário descobrir que requisitos eram necessários. A primeira tarefa foi buscar um meio de animar um modelo tridimensional. Como este conhecimento era desconhecido pelo aluno era de suma importância pesquisá-lo. Sem isso o projeto não andaria.

A segunda tarefa foi pesquisar artigos recentes sobre este assunto. A idéia era obter o conhecimento necessário para formular as perguntas certas. Alem disso foi pesquisado sobre as técnicas usadas no desenvolvimento de jogos e agentes a fim de descobrir outros requisitos.

4.1 Requisitos

- Um motor gráfico que facilitasse a implementação, e modelos tridimensionais com animações;
- Uma arquitetura que permitisse a representação de um agente utilizando o motor gráfico escolhido;
- Um modelo que represente um comportamento para o agente;
- Um modelo que capacite o agente aprender com o jogador.

Além disso, a arquitetura do sistema deveria ser simples e adequar-se ao motor gráfico. Deveria descrever estruturas mínimas para a representação de agentes em um mundo virtual.

Com a arquitetura feita dever-se-ia criar um modelo de comportamento seguindo uma das técnicas estudadas. E finalmente integrar a capacidade do agente aprender, caso seu comportamento não fosse autônomo ou inteligente por si só.

4.2 Implementação

4.2.1 Motor gráfico

Foi pesquisado um número considerável de motores gráficos durante o decorrer do trabalho de conclusão de curso. Levou-se um tempo precioso procurando um que fosse relativamente estável, gratuito e que fosse simples de usar. Nenhum alcançou todas as expectativas, mas tinham características interessantes.

Relação dos Motores Gráficos pesquisados:

- **Genesis3D (Eclipse Entertainment):**
    - Vantagens:
        - Distribuído sob forma de SDK. Provê diversas ferramentas como ActView para visualizar atores, AStudio para construir um ator com objetos e animações e GEDIT para a construção de níveis;
        - Integra-se ao Visual C++ de maneira rápida;
        - Possui um exemplo de um jogo simples em primeira pessoa;
        - Gratuito.
    - Desvantagens:
        - Não possui atualização oficial recente;
        - Usa biblioteca Glide que é antiga;
        - Pouca documentação e os exemplos não possuem comentários detalhados, tornando-os confusos e massivos;
        - Comunidade pouco ativa.
- **The Nebula Device (RADON LABS):**
    - Vantagens:
        - Poderoso e testado em um jogo comercial (Project Nomad);

        - Totalmente de graça.
    - Desvantagens:
        - Sem suporte oficial;
        - Interface complicada e de difícil aprendizado;
        - Pouca documentação. Código-Fonte pouco comentado e em diferentes línguas.
- **The Nebula Device 2 (RADON LABS):**
    - Vantagens:
        - Também distribuído em forma de SDK;
        - Possui muitos avanços tecnológicos em comparação ao TND (The Nebula Device).
    - Desvantagens:
        - Pouca documentação e exemplos;
        - Segue o mesmo estilo de seu predecessor;
        - Pela falta de funções existentes no antigo TND exige muito tempo de programação.
- **IRRLicht:**
    - Vantagens:
        - Interface simplificada; Tornando o tratamento de funções e aspectos da criação de um jogo muito mais simples.
        - Integra-se ao Visual C++ de maneira rápida e simples;
        - Exemplos bem documentados;
        - Comunidade bem ativa.
    - Desvantagens:

        - Exemplos mostram pouco conteúdo, apesar de bem documentados.

- **Fly3D 1.2 (Paralelo Computação)**
    - Vantagens:
        - Distribuído sob forma de SDK; com ferramentas que geram código através de opções (wizards[18]);
        - Possui um exemplo pronto de Jogo.
    - Desvantagens:
        - Não é mais mantido pela empresa e por isso não possui muita documentação disponível;
        - A documentação existente é distribuída através de um livro;
        - Comunidade praticamente inexistente.
- **Fly3D 2.0 (Paralelo Computação)**
    - Vantagens:
        - Grandes melhorias em relação à versão anterior.
    - Desvantagens:
        - Versão completa exige compra de um livro;
        - Comunidade praticamente inexistente.

Dentre todos os motores gráficos vistos, dois se destacam como promissores para este projeto. Um é o Genesis3D, que apesar de uma arquitetura mais antiga, possui os elementos básicos que não são objeto deste trabalho, como tratamento de física, colisões, carregamento de cenário e modelos. No entanto o modelo que o Gênesis usa é um modelo proprietário, que pode ser obtido de um modelo MD2.

[18] Wizard – ferramente que guia o usuário através de opções e o ajudam a executar uma tarefa.

O outro é o IRRLicht, que foi lançado no começo do ano de 2003 e vem tendo progresso considerável. Já existem alguns projetos de jogos baseados neste motor gráfico.

Com a idéia de criar uma interface simplificada que resolvessem muitos dos problemas dos programadores, o IRRlicht mostrou um grande potencial. Existem poucos exemplos práticos de suas funções mais obscuras, mas o suficiente do básico para dar uma idéia do todo. Sua comunidade bem ativa pôde preencher a necessidade de exemplos.

Sendo assim acabou-se escolhendo o IRRLicht como Motor Gráfico.

4.2.2 Modelo de Animação

Era necessário encontrar modelos tridimensionais de agentes que poderiam ser usados neste projeto. Encontraram-se uma quantidade razoável de modelos tridimensionais no formato MD2, alguns poucos no formato MD3. Além de alguns poucos modelos no formato obj e max.

Como o IRRLIcht, na época em sua versão 0.4, suportava o MD2, mas não o MD3 (O suporte a esse formato só veio na versão 0.4.2 que era instável). Decidiu-se começar os primeiros testes com MD2, então.

Jogos mais novos, incluindo versões mais recentes de Quake 3, usam um modelo baseado em animação por “ossos com pesos”, ou *Bone Weighting Animation*. Basicamente se atribui pesos a movimentação de diferentes juntas usando os princípios de *kinematics*, ou animação por esqueleto hierárquico. Como visto anteriormente esta técnica torna a animação mais macia, pois possibilita o ajuste entre ações sem que o Agente pareça “robotizado”. Porém nesta época o IRRLicht

aparentemente não suportar essas funções, e não havia sido achado modelos tridimensionais que tivessem um formato que suportasse essas funções.

Usando o IRRLicht é possível usar o MD2 de modo consideravelmente simples. Sendo que o objeto do projeto é a implementação de conceitos de IA. Ter um modelo que rode animações simples é suficiente.

4.2.3 Estrutura de um Jogo

Para implementar um agente acabou-se descobrindo que era necessário ter certas estruturas e funções que permitissem a manipulação dos agentes de modo que eles interagissem com o mundo de forma simples, eficiente e lógica.

Em primeira instância tentou-se contornar a situação utilizando atalhos e simplificações. Mas viu-se que para dar continuidade ao projeto era necessária uma estrutura básica de um jogo. Consistida de controladores de níveis, hierarquia de entidades, e um sistema de passagem de mensagem.

Os exemplos do IRRLicht não eram elaborados o suficientes para facilitar o trabalho. E implementar essas funções não era escopo principal do projeto. Algum esforço foi empregado na concepção desta arquitetura, porém mais tarde descobriu-se um *framework*[19] para desenvolvimento de jogos tipo RPG e compatível com o IRRLicht. Paul Zirkle (2004) criou o IRRLichtRPG, e neste *framework* havia o necessário para começar a modelagem de sistemas para inteligência artificial. Esse *framework* ainda estava em processo de desenvolvimento. As estruturas básicas do jogo, como hierarquias de entidades, controle de nível, sistema de eventos e passagem de mensagens, estavam utilizáveis, mas não totalmente implementadas.

[19] Framework – uma descrição simplificada de uma entidade ou processo complexo.

Uma das funções deste *framework* que não havia sido implementada era um módulo básico de Inteligência Artificial. Assim provou-se essencial para a continuidade deste projeto.

Diagrama de classes

Para clarificar o uso deste *framework* é conveniente apresentar um diagrama de classes das estruturas mais importantes.

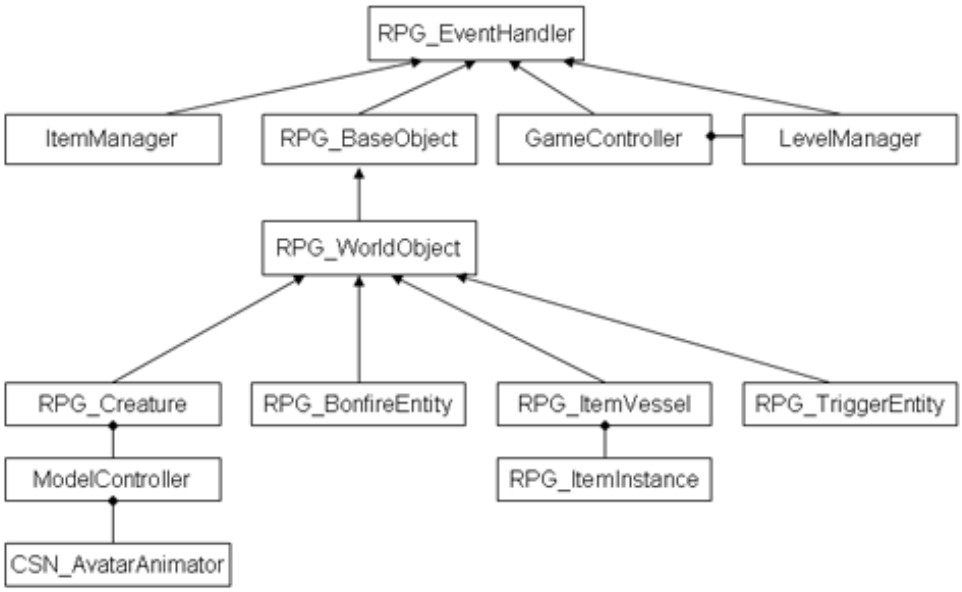


Diagrama de classe relevantes do IRRLichtRPG *framework.*

Neste diagrama de classe a classe que seria usada para basear o agente inteligente seria a classe RPG_Creature. Ela descreve uma entidade que se movimenta no mundo virtual por própria vontade. Essa mesma classe pode ser usada para descrever tanto NPCs e avatares de jogadores. Ela é uma especialização da classe RPG_WorldObject que descreve objetos estáticos sujeitos a forças externas.

A classe RPG_Creature é associada a classe ModelController que possui a função de representar a conexão lógica entre um nodo de cenário (sceneNode)

móvel e a animação específica para tanto. Para isso ele repassa a informação usando a classe CSN_AvatarAnimator, que simplesmente associa o estado da animação a animação correta. Ou seja, se a entidade criatura é ordenada a se mover, além da mudança de posição o ModelController deve associar a animação adequada através do CSN_AvatarAnimator. Esse modelo é muito parecido com o ActionTable proposto por Jeff Orkin(2002).

As classes GameController e LevelManager coordenam e monitoram aspectos específicos da lógica do jogo, além de se encarregar de repassar os eventos para as entidades a medida que ocorrem.

Captura de tela do IRRLichtRPG *framework.*

Na cena acima o modelo identificado como “A” representa uma instância da classe RPG_Creature controlada pelo jogador. O modelo identificado como “B” representa tambérm uma instância da classe RPG_creature, só que os eventos de

entrada de dados não são repassados para ela. Assim a instância “B” é exatamente igual à instância “A”, alterando somente o modelo tridimensional carregado.

A instância “C” é uma nova entidade, que ainda está em fase de teste no IRRLichtRPG. Representa uma instância da classe RPG_ItemVessel e como não foi utilizada neste projeto, foi removida da demonstração.

Controlador da Inteligência Artificial

Para modelar um sistema de inteligência artificial que controle uma entidade é possível criar uma classe AI_Controller que repassaria os comandos a entidade de maneira igual ou similar a maneira que o jogador controla uma entidade. Seria o cérebro da criatura usando uma analogia biológica.

O gerente de nível ou *level manager* deve passar os eventos que ocorrem ao AI_Controller. E este, utilizando uma técnica de inteligência artificial, determinar a ação da entidade associada a ele.

4.2.4 Inteligência Artificial

No aspecto de inteligência artificial foram buscadas maneiras de implementar os conceitos vistos anteriormente.

Para começar os protótipos neste aspecto fio escolhido uma máquina de estados finitos para o comportamento de um agente baseado em ação-reação. Seria composto de estados simples como “lutar”, “vagar”, “fugir”. Cada estado seria, então, aperfeiçoado. Mais ou menos como proposto por Phil Carlisle(2002).

O modelo possui cinco estados (“ocioso”, “vagar”, “fugir”, “lutar”, “curar”), quatro condições (“inimigo avistado”, “fora de alcance”, ”curado” e “pouca vida”) monitorando duas variáveis (“vida” e “localização do inimigo”).

A figura a seguir mostra as transições de cada e estado e em que condições elas são efetuadas.

Além disso, cada estado tem ações associadas a eles. O estado ocioso não possui ações especificas. Neste estado o agente apenas permanece parado esperando determinados eventos ocorrerem.

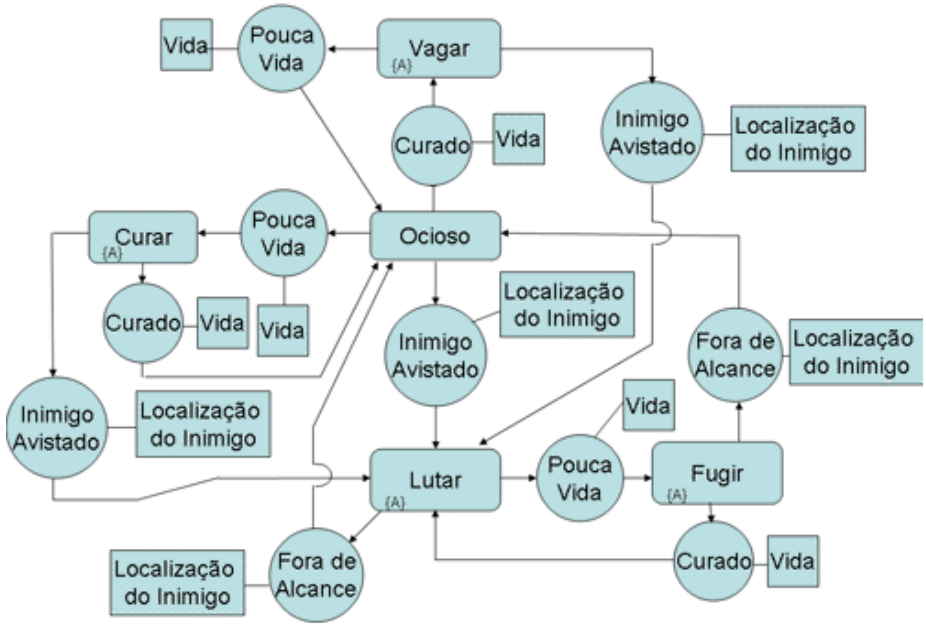


Diagrama de estados do modelo proposto de comportamento do agente.

Quando a condição de inimigo avistado é satisfeita o agente vai para o estado de lutar. Essa transição é uma maneira preventiva. Pois se no próximo ciclo a condição de pouca vida for satisfeita ele tenta fugir, mas pelo menos no primeiro ciclo, se houver oportunidade ele efetua pelo menos um ataque.

No estado de fugir ele vai tentar manter o inimigo fora de sua linha de visão, fugindo de costas para longe. Quando não conseguir mais enxergar o inimigo ele resume ao estado ocioso. Caso se cure durante a fuga ele volta ao estado lutar.

Se lutando o inimigo desaparece, caso morra ou fuja, ele volta ao estado ocioso.

No estado ocioso, caso o inimigo não for avistado e o agente estiver curado, ele passa o estado vagar. Neste estado ele percorre o mundo até encontrar um inimigo, ou ficar com pouca vida. Se ficar com pouca vida ele resume ao estado ocioso, se encontrar um inimigo ele muda para o estado lutar.

Se ele estiver no estado ocioso e tiver pouca vida, ele passa a se curar quando curado volta ao estado ocioso. Mas se no processo de cura ele avista um inimigo ele passa ao estado Lutar.

Para implementar essa função foi criado um método "AI_Controller::think(irr::u32 enemySpoted, irr::core::vector3df colision)" que receberia o número de identificação do inimigo, e o vetor de colisão. A intenção era averiguar se um inimigo havia sido visto, e se estava longe ou perto.

Com o controlador monitorando a saúde ou vida da criatura a qual estava relacionada seria possível identificar se as condições de transição eram satisfeitas ou não.

A cada ciclo de jogo (*game cycle*) o método executava uma ação, ou passo de ação, no caso da ação ser perseguir ou fugir. Ou mudaria o estado da máquina de estados. Estes estados são definidos como uma estrutura enumerada chamada tAI_STATE com os seguintes valores: AIS_IDLE, AIS_WANDER, AIS_FIGHT, AIS_FLEE, AIS_HEAL.

Em caráter temporário o AI_Controller teria alguns outros aspectos que mais tarde seriam implementados em classes diferentes. Os atributos, m_goodHP e m_lowHP, eram atributos que definiam o como considerar um agente curado ou com pouca vida. AI_BUSY determinava se o controlador estava no meio de uma tarefa. Como por exemplo, movendo em direção a um ponto especifico. m_enemyId determinava um inimigo específico que o controlador deveria monitorar. m_visionArc

determinava o arco de visão em graus que o agente tinha, Sendo possível determinar se inimigo podia ser visto. m_curState apenas dizia o estado atual da máquina de estados. E o atributo m_body era um ponteiro para a entidade que o controlador deveria controlar.

4.2.5 Ferramentas utilizadas

Durante o processo foram utilizadas diferentes ferramentas. Eis um detalhamento delas:

- IRRLicht – Para os primeiros testes foi utilizado a versão 0.4 da biblioteca do motor gráfico, mas a medida que novas versões mais estáveis foram lançadas foi-se adaptando na medida do possível. Atualmente está se usando a versão 0.6, com melhorias consideráveis sem prejudicar o código já implementado.
- IRRLichtRPG – Criado por Paul Zikle(2004), este *framework* está sendo distribuído em sua versão 0.2.1f-stable. É compatível com a versão 0.4.2 e superiores do IRRLicht.
- Visual Studio C++ 6.0 – Ambiente de programação para linguagem C++ da Microsoft.

4.3 Resultados

Analisemos os objetivos e verifiquemos os resultados.

O primeiro objetivo foi estudar o estado da arte no uso de inteligência artificial em jogos tridimensionais, além do processo de desenvolvimento de um agente inteligente.

Esse objetivo foi atingido, pois fora estudado várias tecnologias e artigos científicos sobre o assunto. O conhecimento adquirido do processo de

desenvolvimento de jogos, e mais especificamente de agentes inteligentes foi esclarecedor.

Vimos que o desenvolvimento de agentes para jogos depende muito do tipo de jogo e tecnologias disponíveis. A navegação em um jogo baseados em gráficos 2D seria um tanto diferente em um baseado em gráficos 3D, ou se é ou não online. Isso, além da função que este agente teria em um dado jogo, que também influenciaria no comportamento deste agente. NPCs que assumiriam um papel de mercador em um jogo de RPG teriam comportamentos diferentes que NPCs que assumiriam o papel de Inimigos ou até aliados.

Enquanto o primeiro NPC, o mercador, não necessitaria, a princípio, de uma solução complexa para sua navegação no mundo, os NPCs que fariam papeis de inimigos e aliados necessitariam. Todavia, o mercador utilizaria recursos de interação com o usuário os quais seriam diferentes dos recursos usados pelo NPC inimigo. Por exemplo, o mercador poderia tentar barganhar com um usuário e este processo poderia ser influenciado com o seu humor, ou reação para com o PC (Player Character). Uma coisa que um inimigo implementaria diferente, em vez de barganhar com ele, o NPC decidiria o curso de uma ação baseado na reação para com o PC. Poderia ser uma investida ou até recuar e chamar reforços.

Neste projeto escolheu-se implementar-se um agente inteligente que seria um adversário. Ele enfrentaria o PC, perseguindo ou fugindo quando necessário.

O segundo objetivo era averiguar as possibilidades de implementação estudando os requisitos fundamentais no desenvolvimento de um agente inteligente. Isto é, adquirir um motor gráfico que atenda as especificações, criar ou adquirir estruturas fundamentais que possibilitem o desenvolvimento propriamente dito do

agente inteligente, e determinar se é possível implementar tal agente dentro desses modelos.

Este objetivo foi complicado de perseguir. O conhecimento imaturo inicial do desenvolvimento prático de jogos dificultou a obtenção de tais tecnologias. Muito tempo foi gasto na procura e desenvolvimento desses requisitos. Desde a obtenção do e familiarização com o motor gráfico, até o desenvolvimento e posterior obtenção de um *framework* para criação de jogos no motor gráfico escolhido. Apesar de não implementar um agente inteligente, conseguiu-se adquirir conhecimento para começar tal processo. Assim o objetivo de determinar a possibilidade de tal projeto foi cumprido, sendo a idéia fazer protótipos cada vez mais complexos partindo de uma base simples.

Nesta abordagem começou-se testando como controlar o modelo 3d através de animações e logo viu-se que era necessário uma estrutura mais elaborada para continuar o projeto. Os protótipos partiram de simples classes para definir um agente e como controla-lo.

O *framework* que finalmente foi utilizado neste projeto, o IRRlichtRPG por Paul Zikle (2004), somente foi encontrado mais tarde, mas avançou o projeto ao ponto de possibilitar o próximo passo: A implementação de um comportamento.

O terceiro objetivo era modelar um comportamento para o agente usando técnicas apropriadas e dentro das limitações das ferramentas.

Esse objetivo foi atingido na modelagem de um sistema simples de inteligência artificial utilizando-se de uma máquina de estados finitos para descrever o comportamento de um agente. Esta máquina de estados finitos torna o agente um

agente reativo ou ação-reação, como visto nos trabalhos de Niederberger (2002) e Russel e Norvig (1996).

Este sistema foi visto como uma boa maneira de familiarizar-se com as ferramentas. A idéia era criar protótipos cada vez mais complexos. E ao final introduzir técnicas de aprendizagem ao agente.

O quarto e último objetivo era implementar um modelo de inteligência artificial na estrutura escolhida.

O objetivo não foi completamente atingido, porém os protótipos produzidos chegavam perto de implementar tal modelo. Testando características simples de passagem de mensagem e controle por meio de manipulação do sistema de eventos do *framework* de jogos tipo RPG.

O sistema de eventos do IRRlichtRPG não estava 100% pronto, mas estava utilizável. A mesma classe que definia um objeto tipo criatura era usada tanto para definir o PC quanto para definir um NPC. Assim o sistema criado deveria poder controlar o NPC usando a mesma interface que o Jogador usaria para controlar o PC.

A Máquina de estados foi, então, implementada nesta arquitetura e o sistema de eventos usado para passar as mensagens como explicado anteriormente.

5. CONCLUSÕES

O estudo do desenvolvimento de um jogo foi de suma importância a este projeto, visto que nas referências sobre inteligência artificial assumia-se que o desenvolvedor já obtivesse ferramentas que possibilitassem a manipulação de aspectos de jogos. Sendo a proposta criar um agente inteligente para jogos tridimensionais tipo RPG, foi essencial obter não somente um bom motor gráfico, porém desenvolver uma arquitetura simples que simule um jogo.

Felizmente obteve-se o IRRLichtRPG que possibilitou a continuidade do projeto, pois sem ele foi necessário desenvolver uma arquitetura antes de passar a desenvolver a inteligência artificial. O que atrasou consideravelmente o cronograma.

Viu-se então que ter um motor gráfico potente é apenas uma pequena, embora importante, parte do desenvolvimento de um jogo. Estudando a literatura e através prática averiguou-se que as técnicas de inteligência artificial para criar um agente de jogo estão intimamente ligadas com o design do jogo desenvolvido e que dar a percepção de inteligência ao um agente não significa que o agente seria modelado para imitar exatamente o comportamento de um jogador. O agente pode ter ajuda de um mundo bem estruturado e truques que dão a percepção de inteligência a um agente.

Trabalhos Futuros

Como trabalho futuro poder-se-ia continuar a implementação do modelo do controlador de IA, atentando para torná-lo flexível e incorporar outras técnicas de IA como um *pathfinding* elaborado em vez do modelo simplificado usado. Para tanto, seria necessário alterar parte da implementação do IRRLichtRPG. A nova versão do

IRRLichtRPG, ainda em desenvolvimento na data que este relatório foi escrito, terá seu sistema de eventos refeito, o que facilitará a tarefa.

Outra proposta é incorporar as funcionalidades de curvas de resposta como visto num artigo de Bob Alexander (2002), para gerar comportamentos mais parecidos com jogadores nos agentes. Introduzindo a capacidade de errar de forma normal para um jogador. Além da implementação das outras técnicas citadas neste projeto.

6. FONTES BIBLIOGRÁFICAS

ALEXANDER, Robert. **AI Programming Wisdom**: The Beauty of Response Curves. p. 78-82. Hingham, Massachusetts: Charles River Media, 2002.

ALIAS|WAVEFRONT. **The Art of Maya**. Character Animation. p. 100 - 114. ISBN-1894893131

CARLISLE, Phil. **AI Programming Wisdom**: Designing a GUI Tool to Aid in the Development of Finite-State Machines. p. 71-77. Hingham, Massachusetts: Charles River Media, 2002.

CERVO, Amando Luiz; BERVIAM, Pedro Alcino. **Metodologia Científica.** 4ºed. São Paulo: Makron Books, 1996. p.20-22,44-45,48-48.

CHAMPANDARD, Alex J. **AI Programming Wisdom**: The Dark Art of Neural Networks. p. 640-651. Hingham, Massachusetts: Charles River Media, 2002.

CHAMPANDARD, Alex J. **The Future of Game AI**: Intelligent Agents. Disponível em: <http://ai-depot.com/GameAI/Agent-Intelligence.html>. Acessado em: 24 jul. 2003.

CONTINUUM Entertainment. Disponível em: <http://www.continuum.com.br>. Acessado em: 27 abr. 2003.

DFC Intelligence. Sessão de Artigos da Indústria de Jogos. *Game Industry Market Forecasts.* Disponível em: <*http://www.dfcint.com/game_article/mar03article.html*>. Acessado em: 28 abr. 2003.

DFC Intelligence. Sessão de Artigos da Indústria de Jogos. *Game Industry Market Forecasts part 2.* Disponível em: <*http://www.dfcint.com/game_article/april03article.html*>. Acessado em: 28 abr. 2003.

ESPAÇO Informática Ltda. Disponível em: <http://www.hades2.com/espaco/>. Acessado em: 27 abr. 2003.

FALSHIN, Odília. **Fundamentos de Metodologia.** São Paulo: Atlas, 1993. p. 36-38,101-104.

GAMASUTRA. **Game AI:** The State of the Industry. Disponível em: <http://www.gamasutra.com/features/20001101/woodcock_01.htm>. Acessado em: 25 de jun. 2003.

ECLIPSE Entertainment. Genesis3D Engine. Disponível em: <http://www.genesis3d.com/>. Acessado em: 13 de Jan. de 2004.

GREENLAND Studios. Disponível em: <http://www.greenlandstudios.com>. Acessado em: 27 abr. 2003.

HIGGINS, Dan. **AI Programming Wisdom**: Generic A* Pathfinding. p. 114-121. Hingham, Massachusetts: Charles River Media, 2002.

HUANG, Yao-Chin. **Prototyping**: Throwaway or Evolutionary? 1998. Disponível em: <http://www.scis.nova.edu/~yaochin/660-p.htm>. Acessado em: 3 ago. 2003.

IGNIS Games. IGNIS Entretenimento e Informática S/C Ltda. Disponível em: <http://www.ignisgames.com.br>. Acessado em: 27 abr. 2003.

IGNIS Games. FAQ do Jogo Erynis. Disponível em: <http://www.ignisgames.com.br/erynis/index.php>. Acessado em: 27 abr. 2003.

IRRLicht Engine. Disponível em: <http://irrlicht.sourceforge.net> Acessado em: 20 de Dez. de 2003.

LARAMÉE, François Dominic. **AI Programming Wisdom**: Evolving the Perfect Troll. p. 629-639. Hingham, Massachusetts: Charles River Media, 2002.

LUCAS ARTS entertainment. Sessão de ofertas de trabalhos. Disponível em: <http://www.lucasarts.com/jobs>. Acessado em: 28 de Jun. de 2003.

NAREYEK, Alexander. **Constraint-Based Agents** - An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in

Open and Dynamic Worlds. 2001. 176p. Tese (PhD em Ciência da Computação) - Technical University of Berlin, Departamento de Ciência da Computação, Alemanha. Disponível em: <http://www.ai-center.com/references/nareyek-01-cba.html>. Acessado em: 27 jun. de 2003.

RADON LABS. The Nebula Device 1 e 2. Disponíveis em: <http://nebuladevice.sourceforge.net/>. Acessado em: 05 de Jan. de 2004.

NIEDERBERGER, Christoph. GROSS, H. Markus. **Towards a Game Agent**. Computer Science Dep. CS Technical Report #377. ETH, Zurique – Suíça. 26 de ago. de 2002.

PARALELO COMPUTAÇÃO. Fly3d Engine. Disponível em: <http://www.fly3d.com.br/>. Acessado em: 20 de Jan. de 2004.

RUSSEL, Stuart J. NORVIG, Peter. *Artificial Intelligence - A Modern Approach*. PrenticeHall, 1996. **ISBN -** 0137903952

SALOMON, Délcio Vieira. **Como fazer uma Monografia de trabalho científico.** Belo Horizonte: Interlivros, 1978. p.137-138,140-147.

TOZOUR, Paul. **AI Programming Wisdom**: The Evolution of Game AI. p. 3-15. Hingham, Massachusetts: Charles River Media, 2002.

TOZOUR, Paul. **AI Programming Wisdom**: Building a Near-Optimal Navigation Mesh. p. 171-185. Hingham, Massachusetts: Charles River Media, 2002.

VILIEGAS, Renato. A Reta Final. **Electronic Gaming Monthly Brasil**. São Paulo. n.13. p.49-53. 2003.

VOSINAKIS, Stypos. PANAYIOTOPOULOS, Themis. **SimHuman**: A Platform for Real-Time Virtual Agents with Planning Capabilities. Knowledge Engeneering Laboratory. Department of Informatics, University of Piraeus. Grécia, 2001.

ZAROZINSKI, Michael. **AI Programming Wisdom**: An Open-Source Fuzzy Logic Library. p. 90-101. Hingham, Massachusetts: Charles River Media, 2002.

ZIRKLE, Paul. **IRRLichtRPG.** Disponível em: <http://www.skyesurfer.net/keless/IrrLicht/RPG/>. Acessado em 22 de Abril de 2004.

7. ANEXOS

7.1 Anexo I – Artigo

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO DE TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Uso de Inteligência Artificial na animação de Agentes Autônomos Inteligentes em jogos

Brian Schmitz Tani

Ciências da Computação

Resumo

Esta pesquisa visa estudar diversas técnicas de animação de agentes inteligentes, buscando conciliar os estudos com o ambiente propício para jogos em terceira dimensão. Além de verificar quais técnicas estão sendo usadas na área de inteligência artificial para atingir tal objetivo.

Palavras chave: Agentes Autônomos, Agentes Autônomos Inteligentes, Animação de Agentes.

Abstract

This research aims to study several techniques for animating intelligent agents, conciliating those studies with an environment suited for 3d games. Furthermore, we will verify which techniques are being used in the artificial intelligence area to achieve this goal.

Keywords: Autonomous Agent, Intelligent Autonomous Agent, Agent Animation.

1 A Inteligência Artificial em Jogos

Os jogos modernos estão exigindo cada vez mais complexidade. E de longa data utilizam conceitos de inteligência artificial em diversos níveis. O interesse público neste assunto, como Nareyek (2002) aponta, é tanto pelo mercado multibilionário dos jogos como pelo interesse no comportamento inteligente nos jogos.

Existem diversas técnicas usadas para conferir certo grau de inteligência em agentes de jogos. E segundo Paul Tozur (2002a), da Ion Storm Austin, a inteligência

artificial é mais bem usada em parceria com bons projetos de jogos (game design), uma simbiose entre os dois por assim dizer. Sendo assim não se trata somente de que técnica usar, mas sim como usa-la de acordo com o projeto do jogo. Algumas técnicas de inteligência artificial podem ser muito boas em jogos de estratégia baseados em turnos, mas a mesma técnica poderia ser inviável em jogos em tempo real.

Como visto em Niederberger (2002) a inteligência artificial pode ser definida como a área da ciência da computação voltada a atribuir um comportamento inteligente a máquinas, seja este comportamento humano ou não. E desta forma pode ser usada em diversas áreas do conhecimento, desde economia até jogos. É possível classificar a Inteligência Artificial neste aspecto em quatro grandes abordagens:

- **Agir Humanamente**: Esta abordagem visa dar uma característica humana à máquina. E para testar isso se usa o teste de Turing, onde um participante humano tenta identificar somente através de conversas em bate-papos quais dos outros participantes é uma máquina.
- **Pensar Humanamente**: Pode ser coberto pelo campo da ciência cognitiva, associando modelos da IA com técnicas experimentais do campo da psicologia. Nesta abordagem o objetivo é imitar o mais completamente possível o processo cognitivo humano, sendo a solução deixada em segundo plano, comparada ao processo que levou a ela.
- **Pensar Racionalmente**: Usar ferramentas lógicas para resolver problemas. No entanto esta abordagem requer um modelo acurado da realidade, o que a torna, por muitas vezes, impraticável. E mais, fazer funcionar algo em teoria é diferente de fazê-lo na prática.

- **Agir Racionalmente**: Agir no intuito de alcançar um objetivo próprio. Que leva a criação de agentes racionais, que agem de forma a inferir sobre que maneira deverá agir dada uma circunstância, além de tomar decisões baseadas em dados incompletos, ou não baseadas em inferências (ex. Reflexos).

Em todas estas abordagens várias áreas da inteligência artificial são estudadas. E mesmo essas áreas não são mutuamente exclusivas, elas se sobrepõem e tentam atacar um problema de diversos ângulos. No entanto, os jogos modernos têm características próprias como visto em Nareyek (2002):

- **Tempo Real** – pouco tempo de processamento disponível. Raciocínios complexos requerem maior processamento, porém garantem respostas mais acuradas.
- **Dinamismo** – Jogos provêm um ambiente altamente dinâmico. Isto significa que os personagens devem ser capazes de lidar com diversos tipos de situações, e muitas vezes situações não previsíveis.
- **Conhecimento Incompleto** – Um personagem de jogo não tem conhecimento completo do mundo o qual está inserido. É necessário computar quão longe e o que um determinado personagem consegue enxergar ou ouvir. Além de saber navegar em qualquer ambiente, de forma correta. Ou seja, no caso de um jogo de tiro, em primeira pessoa, não se espera que um personagem atravesse paredes ou que fique preso em um determinado local.
- **Recursos** – O mundo do personagem ou seus recursos podem ser restritos.

Nareyek(2002) explica claramente a importância dessas técnicas e IA nos jogos e como se é vista a personificação desta nestes jogos quando diz:

> "Técnicas de IA podem ser aplicadas em uma variedade de tarefas nos jogos modernos de computador. Um jogo que usa redes probabilísticas para predizer o movimento que o jogador irá fazer, e deste modo aumentar a taxa de amostragem de gráficos, pode estar em um nível alto de IA. Mas apesar da IA nem sempre ser personificada, a noção de inteligência artificial em jogos de computadores é principalmente relacionada a personagens. Estes personagens podem ser vistos como *agentes*, suas propriedades se encaixando perfeitamente no conceito de agente de inteligência artificial".(NAREYEK, 2002).

2 Agentes

Mas o que é um agente? A noção de agente na literatura tem várias interpretações, alguns autores até dizem não ser possível dar uma definição que acate a todas as demandas.

Niederberger(2002) define um agente como sendo um sistema, seja físico o em software, com as seguintes propriedades:

- **Autonomia**: agentes devem operar sem intervenção direta de humanos ou outros, e ter algum controle sobre suas ações e estados internos.
- **Habilidade Social**: agentes devem poder se comunicar com outros agentes ou humanos através de algum tipo de linguagem de comunicação de agentes.
- **Reatividade**: um agente deve perceber o ambiente a sua volta e responder de forma adequada e em tempo hábil.
- **Pró-atividade**: um agente não deve ser somente reativo, deve ser tomar iniciativa para que possa atingir seu objetivo.

Se restringirmos a visão do que é agente para os vermos como um softbot, ou agentes de software, Etzioni tem uma definição mais apropriada:

> "Um *softbot* é um agente que interage com um ambiente em software através de comandos sensores e interpretando a resposta do ambiente. Os atuadores dos softbots são comandos que mudam o estado do ambiente exterior. Os sensores dos softbots são comandos que fornecem informações". (ETZIONI, 1994 apud NIEDERBERGER, 2002).

Em suma um agente inteligente pode ser visto como uma entidade autônoma que interage com o ambiente, com outros agentes ou com humanos através de uma linguagem própria, ou reagindo a estímulos e tomando a iniciativa com pró-ações que o favorecem. E além destas propriedades é, ou conceituado ou implementado usando conceitos mais aplicados a humanos.

2.1 Agentes autônomos

Segundo Niederberger (2002), e respaldado na tese de Nareyek (2002), um agente ideal é aquele que sempre toma a ação que é esperada para maximizar sua medida de performance, dada as percepções, até o momento, vistas. Um agente é autônomo até o ponto que suas escolhas de ações dependem de sua própria experiência, em vez de se basear em conhecimentos ditados pelo desenvolvedor (RUSSEL, Stuart J. e NORVIG, Peter, 1996).

Um programa agente consiste principalmente da execução repetida de tarefas como percepção, inferência, seleção e ação.

A tarefa de percepção coleta informações sobre o ambiente, seu estado desde a última ação. Com essas informações e baseado no estado do sistema após a última ação o agente consegue inferir o que tem de ser feito a seguir. Com isso ele terá um número de possíveis ações o qual deverá selecionar uma e então executá-la.

Nós podemos então dividir esses agentes autônomos por ordem de complexidade.

Tipos de agentes:

- **Agente Reativo ou baseado em ação-reação.**

Estes agentes possuem um modelo bem simples no qual percebem o ambiente, e baseado em uma condição pré-imposta executam uma ação. São muito rápidos e se o sistema não for complexo, tende a ser uma boa solução. No entanto sua aplicação em jogos é muito restrita, e mesmo se introduzirmos o conceito de máquina de estados, como Niederberger sugere, ainda sim sua aplicação seria pequena. Pois em ambos os casos o desenvolvedor deverá prever todas as possibilidades de reações.

Ex: se (motor_falhando) então {carro.pare}.

- **Agente com base em objetivos ou agentes deliberativos**.

Quando introduzimos o conceito de busca de objetivos significa que o agente não mais responde somente a percepções do ambiente externo. Ele as analisa tentando avaliar a melhor solução que o leve a atingir seu objetivo. Nareyek diz isso claramente.

> "(...)Os objetivos e um modelo de mundo contendo informações sobre os requisitos da aplicação e as conseqüências de ações são representadas explicitamente. Um sistema interno de planejamento baseado em refinamento usa a informação do modelo de mundo para criar um plano que alcança o objetivo do agente."
> "O problema básico do planejamento é dado pela descrição inicial do mundo, uma descrição parcial do objetivo do mundo, e um conjunto de ações/operadores que mapeiam uma descrição parcial de mundo para uma outra descrição parcial de mundo. A solução é a seqüência de ações que levam da descrição inicial do mundo para a descrição do objetivo do mundo e é chamado de plano. O problema pode ser enriquecido incluindo-se mais aspectos, como problemas temporais e incertezas, ou requerendo a otimização de certa propriedade. (...)."
> "O problema com agentes deliberativos é sua lentidão. Todas as vezes que a situação é diferente da que foi antecipada pelo processo de planejamento, o plano deve ser recomputado. Computar planos pode ser muito demorado, e considerando os requisitos de tempo-real em um ambiente complexo se torna fora de questão" (NAREYEK, 2002)

- **Agente baseado em otimização (Utility-based Agents).**

Agentes baseados em otimização vão além dos agentes baseados em objetivos, quando analisam quais ações devem seguir baseados em quão satisfeitos ficarão com o resultado.

Um exemplo para este tipo de agente pode ser este: Ao escolher atravessar o rio o agente se depara com as seguintes soluções: nadar, pegar um barco ou cruzar a ponte. Ele está sem dinheiro para pagar o pedágio da ponte, então ele descarta essa solução, e quaisquer das duas outras soluções o fariam atravessar o rio, mas ele escolhe pegar o barco, pois não ficará molhado.

- **Agentes Híbridos**

Estes agentes usam de outros tipos de arquitetura de agentes para formar uma arquitetura híbrida que se espera ser capaz de resolver certos problemas. Um exemplo seria o uso de uma arquitetura que se assemelhasse a um agente baseado em objetivo, porém quando não fosse possível recomputar seu plano, assumia uma resposta reativa. Ainda sim, não é rápido o suficiente para os requisitos dos jogos modernos, podendo criar planos para coisas que já aconteceram.

- **Agente Qualquer-Hora (Anytime Agent)**

Este tipo de agente é chamado assim, pois computa seus planos somente quando possível e somente por quanto tempo for permitido. Deste modo ele sempre terá um plano, mesmo quando tiver pouco tempo. Eles trabalham dinamicamente, aperfeiçoando seus planos à medida que tem tempo disponível, porém ao final do tempo respondem com uma ação baseada no plano que computaram até o momento, mesmo que seja imperfeito. Assim consegue-se uma continuidade entre reação e planejamento. Nareyek diz.

> "Para horizontes de curto tempo de computação, somente planos primitivos (reações) estarão disponíveis, tempos mais longos de computação serão usados para melhorar e otimizar o plano do agente. Quanto mais tempo estiver disponível para as computações do agente, mais inteligente será o comportamento resultante. Adiante, a melhoria iterativa habilita o processo de planejamento a facilmente se adaptar a situações inesperadas ou que mudaram. Esta classe de agentes é muito importante para aplicações de jogos de computador." NAREYEK, 2002.

Com isso podemos dizer que os agentes anytime são os mais adaptados para os ambientes dos jogos que exijam grande verossimilhança com a realidade. Niederberger (2002) comenta que estes ambientes são acessíveis, não-determinístico, e não episódico. Comenta também que para jogo de um jogador, o ambiente pode ser estático, invariável, porém para jogos distribuídos de múltiplos jogadores, pode ser altamente dinâmico, dependendo da implementação. Jogos podem ser ainda, discretos ou contínuos; enquanto jogos do tipo “pula-e-corre” são discretos, um simulador de vôo ou um MMORPG são contínuos.

No entanto antigamente não se havia o poder de processamento requerido para comportamentos mais complexos, então normalmente usava-se de artifícios como aplicar regras diferenciadas aos NPCs. Tornando o jogo um pouco mais difícil, pois o jogador jogava em desigualdade. Porém apesar de ainda usar tais artimanhas, hoje em dia mais e mais tempo de processamento está sendo dedicado somente às tarefas de computação para IA.

3 Requisitos da Implementação

Para implementar o agente autônomo era necessários certos requisitos como um bom motor gráfico, que pudesse satisfazer as necessidades propostas, modelos que pudessem ser usados e uma boa estrutura de um jogo 3d.

3.1 Motor Gráfico

Para resolver o problema do Motor Gráfico foi pesquisada uma variedade de motores gráficos, dentre eles o escolhido foi o IRRLicht pelas seguintes características:

- Interface simplificada; Tornando o tratamento de funções e aspectos da criação de um jogo muito mais simples.

- Integra-se ao Visual C++ de maneira rápida e simples;
- Exemplos bem documentados;
- Comunidade bem ativa.

3.2 Modelo de animação

Tendo o motor gráfico escolhido resolveu-se usar um modelo 3d gratuito para facilitar o trabalho de implementação. Como os modelos MD2 são bastante difundidos e outros modelos não tem tanta divulgação e por essa razão não são amplamente usados em produções independentes, a sua escolha foi direta.

O modelo adquirido segue o padrão MD2 com suas ações previamente animadas e armazenadas no arquivo.

3.3 Estrutura de um Jogo

Era necessário criar agora uma estrutura simples de um jogo para que pudesse haver um mundo o qual o agente deveria navegar.

A navegação neste mundo é importante e deve-se valer de sentidos que avisem o agente que ele está diante de uma parede ou chão. Deve haver um sentido de força de gravidade, para que ele possa andar sobre uma superfície.

O motor gráfico provê todas essas facilidades, mas é necessário programa-las para que seja possível utiliza-las.

A tarefa de programação levou ao uso de um *framework* para jogos tipo RPG. Esse *framework* chama-se IRRLichtRPG e foi criado por Paul Zickle, um membro ativo da comunidade IRRLicht.

Este *framework,* embora incompleto, continha as características básicas de um jogo como um sistema de eventos para passar mensagens, uma hierarquia de objetos para identificar objetos estáticos do mundo, como caixas e afins, com objetos dinâmicos como criaturas.

Para clarificar o uso deste *framework* é conveniente apresentar um diagrama de classes das estruturas mais importantes.

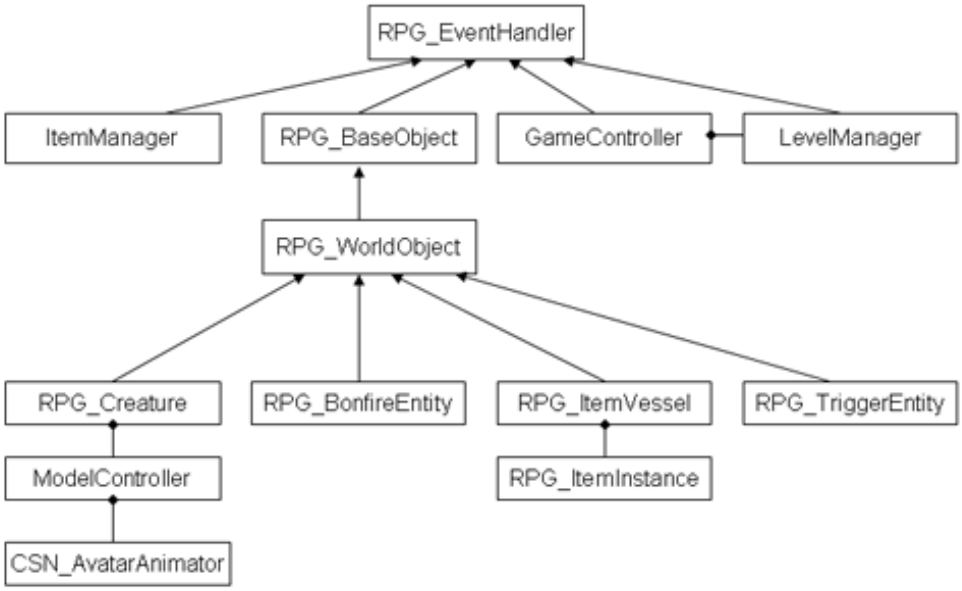


Diagrama de classe relevantes do IRRLichtRPG *framework.*

Neste diagrama de classe a classe que seria usada para basear o agente inteligente seria a classe RPG_Creature. Ela descreve uma entidade que se movimenta no mundo virtual por própria vontade. Essa mesma classe pode ser usada para descrever tanto NPCs e avatares de jogadores. Ela é uma especialização da classe RPG_WorldObject que descreve objetos estáticos sujeitos a forças externas.

As classes GameController e LevelManager coordenam e monitoram aspectos específicos da lógica do jogo, além de se encarregar de repassar os eventos para as entidades a medida que ocorrem.

4 Modelagem proposta

Para modelar um sistema de inteligência artificial que controle uma entidade é possível criar uma classe AI_Controller que repassaria os comandos a entidade de

maneira igual ou similar a maneira que o jogador controla uma entidade. Seria o cérebro da criatura usando uma analogia biológica.

O gerente de nível ou *level manager* deve passar os eventos que ocorrem ao AI_Controller. E este, utilizando uma técnica de inteligência artificial, determinar a ação da entidade associada a ele.

4.1 Inteligência Artificial

No aspecto de inteligência artificial foram buscadas maneiras de implementar os conceitos vistos anteriormente.

Para começar os protótipos neste aspecto fio escolhido uma máquina de estados finitos para o comportamento de um agente baseado em ação-reação. Seria composto de estados simples como “lutar”, “vagar”, “fugir”. Cada estado seria, então, aperfeiçoado. Mais ou menos como proposto por Phil Carlisle(2002).

O modelo possui cinco estados (“ocioso”, “vagar”, “fugir”, “lutar”, “curar”), quatro condições (“inimigo avistado”, “fora de alcance”, ”curado” e “pouca vida”) monitorando duas variáveis (“vida” e “localização do inimigo”).

A figura a seguir mostra as transições de cada e estado e em que condições elas são efetuadas.

Além disso, cada estado tem ações associadas a eles. O estado ocioso não possui ações especificas. Neste estado o agente apenas permanece parado esperando determinados eventos ocorrerem.

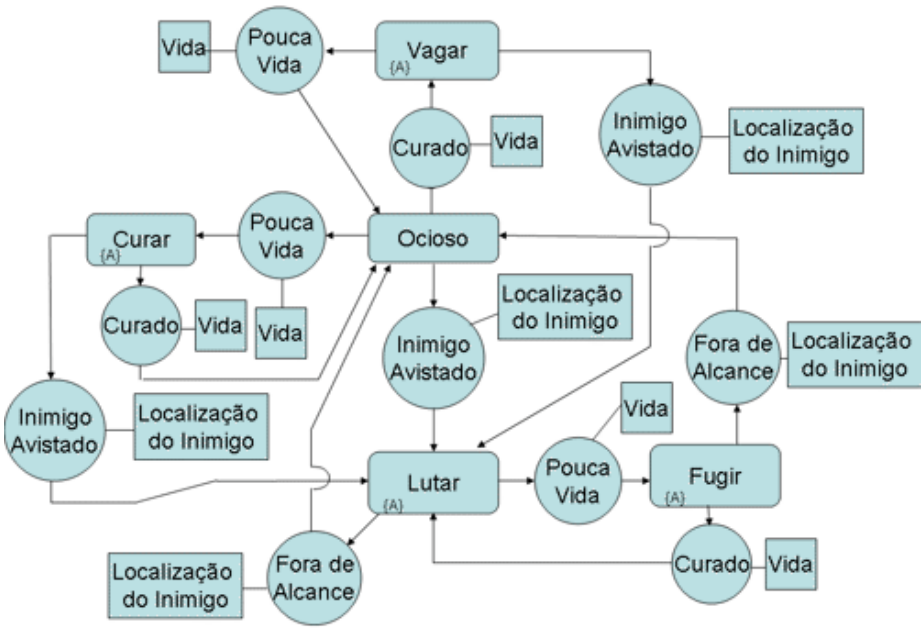


Diagrama de estados do modelo proposto de comportamento do agente.

Quando a condição de inimigo avistado é satisfeita o agente vai para o estado de lutar. Essa transição é uma maneira preventiva. Pois se no próximo ciclo a condição de pouca vida for satisfeita ele tenta fugir, mas pelo menos no primeiro ciclo, se houver oportunidade ele efetua pelo menos um ataque.

No estado de fugir ele vai tentar manter o inimigo fora de sua linha de visão, fugindo de costas para longe. Quando não conseguir mais enxergar o inimigo ele resume ao estado ocioso. Caso se cure durante a fuga ele volta ao estado lutar.

Se lutando o inimigo desaparece, caso morra ou fuja, ele volta ao estado ocioso.

No estado ocioso, caso o inimigo não for avistado e o agente estiver curado, ele passa o estado vagar. Neste estado ele percorre o mundo até encontrar um inimigo, ou ficar com pouca vida. Se ficar com pouca vida ele resume ao estado ocioso, se encontrar um inimigo ele muda para o estado lutar.

Se ele estiver no estado ocioso e tiver pouca vida, ele passa a se curar quando curado volta ao estado ocioso. Mas se no processo de cura ele avista um inimigo ele passa ao estado Lutar.

Para implementar essa função foi criado um método “AI_Controller::think(irr::u32 enemySpoted, irr::core::vector3df colision)” que receberia o número de identificação do inimigo, e o vetor de colisão. A intenção era averiguar se um inimigo havia sido visto, e se estava longe ou perto.

Com o controlador monitorando a saúde ou vida da criatura a qual estava relacionada seria possível identificar se as condições de transição eram satisfeitas ou não.

A cada ciclo de jogo (*game cycle*) o método executava uma ação, ou passo de ação, no caso da ação ser perseguir ou fugir. Ou mudaria o estado da máquina de estados. Estes estados são definidos como uma estrutura enumerada chamada tAI_STATE com os seguintes valores: AIS_IDLE, AIS_WANDER, AIS_FIGHT, AIS_FLEE, AIS_HEAL.

Em caráter temporário o AI_Controller teria alguns outros aspectos que mais tarde seriam implementados em classes diferentes. Os atributos, m_goodHP e m_lowHP, eram atributos que definiam o como considerar um agente curado ou com pouca vida. AI_BUSY determinava se o controlador estava no meio de uma tarefa. Como por exemplo, movendo em direção a um ponto especifico. m_enemyId determinava um inimigo específico que o controlador deveria monitorar. m_visionArc determinava o arco de visão em graus que o agente tinha, Sendo possível determinar se inimigo podia ser visto. m_curState apenas dizia o estado atual da máquina de estados. E o atributo m_body era um ponteiro para a entidade que o controlador deveria controlar.

5 Resultados

A modelagem de um sistema simples de inteligência artificial utilizando-se de uma máquina de estados finitos para descrever o comportamento de um agente. Esta máquina de estados finitos torna o agente um agente reativo ou ação-reação, como visto nos trabalhos de Niederberger (2002) e Russel e Norvig (1996).

Este sistema foi visto como uma boa maneira de familiarizar-se com as ferramentas. A idéia era criar protótipos cada vez mais complexos. E ao final introduzir técnicas de aprendizagem ao agente.

Os protótipos produzidos chegavam perto de implementar tal modelo. Testando características simples de passagem de mensagem e controle por meio de manipulação do sistema de eventos do *framework* de jogos tipo RPG.

O sistema de eventos do IRRlichtRPG não estava 100% pronto, mas estava utilizável. A mesma classe que definia um objeto tipo criatura era usada tanto para definir o PC quanto para definir um NPC. Assim o sistema criado deveria poder controlar o NPC usando a mesma interface que o Jogador usaria para controlar o PC.

A Máquina de estados foi, então, implementada nesta arquitetura e o sistema de eventos usado para passar as mensagens como explicado anteriormente.

6 Conclusões

O estudo do desenvolvimento de um jogo foi de suma importância a este projeto, visto que nas referências sobre inteligência artificial assumia-se que o desenvolvedor já obtivesse ferramentas que possibilitassem a manipulação de aspectos de jogos. Sendo a proposta criar um agente inteligente para jogos tridimensionais tipo RPG, foi essencial obter não somente um bom motor gráfico, porém desenvolver uma arquitetura simples que simule um jogo.

Felizmente obteve-se o IRRLichtRPG que possibilitou a continuidade do projeto, pois sem ele foi necessário desenvolver uma arquitetura antes de passar a desenvolver a inteligência artificial. O que atrasou consideravelmente o cronograma.

Viu-se então que ter um motor gráfico potente é apenas uma pequena, embora importante, parte do desenvolvimento de um jogo. Estudando a literatura e através prática averiguou-se que as técnicas de inteligência artificial para criar um agente de jogo estão intimamente ligadas com o design do jogo desenvolvido e que dar a percepção de inteligência ao um agente não significa que o agente seria modelado para imitar exatamente o comportamento de um jogador. O agente pode ter ajuda de um mundo bem estruturado e truques que dão a percepção de inteligência a um agente.

Referências Bibliográficas

ALEXANDER, Robert. **AI Programming Wisdom**: The Beauty of Response Curves. p. 78-82. Hingham, Massachusetts: Charles River Media, 2002.

ALIAS|WAVEFRONT. **The Art of Maya**. Character Animation. p. 100 - 114. ISBN-1894893131

CARLISLE, Phil. **AI Programming Wisdom**: Designing a GUI Tool to Aid in the Development of Finite-State Machines. p. 71-77. Hingham, Massachusetts: Charles River Media, 2002.

CERVO, Amando Luiz; BERVIAM, Pedro Alcino. **Metodologia Científica.** 4ºed. São Paulo: Makron Books, 1996. p.20-22,44-45,48-48.

CHAMPANDARD, Alex J. **AI Programming Wisdom**: The Dark Art of Neural Networks. p. 640-651. Hingham, Massachusetts: Charles River Media, 2002.

CHAMPANDARD, Alex J. **The Future of Game AI**: Intelligent Agents. Disponível em: <http://ai-depot.com/GameAI/Agent-Intelligence.html>. Acessado em: 24 jul. 2003.

CONTINUUM Entertainment. Disponível em: <http://www.continuum.com.br>. Acessado em: 27 abr. 2003.

DFC Intelligence. Sessão de Artigos da Indústria de Jogos. *Game Industry Market Forecasts.* Disponível em: <*http://www.dfcint.com/game_article/mar03article.html*>. Acessado em: 28 abr. 2003.

DFC Intelligence. Sessão de Artigos da Indústria de Jogos. *Game Industry Market Forecasts part 2.* Disponível em: <*http://www.dfcint.com/game_article/april03article.html*>. Acessado em: 28 abr. 2003.

ESPAÇO Informática Ltda. Disponível em: <http://www.hades2.com/espaco/>. Acessado em: 27 abr. 2003.

FALSHIN, Odília. **Fundamentos de Metodologia.** São Paulo: Atlas, 1993. p. 36-38,101-104.

GAMASUTRA. **Game AI:** The State of the Industry. Disponível em: <http://www.gamasutra.com/features/20001101/woodcock_01.htm>. Acessado em: 25 de jun. 2003.

ECLIPSE Entertainment. Genesis3D Engine. Disponível em: <http://www.genesis3d.com/>. Acessado em: 13 de Jan. de 2004.

GREENLAND Studios. Disponível em: <http://www.greenlandstudios.com>. Acessado em: 27 abr. 2003.

HIGGINS, Dan. **AI Programming Wisdom**: Generic A* Pathfinding. p. 114-121. Hingham, Massachusetts: Charles River Media, 2002.

HUANG, Yao-Chin. **Prototyping**: Throwaway or Evolutionary? 1998. Disponível em: <http://www.scis.nova.edu/~yaochin/660-p.htm>. Acessado em: 3 ago. 2003.

IGNIS Games. IGNIS Entretenimento e Informática S/C Ltda. Disponível em: <http://www.ignisgames.com.br>. Acessado em: 27 abr. 2003.

IGNIS Games. FAQ do Jogo Erynis. Disponível em: <http://www.ignisgames.com.br/erynis/index.php>. Acessado em: 27 abr. 2003.

IRRLicht Engine. Disponível em: <http://irrlicht.sourceforge.net> Acessado em: 20 de Dez. de 2003.

LARAMÉE, François Dominic. **AI Programming Wisdom**: Evolving the Perfect Troll. p. 629-639. Hingham, Massachusetts: Charles River Media, 2002.

LUCAS ARTS entertainment. Sessão de ofertas de trabalhos. Disponível em: <http://www.lucasarts.com/jobs>. Acessado em: 28 de Jun. de 2003.

NAREYEK, Alexander. **Constraint-Based Agents** - An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds. 2001. 176p. Tese (PhD em Ciência da Computação) - Technical University of Berlin, Departamento de Ciência da Computação, Alemanha. Disponível em: <http://www.ai-center.com/references/nareyek-01-cba.html>. Acessado em: 27 jun. de 2003.

RADON LABS. The Nebula Device 1 e 2. Disponíveis em: <http://nebuladevice.sourceforge.net/>. Acessado em: 05 de Jan. de 2004.

NIEDERBERGER, Christoph. GROSS, H. Markus. **Towards a Game Agent**. Computer Science Dep. CS Technical Report #377. ETH, Zurique – Suíça. 26 de ago. de 2002.

PARALELO COMPUTAÇÃO. Fly3d Engine. Disponível em: <http://www.fly3d.com.br/>. Acessado em: 20 de Jan. de 2004.

RUSSEL, Stuart J. NORVIG, Peter. *Artificial Intelligence - A Modern Approach*. PrenticeHall, 1996. **ISBN -** 0137903952

SALOMON, Délcio Vieira. **Como fazer uma Monografia de trabalho científico.** Belo Horizonte: Interlivros, 1978. p.137-138,140-147.

TOZOUR, Paul. **AI Programming Wisdom**: The Evolution of Game AI. p. 3-15. Hingham, Massachusetts: Charles River Media, 2002.

TOZOUR, Paul. **AI Programming Wisdom**: Building a Near-Optimal Navigation Mesh. p. 171-185. Hingham, Massachusetts: Charles River Media, 2002.

VILIEGAS, Renato. A Reta Final. **Electronic Gaming Monthly Brasil**. São Paulo. n.13. p.49-53. 2003.

VOSINAKIS, Stypos. PANAYIOTOPOULOS, Themis. **SimHuman**: A Platform for Real-Time Virtual Agents with Planning Capabilities. Knowledge Engeneering Laboratory. Department of Informatics, University of Piraeus. Grécia, 2001.

ZAROZINSKI, Michael. **AI Programming Wisdom**: An Open-Source Fuzzy Logic Library. p. 90-101. Hingham, Massachusetts: Charles River Media, 2002.

ZIRKLE, Paul. **IRRLichtRPG.** Disponível em: <http://www.skyesurfer.net/keless/IrrLicht/RPG/>. Acessado em 22 de Abril de 2004.

7.2 Anexo II – Código-Fonte do Projeto

7.2.1 – AI_States.h

```
//Enumarate all states for the State Machine

enum tAI_STATE {
 AIS_IDLE = 0,      //Standing
 AIS_WANDER,     //Moving Around
 AIS_FIGHT,         //Fighting
 AIS_FLEE,             //Fleeing
 AIS_HEAL,             //Healing

 AIS_NUM_STATES
};
```

7.2.2 – AI_Controller.cpp

```
// ai_controller.cpp: implementation of the AI_Controller class.
//
//////////////////////////////////////////////////////////////////////
#include "ai_controller.h"
```

```
//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

AI_Controller::AI_Controller(RPG_Creature* body, rpg_EventHandler*
eventPortal)
{
      //Temporary::may change to fuzzy
      m_goodHP = 75;              //Minimum HP to consider good
      m_lowHP = 25;                     //Maximum HP to consider low
      m_healingFactor = 1;          //healing factor

      ai_in_progress = false;

      m_enemyId = -1;                   //No Enemies assingned
      m_visionArc = 120;                //120 degrees
      m_minDistance = 20.0f;        //Attack Range

      m_curState = AIS_IDLE;        //Idle state
      m_body = body;                    //sets the body pointer;

      turning_right = false;
      turning_left = false;
      moving_forward = false;
      moving_backwards = false;
      stuck = false;

      m_maxTimeBetweenAttacks = 30;
      m_timeBetweenAttacks = 0;
      m_maxTimeActions = 20;
      m_timeActions = 0;
      m_maxTimeOut = 100;
      m_timeOut = 0;

      m_myLastPos = m_body->get3DModel()->getNode()->getPosition();

      m_pEventPortal = eventPortal;
      printf("AI_CONTROLLER creation: Success! Attached to creature
id: %d\n",body->getID());
      printState();
}

AI_Controller::~AI_Controller()
{

}
bool AI_Controller::moveTorwardDestination(){
      irr::core::vector3df l_size =
irr::core::vector3df(1000.0f,0.0f,0.0f);
      irr::core::vector3df l_start = m_body->get3DModel()->getFrontPoint();
      irr::core::vector3df l_end = l_start + m_body->get3DModel()-
>getFacing() + l_size;
      //moving torwards destination
      //printf("moving torward destination
[%f,%f,%f]\n",m_tgtDestination.X,m_tgtDestination.Y,m_tgtDestination.Z);
      if((m_tgtDestination.X > l_start.X) && (m_tgtDestination.Z <
l_start.Z - 5.0f) && (m_tgtDestination.Z < l_end.Z - 5.0f)) {
            //Rotate to the right
            //printf("Rotate RIGHT");
            if (!turning_right){
```

```
                  turning_right = true;
                  turning_left = false;
                  moving_forward = false;
                  moving_backwards = false;

                  SRPG_Event newEvt;
                  newEvt.EventType = RPG_EVT_CREATURE;
                  newEvt.Creature.CreatureEventType = RPG_CEVT_MV_RIGHT;
                  newEvt.Creature.cid = m_body->getID();
                  newEvt.Creature.stopped = false;
                  m_pEventPortal->OnRPGEvent( newEvt );
            }
            return true;
      }else{
            turning_right = false;
            SRPG_Event newEvt;
            newEvt.EventType = RPG_EVT_CREATURE;
            newEvt.Creature.CreatureEventType = RPG_CEVT_MV_RIGHT;
            newEvt.Creature.cid = m_body->getID();
            newEvt.Creature.stopped = true;
            m_pEventPortal->OnRPGEvent( newEvt );
      }


      if((m_tgtDestination.X > l_start.X) && (m_tgtDestination.Z >
l_start.Z + 5.0f ) && (m_tgtDestination.Z > l_end.Z + 5.0f)){
            //Rotate to the left
            //printf("Rotate LEFT");
            if (!turning_left){
                  turning_right = false;
                  turning_left = true;
                  moving_forward = false;
                  moving_backwards = false;

                  SRPG_Event newEvt;
                  newEvt.EventType = RPG_EVT_CREATURE;
                  newEvt.Creature.CreatureEventType = RPG_CEVT_MV_LEFT;
                  newEvt.Creature.cid = m_body->getID();
                  newEvt.Creature.stopped = false;
                  m_pEventPortal->OnRPGEvent( newEvt );
            }
            return true;

      }else{
            turning_left = false;
            SRPG_Event newEvt;
            newEvt.EventType = RPG_EVT_CREATURE;
            newEvt.Creature.CreatureEventType = RPG_CEVT_MV_LEFT;
            newEvt.Creature.cid = m_body->getID();
            newEvt.Creature.stopped = true;
            m_pEventPortal->OnRPGEvent( newEvt );
      }
      if((m_tgtDestination.X > l_start.X) &&
            (m_tgtDestination.Z <= l_start.Z + 5.0f) && (m_tgtDestination.Z
>= l_start.Z - 5.0f) &&
            (m_tgtDestination.Z <= l_end.Z + 5.0f) && (m_tgtDestination.Z
>= l_end.Z - 5.0f)){
            if (!moving_forward){
                  turning_right = false;
                  turning_left = false;
                  moving_forward = true;
```

```
                  moving_backwards = false;

                  SRPG_Event newEvt;
                  newEvt.EventType = RPG_EVT_CREATURE;
                  newEvt.Creature.CreatureEventType = RPG_CEVT_MV_FORWARD;
                  newEvt.Creature.cid = m_body->getID();
                  newEvt.Creature.stopped = false;
                  m_pEventPortal->OnRPGEvent( newEvt );
            }
            return true;
      }
      if (moving_forward){
            SRPG_Event newEvt;
            newEvt.EventType = RPG_EVT_CREATURE;
            newEvt.Creature.CreatureEventType = RPG_CEVT_MV_FORWARD;
            newEvt.Creature.cid = m_body->getID();
            newEvt.Creature.stopped = true;
            m_pEventPortal->OnRPGEvent( newEvt );
            moving_forward = false;
      }
      return false;
}

bool AI_Controller::moveAwayFromDestination(){

      irr::core::vector3df l_size =
irr::core::vector3df(1000.0f,0.0f,0.0f);
      irr::core::vector3df l_start = m_body->get3DModel()->getFrontPoint();
      irr::core::vector3df l_end = l_start + m_body->get3DModel()-
>getFacing() + l_size;
      //moving torwards destination
      //printf("moving torward destination
[%d,%d,%d]\n",m_tgtDestination.X,m_tgtDestination.Y,m_tgtDestination.Z);

      if((m_tgtDestination.X > l_start.X) && (m_tgtDestination.Z >
l_start.Z + 5.0f) && (m_tgtDestination.Z > l_end.Z +5.0f)){
            //Rotate to the right
            //printf("Rotate RIGHT");
            if (!turning_right){
                  turning_right = true;
                  turning_left = false;
                  moving_forward = false;
                  moving_backwards = false;

                  SRPG_Event newEvt;
                  newEvt.EventType = RPG_EVT_CREATURE;
                  newEvt.Creature.CreatureEventType = RPG_CEVT_MV_RIGHT;
                  newEvt.Creature.cid = m_body->getID();
                  newEvt.Creature.stopped = false;
                  m_pEventPortal->OnRPGEvent( newEvt );
            }
            return true;
      }else{
            turning_right = false;
            SRPG_Event newEvt;
            newEvt.EventType = RPG_EVT_CREATURE;
            newEvt.Creature.CreatureEventType = RPG_CEVT_MV_RIGHT;
            newEvt.Creature.cid = m_body->getID();
            newEvt.Creature.stopped = true;
            m_pEventPortal->OnRPGEvent( newEvt );
      }
```

```
      if((m_tgtDestination.X > l_start.X) && (m_tgtDestination.Z <
l_start.Z -5.0f) && (m_tgtDestination.Z < l_end.Z -5.0f)) {

            //Rotate to the left
            //printf("Rotate LEFT");
            if (!turning_left){
                  turning_right = false;
                  turning_left = true;
                  moving_forward = false;
                  moving_backwards = false;

                  SRPG_Event newEvt;
                  newEvt.EventType = RPG_EVT_CREATURE;
                  newEvt.Creature.CreatureEventType = RPG_CEVT_MV_LEFT;
                  newEvt.Creature.cid = m_body->getID();
                  newEvt.Creature.stopped = false;
                  m_pEventPortal->OnRPGEvent( newEvt );
            }
            return true;
      }else{
            turning_left = false;
            SRPG_Event newEvt;
            newEvt.EventType = RPG_EVT_CREATURE;
            newEvt.Creature.CreatureEventType = RPG_CEVT_MV_LEFT;
            newEvt.Creature.cid = m_body->getID();
            newEvt.Creature.stopped = true;
            m_pEventPortal->OnRPGEvent( newEvt );
      }
      if((m_tgtDestination.X > l_start.X) &&
            (m_tgtDestination.Z <= l_start.Z + 5.0f) && (m_tgtDestination.Z
>= l_start.Z - 5.0f) &&
            (m_tgtDestination.Z <= l_end.Z + 5.0f) && (m_tgtDestination.Z
>= l_end.Z - 5.0f)){

            if (!moving_backwards){
                  turning_right = false;
                  turning_left = false;
                  moving_forward = false;
                  moving_backwards = true;

                  SRPG_Event newEvt;
                  newEvt.EventType = RPG_EVT_CREATURE;
                  newEvt.Creature.CreatureEventType = RPG_CEVT_MV_BACKWARD;
                  newEvt.Creature.cid = m_body->getID();
                  newEvt.Creature.stopped = false;
                  m_pEventPortal->OnRPGEvent( newEvt );
            }
            return true;
      }
      if (moving_backwards){
            SRPG_Event newEvt;
            newEvt.EventType = RPG_EVT_CREATURE;
            newEvt.Creature.CreatureEventType = RPG_CEVT_MV_BACKWARD;
            newEvt.Creature.cid = m_body->getID();
            newEvt.Creature.stopped = true;
            m_pEventPortal->OnRPGEvent( newEvt );
            moving_backwards = false;
      }
      return false;
}
```

```
void AI_Controller::attack(RPG_ID_TYPE hit){

	if(m_timeBetweenAttacks==0){

		printf("Attacking enemy: %d\n",hit);

		SRPG_Event attackEvt;
		attackEvt.EventType = RPG_EVT_CREATURE;
		attackEvt.Creature.CreatureEventType = RPG_CEVT_ATTACK;
		attackEvt.Creature.cid = m_body->getID();
		attackEvt.Creature.stopped = false;
		m_pEventPortal->OnRPGEvent( attackEvt );

	}
}

bool AI_Controller::think(RPG_ID_TYPE hit, irr::core::vector3df position){
	//printf("Thinking: [%d][HEALTH=%d]\n",hit,*m_body->getHP());
	updateCounter();
	switch(m_curState){
		case AIS_IDLE:
			if (moving_forward){
				printf("!!!STOPPED!!!");
				SRPG_Event newEvt;
				newEvt.EventType = RPG_EVT_CREATURE;
				newEvt.Creature.CreatureEventType =
RPG_CEVT_MV_FORWARD;
				newEvt.Creature.cid = m_body->getID();
				newEvt.Creature.stopped = true;
				m_pEventPortal->OnRPGEvent( newEvt );
				moving_forward = false;
			}
			if (moving_backwards){
				printf("!!!STOPPED!!!");
				SRPG_Event newEvt;
				newEvt.EventType = RPG_EVT_CREATURE;
				newEvt.Creature.CreatureEventType =
RPG_CEVT_MV_BACKWARD;
				newEvt.Creature.cid = m_body->getID();
				newEvt.Creature.stopped = true;
				m_pEventPortal->OnRPGEvent( newEvt );
				moving_backwards = false;
			}
			if (turning_right){
				printf("!!!STOPPED!!!");
				SRPG_Event newEvt;
				newEvt.EventType = RPG_EVT_CREATURE;
				newEvt.Creature.CreatureEventType =
RPG_CEVT_MV_RIGHT;
				newEvt.Creature.cid = m_body->getID();
				newEvt.Creature.stopped = true;
				m_pEventPortal->OnRPGEvent( newEvt );
				turning_right = false;
			}
			if (turning_left){
				printf("!!!STOPPED!!!");
				SRPG_Event newEvt;
				newEvt.EventType = RPG_EVT_CREATURE;
				newEvt.Creature.CreatureEventType =
RPG_CEVT_MV_LEFT;
```

```
                        newEvt.Creature.cid = m_body->getID();
                        newEvt.Creature.stopped = true;
                        m_pEventPortal->OnRPGEvent( newEvt );
                        turning_left = false;
                  }
                  if(enemySpoted(hit)){
                        setState(AIS_FIGHT);
                        printState();
                        goto exit;
                  }
                  if(goodHealth()){
                        setState(AIS_WANDER);
                        printState();
                        goto exit;
                  }else{
                        if(!lowHealth()){
                              //Temporary, until caution state
                              setState(AIS_HEAL);
                              printState();
                              goto exit;
                        }else{
                              setState(AIS_HEAL);
                              printState();
                              goto exit;
                        }
                  }

            case AIS_WANDER:
                  if(enemySpoted(hit)){
                        m_lastKnownPos = position;
                        setState(AIS_FIGHT);
                        if(stuck){
                              SRPG_Event newEvt;
                              newEvt.EventType = RPG_EVT_CREATURE;
                              newEvt.Creature.CreatureEventType =
RPG_CEVT_MV_RIGHT;
                              newEvt.Creature.cid = m_body->getID();
                              newEvt.Creature.stopped = true;
                              m_pEventPortal->OnRPGEvent( newEvt );
                              stuck = false;
                        }
                        printState();
                        goto exit;
                  }
                  if(goodHealth()){
                        if(!ai_in_progress){

      //setDestination(irr::core::vector3df(34.081306f,74.571106f,-
58.961212f)); //any random destination for now
                              setDestination(m_lastKnownPos);
                              //printf("[%f,%f,%f]\n",m_lastKnownPos.X,
m_lastKnownPos.Y, m_lastKnownPos.Z);
                              ai_in_progress = true;
                        }
                        if (!stuck) {
                              moveTorwardDestination();
                        }
                        if(m_timeOut == 0){
                              if(is_stuck()){
                                    printf("stuck!");
                                    setState(AIS_IDLE);
```

```
                                    printState();
                                    /*SRPG_Event newEvt;
                                    newEvt.EventType = RPG_EVT_CREATURE;
                                    newEvt.Creature.CreatureEventType =
RPG_CEVT_MV_FORWARD;
                                    newEvt.Creature.cid = m_body->getID();
                                    newEvt.Creature.stopped = true;
                                    m_pEventPortal->OnRPGEvent( newEvt );
                                    moving_forward = false;

                                    newEvt.EventType = RPG_EVT_CREATURE;
                                    newEvt.Creature.CreatureEventType =
RPG_CEVT_MV_RIGHT;
                                    newEvt.Creature.cid = m_body->getID();
                                    newEvt.Creature.stopped = false;
                                    m_pEventPortal->OnRPGEvent( newEvt );
                                    stuck = true;*/
                                    m_body->get3DModel()->getNode()-
>setPosition(irr::core::vector3df(34.081306f,0.571106f,-58.961212f));
                              }
                        }
                        goto exit;
                  }else{
                        if(!lowHealth()){
                              //Temporary, until caution state
                              setState(AIS_IDLE);
                              printState();
                              goto exit;
                        }else{
                              setState(AIS_IDLE);
                              printState();
                              goto exit;
                        }
                  }

            case AIS_FIGHT:
                  if (moving_backwards){
                        SRPG_Event newEvt;
                        newEvt.EventType = RPG_EVT_CREATURE;
                        newEvt.Creature.CreatureEventType =
RPG_CEVT_MV_BACKWARD;
                        newEvt.Creature.cid = m_body->getID();
                        newEvt.Creature.stopped = true;
                        m_pEventPortal->OnRPGEvent( newEvt );
                        moving_backwards = false;
                  }
                  if(enemySpoted(hit)){
                        m_lastKnownPos = position;
                        //printf("[%f,%f,%f]\n",position.X, position.Y,
position.Z);
                        if(goodHealth()){
                              if(enemyIsTooFar(position)){
                                    //move torwards the enemy (persuit)
                                    setDestination(position);
                                    moveTorwardDestination();
                                    goto exit;
                              }else{//Enemy In range
                                    //attack
                                    if (moving_forward){
                                          SRPG_Event newEvt;
```

```
                                          newEvt.EventType =
RPG_EVT_CREATURE;
                                          newEvt.Creature.CreatureEventType
= RPG_CEVT_MV_FORWARD;
                                          newEvt.Creature.cid = m_body-
>getID();
                                          newEvt.Creature.stopped = true;
                                          m_pEventPortal-
>OnRPGEvent( newEvt );
                                          moving_forward = false;
                                    }
                                    attack(hit);
                                    goto exit;
                              }
                        }else{
                              if(!lowHealth()){
                                    //Temporary, until caution state

                                    //Taken out for testing
                                    if(enemyIsTooFar(position)){
                                          //move torwards the enemy
                                          setDestination(position);
                                          moveTorwardDestination();
                                          goto exit;
                                    }else{//Enemy In range
                                          //attack
                                          attack(hit);
                                          goto exit;
                                    }
                              }else{ //if lowHealth is true.
                                    setState(AIS_FLEE);
                                    printState();
                                    goto exit;
                              }
                        }
                  }
                  //Enemy not seen
                  setState(AIS_IDLE);
                  printState();
                  goto exit;

            case AIS_FLEE:
                  if (moving_forward){
                        SRPG_Event newEvt;
                        newEvt.EventType = RPG_EVT_CREATURE;
                        newEvt.Creature.CreatureEventType =
RPG_CEVT_MV_FORWARD;
                        newEvt.Creature.cid = m_body->getID();
                        newEvt.Creature.stopped = true;
                        m_pEventPortal->OnRPGEvent( newEvt );
                        moving_forward = false;
                  }
                  if(enemySpoted(hit)){
                        m_lastKnownPos = position;
                        if(goodHealth()){
                              setState(AIS_FIGHT);
                              printState();
                              goto exit;
                        }else{
                              if(!lowHealth()){
                                    //Temporary, until caution state
```

```
                                    setState(AIS_FIGHT);
                                    printState();
                                    goto exit;
                              }else{
                                    //move away from the enemy
                                    setDestination(position);
                                    moveAwayFromDestination();
                                    goto exit;
                              }
                        }
                  }//Enemy not seen
                  setState(AIS_IDLE);
                  printState();
                  goto exit;

            case AIS_HEAL:
                  if(enemySpoted(hit)){
                        m_lastKnownPos = position;
                        setState(AIS_FIGHT);
                        printState();
                        goto exit;
                  }
                  if(goodHealth()){
                        setState(AIS_IDLE);
                        printState();
                        goto exit;
                  }else{
                        if(!lowHealth()){
                              //Temporary, until caution state
                              //healing code
                              heal();
                              goto exit;
                        }else{
                              //healing code
                              heal();
                              goto exit;
                        }
                  }

            default:
                  //INVALID STATE
                  m_myLastPos = m_body->get3DModel()->getFrontPoint();
                  return false;
            }
exit:   if(m_timeActions == 0){
                  m_timeActions = m_maxTimeActions;
                  m_myLastPos = m_body->get3DModel()->getFrontPoint();
            }
            return true;
}
```

7.2.3 – AI_Controller.h

```
// ai_controller.h: interface for the AI_Controller class.
//
//////////////////////////////////////////////////////////////////////

#ifndef AI_CONTROLLER_H
#define AI_CONTROLLER_H

#if _MSC_VER > 1000
#pragma once
```

```
#endif // _MSC_VER > 1000

//#define PI 3.14
#include <stdio.h>
#include "ai_states.h"
#include "../Entities/rpg_creature.h"
#include "../sRPG_Events.h"
#include "../Base_Classes/rpg_eventhandler.h"

class AI_Controller
{
protected:
	irr::s32 m_goodHP;
	irr::s32 m_lowHP;
	irr::s32 m_healingFactor;
	tAI_STATE m_curState;
	bool ai_in_progress;

	irr::u32 m_maxTimeBetweenAttacks;
	irr::u32 m_timeBetweenAttacks;
	irr::u32 m_maxTimeActions;
	irr::u32 m_timeActions;
	irr::u32 m_maxTimeOut;
	irr::u32 m_timeOut;

	RPG_ID_TYPE m_enemyId;

	irr::u32 m_visionArc;
	irr::f64 m_minDistance;

	irr::core::vector3df m_tgtDestination; //destination point to walk to
	irr::core::vector3df m_lastKnownPos; //Last Known Position of the
Target.
	irr::core::vector3df m_myLastPos; //My Last Postion.

	RPG_Creature* m_body;
	rpg_EventHandler* m_pEventPortal;


public:
	AI_Controller(RPG_Creature* body, rpg_EventHandler* eventPortal);
	virtual ~AI_Controller();

	bool moveTorwardDestination(); //in case of wandering and fighting
	bool moveAwayFromDestination(); //in case of fleeing
	void attack(RPG_ID_TYPE enemySpoted); //attack

	bool think(RPG_ID_TYPE hit, irr::core::vector3df position);

	bool turning_right;
	bool turning_left;
	bool moving_forward;
	bool moving_backwards;
	bool stuck;

	//GETTING/SETTING Functions

		//Set the enemy Id, lookout purpose
		void setEnemyId(irr::u32 enemyId) { m_enemyId = enemyId;}

		//Set the new state of the creature's brain
```

```
            void setState(tAI_STATE newState) { m_curState = newState;}

            void setDestination(irr::core::vector3df Destination)
{m_tgtDestination = Destination;}

            tAI_STATE getCurrentState() { return m_curState; }
            RPG_ID_TYPE getEnemyId() { return m_enemyId; }
            irr::u32* getVisionArc() { return &m_visionArc; }

            RPG_Creature* getBody() {return m_body; }
private:
      irr::s32* getGoodHP() { return &m_goodHP; }
      irr::s32* getLowHP() { return &m_lowHP; }

      bool goodHealth() {     return *m_body->getHP() > *getGoodHP();}
      bool lowHealth(){ return *m_body->getHP() < *getLowHP();}

      bool enemyIsTooFar(irr::core::vector3df position){
            irr::core::vector3df l_size =
irr::core::vector3df(1000.0f,0.0f,0.0f);
            irr::core::vector3df l_start = m_body->get3DModel()-
>getFrontPoint();
            irr::core::vector3df l_end = l_start + m_body->get3DModel()-
>getFacing() + l_size;
            irr::f64 dist;

            l_start = m_body->get3DModel()->getFrontPoint();
            dist = l_start.getDistanceFrom(position);
            //printf("DISTANCE [%f]\n",dist);
            if((position.X > l_start.X) &&
               (position.Z <= l_start.Z + 2.0f) && (position.Z >= l_start.Z
- 2.0f) &&
               (position.Z <= l_end.Z + 2.0f) && (position.Z >= l_end.Z -
2.0f)){
                  if (dist < m_minDistance) {
                        //printf("NOT DISTANT\n");
                        return false;
                  }
            }
            //printf("DISTANT\n");
            return true;

      }

      bool enemySpoted(RPG_ID_TYPE hit){
            if(hit<LVLMGR_MAX_ENTS){return true;}else{return false;}
      }

      void heal(){
            if(m_timeBetweenAttacks==0){
                  m_body->setHP(*m_body->getHP()+m_healingFactor);
            }
      };

//Utilitiy
      bool is_stuck(){
            irr::core::vector3df current_pos = m_body->get3DModel()-
>getFrontPoint();


      printf("CURR[%f,%f,%f]\nLAST[%f,%f,%f]",floor(current_pos.X),floor(cu
```

```
rrent_pos.Y),floor(current_pos.Z),floor(m_myLastPos.X),floor(m_myLastPos.Y)
,floor(m_myLastPos.Z));

            if(((floor(current_pos.X) <= floor(m_myLastPos.X +3)) &&
(floor(current_pos.X) >= floor(m_myLastPos.X -3))) &&
                ((floor(current_pos.Y) <= floor(m_myLastPos.Y +3)) &&
(floor(current_pos.Y) >= floor(m_myLastPos.Y -3))) &&
                ((floor(current_pos.Z) <= floor(m_myLastPos.Z +3)) &&
(floor(current_pos.Z) >= floor(m_myLastPos.Z -3))) ){
                    printf("STUCK!\n");
                    return true;
            }
            printf("NOT STUCK!\n");
            return false;
      }

      void updateCounter(){
            if(m_timeBetweenAttacks==0){
                  m_timeBetweenAttacks = m_maxTimeBetweenAttacks;
                  SRPG_Event attackEvt;
                  attackEvt.EventType = RPG_EVT_CREATURE;
                  attackEvt.Creature.CreatureEventType = RPG_CEVT_ATTACK;
                  attackEvt.Creature.cid = m_body->getID();
                  attackEvt.Creature.stopped = true;
                  m_pEventPortal->OnRPGEvent( attackEvt );
            }else{
                  m_timeBetweenAttacks -= 1;
            }
            if(m_timeActions == 0){
                  m_timeActions = m_maxTimeActions;
            }else{
                  m_timeActions -= 1;
            }
            if(m_timeOut == 0){
                  m_timeOut = m_maxTimeOut;
            }else{
                  m_timeOut -= 1;
            }
      }

      void printState(){
            switch(m_curState){
                  case AIS_IDLE:
                        printf("CURRENT STATE IS: IDLE\n");break;
                  case AIS_WANDER:
                        printf("CURRENT STATE IS: WANDER\n");break;
                  case AIS_FIGHT:
                        printf("CURRENT STATE IS: FIGHT\n");break;
                  case AIS_FLEE:
                        printf("CURRENT STATE IS: FLEE\n");break;
                  case AIS_HEAL:
                        printf("CURRENT STATE IS: HEAL\n");break;
            }
      }
};

#endif // AI_CONTROLLER_H
```

7.2.4 – LevelManager.h

```
// Class automatically generated by Dev-C++ New Class wizard

#ifndef LEVELMANAGER_H
#define LEVELMANAGER_H

#include <irrlicht.h>

#include <string>
#include <map>

#include "Base_Classes/RPG_Types.h"
#include "sRPG_Events.h"
#include "Base_Classes/rpg_worldobject.h"
#include "Base_Classes/rpg_eventhandler.h"


#include "../Options/ICE_Options.h"

//graphics
#include "../Graphics/modelcontroller.h"
#include "../Graphics/followingcamera.h"
#include "../Graphics/tempent_factory.h"

//entities
#include "Entities/rpg_entityfactory.h"

//items
#include "Items/itemManager.h"

//AI
#include "AI/ai_controller.h"

/*
//struct needed to compare hash keys for std::map<RPG_ID_TYPE,
modelController *, ltcid>
struct ltcid
{
  bool operator()(RPG_ID_TYPE id1, RPG_ID_TYPE id2) const
  {
    return id1 < id2 ? true : false;
  }
};
*/

enum eLM_Cameras {
      ELM_CAM_3RDP = 0,
      ELM_CAM_1RSTP,

      ELM_CAM_COUNT
};

/*
 * manages the level, loading and unloading needed resources, etc
 * note: contains implementation-specific code
 */
class levelManager : public rpg_EventHandler
{
    RPG_DEVICE_HANDLE m_pDevice;

      rpg_EventHandler* m_pEventPortal;
```

```
    //** level structures **
    irr::scene::IAnimatedMesh*  m_lvlMesh;
    irr::scene::ISceneNode*     m_lvlNode;
    irr::scene::ITriangleSelector* m_lvlTriSel;

    //** entities (items, creatures) list **
      //std::map<std::string, std::string> m_meshNames; //todo: use this
    //std::map<RPG_ID_TYPE, modelController *, ltcid> m_entities;
      //todo: use this
    RPG_WorldObject* m_entities[LVLMGR_MAX_ENTS];

      //To be associated to an entity
      AI_Controller* m_aiControllers[LVLMGR_MAX_ENTS];

      itemManager *m_itemMgr;

      //** player info **
      RPG_ID_TYPE m_avatarID;

    //** cameras **
      eLM_Cameras m_camMode;
    followingCamera *           m_3dpCam; //3rd person cam
    //TODO: add 1rst person camera and allow user to switch to it

      public:
            // class constructor
            levelManager(RPG_DEVICE_HANDLE device, rpg_EventHandler*
eventPortal);
            // class destructor
            ~levelManager();

//TODO: remove hardcode
        void Init();

            bool OnRPGEvent(SRPG_Event event);
            void Update(RPG_TIME_TYPE currTime);

            //TODO: dont hardcode this
            RPG_ID_TYPE getPlayerID() { return m_avatarID; }
            void setPlayerID(RPG_ID_TYPE id) { m_avatarID = id; }

      private:


          bool removeEntity(RPG_ID_TYPE id);

          RPG_ID_TYPE rayToCreature(irr::core::line3d<irr::f32> ray);
      //Spotting the creature within the line of sight
          RPG_ID_TYPE spotCreature(irr::core::vector3df front,
      irr::core::vector3df facing, irr::u32* arc, RPG_ID_TYPE enemy);
};

#endif // LEVELMANAGER_H
```

7.2.5 – LevelManager.cpp

```
// Class automatically generated by Dev-C++ New Class wizard

#include "levelmanager.h" // class's header file
```

```
void loadingScreen(irr::s32 percent, RPG_DEVICE_HANDLE device, wchar_t*
text) {
    irr::video::IVideoDriver* driver = device->getVideoDriver();
    irr::gui::IGUIEnvironment* env = device->getGUIEnvironment();

      static irr::video::ITexture * loadbar = NULL;
      if(!loadbar) loadbar =  driver->getTexture("data/progressbar.BMP");

      driver->beginScene(true, false, irr::video::SColor(0,0,0,0));

      irr::gui::IGUIFont* font = env->getBuiltInFont();

      font->draw(text, irr::core::rect<irr::s32>(50, 100, 50 + 200, 150),
irr::video::SColor(255,255,255,255));

      irr::core::rect< irr::s32 > clip(50, 50, 50 + (256 * percent/100) ,
50 + 32);

       if(loadbar) driver->draw2DImage(loadbar,
        irr::core::position2d< irr::s32 >(50,50),
        irr::core::rect< irr::s32 >(0,0,256,32),
        &clip,
        irr::video::SColor(255, 255, 255, 255),
        false
        );

    driver->endScene();

}

// class constructor
levelManager::levelManager(RPG_DEVICE_HANDLE device, rpg_EventHandler*
eventPortal)
{
    m_pDevice = device;
    m_pEventPortal = eventPortal;

    for(int i=0; i<LVLMGR_MAX_ENTS; i++) {
        m_entities[i] = 0;
    }
      for(int j=0; j<LVLMGR_MAX_ENTS; j++) {
        m_aiControllers[j] = 0;
    }

      m_itemMgr = new itemManager(eventPortal);
}

// class destructor
levelManager::~levelManager()
{
//drop handles
    m_lvlTriSel->drop();
    //m_lvlMesh  //didnt create, so dont drop

//remove nodes
    m_lvlNode->remove();

//delete creations

    delete m_3dpCam;
```

```
    for(int i=0; i<LVLMGR_MAX_ENTS; i++)
        if(m_entities[i]) delete m_entities[i];


      //irr::scene::ISceneManager* smgr = m_pDevice->getSceneManager();
      //smgr->removeAllNodes();

      m_pDevice = NULL; //dont delete a pointer we dont own
}

void levelManager::Init() {
    RPG_DEVICE_HANDLE device = m_pDevice;
    irr::scene::ISceneManager* smgr = m_pDevice->getSceneManager();
    irr::video::IVideoDriver* driver = m_pDevice->getVideoDriver();
      irr::u32    l_id;

loadingScreen(0, device, L"LOADING: item templates");
    m_itemMgr->FillTemplateRegistry();

loadingScreen(10, device, L"LOADING: level mesh");
if(ICE_Options::getPref("lighting") == "T") {
    smgr->addLightSceneNode(0, irr::core::vector3df(0,100,0),
            irr::video::SColorf(1.0f, 0.6f, 0.7f, 1.0f), 600.0f);
}

    m_lvlMesh = smgr->getMesh("data/media/room.3ds");

loadingScreen(20, device, L"LOADING: mapping level");
    smgr->getMeshManipulator()->makePlanarTextureMapping( m_lvlMesh-
>getMesh(0) , 0.008f);
    m_lvlNode = smgr->addAnimatedMeshSceneNode(m_lvlMesh);

loadingScreen(30, device, L"LOADING: level texture");
if(ICE_Options::getPref("lighting") != "T") {
    m_lvlNode->setMaterialFlag(irr::video::EMF_LIGHTING, false);
}
      m_lvlNode->setMaterialTexture(0, driver-
>getTexture("data/media/wall.jpg") );

loadingScreen(40, device, L"LOADING: creating OctTreeTriangleSelector");
      m_lvlTriSel = smgr->createOctTreeTriangleSelector(m_lvlMesh-
>getMesh(0), m_lvlNode);
      //smgr->setShadowColor(irr::video::SColor(220,0,0,0));
if(!m_lvlTriSel) printf("AAAAHHH, IM GOING TO EXPLODE!\n");

loadingScreen(50, device, L"LOADING: creature 1");
      l_id = RPG_entityFactory::makeCreature(m_entities, m_pDevice,
m_pEventPortal, m_lvlTriSel, "data/media/faerie.md2",
"data/media/faerie2.bmp");
      //Attaching AI Controller
      m_aiControllers[l_id] = new
AI_Controller((RPG_Creature*)m_entities[l_id],m_pEventPortal);

loadingScreen(60, device, L"LOADING: creature 2");
    //m_avatarID = RPG_entityFactory::makeCreature(m_entities, m_pDevice,
m_pEventPortal, m_lvlTriSel, "data/media/faerie.md2",
"data/media/faerie5.bmp");
      m_avatarID = RPG_entityFactory::makeCreature(m_entities, m_pDevice,
m_pEventPortal, m_lvlTriSel, "data/media/tris.md2",
"data/media/ctf_b.bmp");
```

```
      m_aiControllers[l_id]->setEnemyId(m_avatarID);
loadingScreen(70, device, L"LOADING: bonfire");
      RPG_entityFactory::makeBonfire( m_entities, m_pDevice,
m_pEventPortal);

loadingScreen(80, device, L"LOADING: camera");
      m_3dpCam = new
followingCamera( ((RPG_Creature*)m_entities[m_avatarID])->get3DModel()-
>getNode() , smgr);
      smgr->setActiveCamera( m_3dpCam->getCam() );
loadingScreen(100, device, L"LOADING: DONE!");


      //RPG_entityFactory::makeItemVessel( m_entities, m_pDevice,
m_pEventPortal, m_lvlTriSel, "data/media/faerie.md2",
"data/media/faerie5.bmp");
}

bool levelManager::removeEntity(RPG_ID_TYPE id) {
    if(id >= LVLMGR_MAX_ENTS) return false;
    if(!m_entities[id]) return false;
      //Added ai removal code
      if(m_aiControllers[id]){
            //Dettach ai_controller
            delete m_aiControllers[id];
            m_aiControllers[id] = NULL;
      }

      //TODO: de-select camera?
      if(id == m_avatarID ) m_3dpCam->setTarget(NULL);

    //remove creature's TriSelector from other object's MetaTriSelectors
    for(RPG_ID_TYPE i=0; i<LVLMGR_MAX_ENTS; i++) {
        if(!m_entities[i] || i == id) continue;

        m_entities[i]->removeTriSelector( m_entities[id]->getBBTriSel() );

    }

    eRPG_ObjTypes entityType = m_entities[id]->getObjType();

    delete m_entities[id];
    m_entities[id] = NULL;

    //*
    if(entityType == ERPG_OTYPE_CREATURE) {
        //let GUI know a creature is removed (GUI will remove HP bar)
        SRPG_Event newEvt;
      newEvt.EventType = RPG_EVT_GUI;
      newEvt.GUI.GUIEventType = RPG_IEVT_REMOVECREATURE;
      newEvt.GUI.cid = id;
      m_pEventPortal->OnRPGEvent( newEvt ) ;
      }
   //*/

    return true;
}



#include <stdio.h>
```

```
bool levelManager::OnRPGEvent(SRPG_Event event) {
      irr::u32 l_id;

    //handle irrlicht devices directly
    // used for stuff like user-preference keys (toggle HP bars, toggle map
display, etc)
/*
    if(event.EventType == RPG_EVT_DEVICE) {
        irr::SEvent devEvt =  (event.Device.Event) ;
        if( devEvt.EventType == irr::EET_KEY_INPUT_EVENT &&
devEvt.KeyInput.PressedDown) {
          if(devEvt.KeyInput.Key == ICE_Options::getKey("toggle_HPBar") ) {
            for(int i=0; i<LVLMGR_MAX_ENTS; i++) {
              if( m_entities[i] && m_entities[i]->getObjType() ==
ERPG_OTYPE_CREATURE)
                  ((RPG_Creature*) m_entities[i])->toggleHPVisible();
            }
          }
        }
    }
*/

      if(event.EventType == RPG_EVT_PLAYER ){
            if(event.Player.PlayerEventType == RPG_PEVT_BIRTH) {
                 //m_avatarID =  RPG_entityFactory::makeCreature(m_entities,
m_pDevice, m_pEventPortal, m_lvlTriSel, "data/media/faerie.md2",
"data/media/faerie5.bmp");
                  m_avatarID = RPG_entityFactory::makeCreature(m_entities,
m_pDevice, m_pEventPortal, m_lvlTriSel, "data/media/tris.md2",
"data/media/ctf_b.bmp");
                  m_3dpCam-
>setTarget( ((RPG_Creature*)m_entities[m_avatarID])->get3DModel()-
>getNode() );
                  return true;
            }
      }

      if(event.EventType == RPG_EVT_SERVER && event.Server.ServerEventType
== RPG_SEVT_WORLDUPDATE) {
            //pass to all entities
            for(int i=0; i<LVLMGR_MAX_ENTS; i++) {
                        if( m_entities[i] ) m_entities[i]-
>OnRPGEvent(event);
            }
            return true;
      }

    if(event.EventType == RPG_EVT_CREATURE &&
(event.Creature.CreatureEventType == RPG_CEVT_BIRTH ||
m_entities[event.Creature.cid]) ) {

      switch(event.Creature.CreatureEventType) {
        case RPG_CEVT_ATTACK:
            if(m_entities[event.Creature.cid]->OnRPGEvent(event))  { //if
can attack
                        if(event.Creature.stopped) return true;
                              //create flame from attack
                              TempEnt_Factory::makeFlame(m_pDevice,
                         ((RPG_Creature*)m_entities[event.Creature.cid])-
>get3DModel()->getFrontPoint(),
```

```
                        ((RPG_Creature*)m_entities[event.Creature.cid])-
>get3DModel()->getFacing() );

                              //check for ray-entity collision
                              RPG_ID_TYPE hit = rayToCreature(

      irr::core::line3d<irr::f32>(

      ((RPG_Creature*)m_entities[event.Creature.cid])->get3DModel()-
>getFrontPoint(),

      ((RPG_Creature*)m_entities[event.Creature.cid])->get3DModel()-
>getFrontPoint() + ((RPG_Creature*)m_entities[event.Creature.cid])-
>get3DModel()->getFacing() )
                                                      );

                              if (hit < LVLMGR_MAX_ENTS) {
                                    if(m_entities[hit]) {
                                    SRPG_Event newEvt;
                                    newEvt.EventType = RPG_EVT_CREATURE;
                                    newEvt.Creature.CreatureEventType =
RPG_CEVT_TAKEDMG;
                                    newEvt.Creature.cid = hit;
                                    newEvt.Creature.fromid =
event.Creature.cid;

                                    //TODO: add event to an event QUE
instead of directly processing
                                    if( m_pEventPortal-
>OnRPGEvent( newEvt ) ) return true;
                                    }
                              }

           }else return false; //end if: can attack
        break; //end case: RPG_CEVT_ATTACK

            case RPG_CEVT_TAKEDMG:
                        if( m_entities[event.Creature.cid]-
>OnRPGEvent(event) ) return true;
            break; //end case: RPG_CEVT_TAKEDMG

            case RPG_CEVT_DEATH:
                  if( m_entities[event.Creature.cid] ) this-
>removeEntity( event.Creature.cid ) ;
            break; //end case: RPG_CEVT_DEATH

            case RPG_CEVT_BIRTH:
                  l_id = RPG_entityFactory::makeCreature(m_entities,
m_pDevice, m_pEventPortal, m_lvlTriSel,  "data/media/faerie.md2",
"data/media/faerie2.bmp" ) ;
                  m_aiControllers[l_id] = new
AI_Controller((RPG_Creature*)m_entities[l_id],m_pEventPortal);
                  m_aiControllers[l_id]->setEnemyId(m_avatarID);
            break; //end case: RPG_CEVT_BIRTH

            default:
            //handle other (move) events here
            if(event.Creature.cid < LVLMGR_MAX_ENTS)  { //if cID is valid
range
              if(m_entities[event.Creature.cid]) { //if creature of cID
exists
```

```
                    if(m_entities[event.Creature.cid]->OnRPGEvent(event) )
return true; //pass event
                }
             }
      }//end switch

    }//end if: RPG_EVT_CREATURE
    return false;
}

RPG_ID_TYPE levelManager::rayToCreature(irr::core::line3d<irr::f32> ray) {
    irr::scene::ISceneNode *node =
        m_pDevice->getSceneManager()->getSceneCollisionManager()-
>getSceneNodeFromRayBB( ray, ERPG_FLAG_CREATURE  );

    if(!node) return 99999; //invalid creature id

//    printf("rToCreature: hit id [%d]\n", node->getID() );

      for(int i=0; i<LVLMGR_MAX_ENTS; i++) {
        if(! m_entities[i] || m_entities[i]->getObjType() !=
ERPG_OTYPE_CREATURE ) continue;
        if( node == ((RPG_Creature*)m_entities[i])->get3DModel()-
>getNode() ) return i;
    }

    return 99999; //invalid creature id
}
//Enemy Visible?
RPG_ID_TYPE levelManager::spotCreature(irr::core::vector3df front,
irr::core::vector3df facing, irr::u32* arc, RPG_ID_TYPE enemy) {
      //The ray will be the length of field of view
      RPG_ID_TYPE hit;
      irr::core::vector3df l_size = irr::core::vector3df(1000,0,0);
      irr::core::vector3df l_start = front;
      irr::core::vector3df l_end = front + facing + l_size;
      irr::s32 signal = 1;
      //printf("ARC [%d]",*arc);
      for(int i = 1; i <= 2*(*arc); i++){
            hit =
rayToCreature(irr::core::line3d<irr::f32>(l_start,l_end));
            if(hit==enemy){

      //printf("SPOTTED![START(%d,%d,%d)|END(%d,%d,%d)]\n",(int)l_start.X,(
int)l_start.Y,(int)l_start.X,(int)l_end.X, (int)l_end.Y, (int)l_end.Z);
                  return hit;
            }else{
                  //printf("RotateBy [%d]\n",signal*i);
                  signal *= -1;
                  l_end.rotateXZBy(signal*i,l_start);
            }
      }
      return 99999;     //invalid creature id
}

void levelManager::Update(RPG_TIME_TYPE currTime) {
      RPG_ID_TYPE hit = 99999;      //invalid creature id
      irr::core::vector3df hit_pos;
    m_3dpCam->Update(currTime);
      for(int i=0; i<LVLMGR_MAX_ENTS; i++) {
```

```
            if(m_aiControllers[i]){
                  hit = spotCreature(((RPG_Creature*)m_entities[i])-
>get3DModel()->getFrontPoint(),
                                                ((RPG_Creature*)m_entities[i])-
>get3DModel()->getFacing(),
                                                m_aiControllers[i]-
>getVisionArc(),
                                                m_aiControllers[i]-
>getEnemyId());
                  if(hit < LVLMGR_MAX_ENTS){hit_pos =
((RPG_Creature*)m_entities[hit])->get3DModel()->getNode()->getPosition();}
                  m_aiControllers[i]->think(hit,hit_pos);
            }
      }
}
```

7.2.6 – RPG_Creature.h

```
// Class automatically generated by Dev-C++ New Class wizard

#ifndef RPG_CREATURE_H
#define RPG_CREATURE_H

#include "../Base_Classes/rpg_worldobject.h" // inheriting class's header
file

#include "../../Graphics/modelcontroller.h"

/*
 * monsters, player avatars, NPCs, etc
 */
class RPG_Creature : public RPG_WorldObject
{
    //stats
    irr::s32 m_maxHP;
      irr::s32 m_hp;

      //inventory

      //device-dependant: visuals
      RPG_DEVICE_MODEL m_avatar;


      public:
            // class constructor
            RPG_Creature(irr::scene::IAnimatedMesh* mesh,
irr::video::SMaterial* material, RPG_DEVICE_HANDLE device,
rpg_EventHandler* eventPortal);
            // class destructor
            virtual ~RPG_Creature();

    virtual irr::scene::ITriangleSelector* getBBTriSel() { return m_avatar-
>getBBTriSel(); }
    virtual void setCollisionDetection( irr::scene::IMetaTriangleSelector*
mtris ) { m_avatar->setCollisionDetection(mtris); }
    virtual void addTriSelector( irr::scene::ITriangleSelector * trisel )
{ m_avatar->addTriSelector(trisel); }
    virtual void removeTriSelector( irr::scene::ITriangleSelector *
trisel ) { m_avatar->removeTriSelector(trisel); }

      virtual bool OnRPGEvent(SRPG_Event event);
```

```
      //device specific implementations:
      virtual void set3DPos(RPG_3D_POS_TYPE newPos) { m_avatar->getNode()-
>setPosition(newPos); }
      virtual RPG_3D_POS_TYPE get3DPos() { return m_avatar->getNode()-
>getPosition(); }
      virtual RPG_DEVICE_MODEL get3DModel() { return m_avatar; }

      //todo: add code to change visibilty of representations
      virtual void setVisible( RPG_BOOL vis ) {m_visible = vis; m_avatar-
>getNode()->setVisible(vis);}


            irr::s32* getHP() { return &m_hp; }
            irr::s32* getMaxHP() { return &m_maxHP; }

            void setHP(irr::s32 newHP);

};

#endif // RPG_CREATURE_H
```

7.2.7 – RPG_Creature.cpp

```
// Class automatically generated by Dev-C++ New Class wizard

#include "rpg_creature.h" // class's header file

// class constructor
RPG_Creature::RPG_Creature(irr::scene::IAnimatedMesh* mesh,
irr::video::SMaterial* material, RPG_DEVICE_HANDLE device,
rpg_EventHandler* eventPortal) : RPG_WorldObject(eventPortal)
{
      m_objType = ERPG_OTYPE_CREATURE;
      m_hp = 100;
      m_maxHP = 100;

      irr::scene::ISceneManager* smgr = device->getSceneManager();
      irr::gui::IGUIEnvironment * env = device->getGUIEnvironment();

      m_avatar = new modelController(mesh, material, smgr);

      m_avatar->getNode()->setID( ERPG_FLAG_CREATURE );

}

// class destructor
RPG_Creature::~RPG_Creature()
{
    //remove avatar model
    delete m_avatar;
}

bool RPG_Creature::OnRPGEvent(SRPG_Event event) {

    if(event.EventType == RPG_EVT_CREATURE &&
        event.Creature.CreatureEventType == RPG_CEVT_TAKEDMG) {

        m_hp -= 10;

        //if I died, tell the world about it
```

```
            if(m_hp <= 0) {
                  SRPG_Event newEvt;
                  newEvt.EventType = RPG_EVT_CREATURE;
                  newEvt.Creature.CreatureEventType = RPG_CEVT_DEATH;
                  newEvt.Creature.cid = m_id;
                  newEvt.Creature.fromid = event.Creature.fromid;

                  //TODO: dont call onRPGEvent directly-- add to a msg QUE
instead
                  m_pEventPortal->OnRPGEvent( newEvt );
            }

        return true;
    }

    //otherwise, send event to model controller
    if( event.EventType == RPG_EVT_CREATURE &&  m_avatar->OnEvent(event) )
return true;

    return false;
}

/*
void RPG_Creature::setHPVisible(RPG_BOOL value) {
    m_HPcounter->setVisible(value);
}

void RPG_Creature::toggleHPVisible() {
    if(m_HPcounter->isVisible() )
        setHPVisible(false);
    else
        setHPVisible(true);
}
*/

void RPG_Creature::setHP(irr::s32 newHP){
      if(newHP<m_maxHP){
            m_hp = newHP;
      }else{
            m_hp = m_maxHP;
      }

      if(m_hp <= 0) {
            SRPG_Event newEvt;
            newEvt.EventType = RPG_EVT_CREATURE;
            newEvt.Creature.CreatureEventType = RPG_CEVT_DEATH;
            newEvt.Creature.cid = m_id;
            newEvt.Creature.fromid = m_id;
            //newEvt.Creature.fromid = event.Creature.fromid;

            //TODO: dont call onRPGEvent directly-- add to a msg QUE
instead
            m_pEventPortal->OnRPGEvent( newEvt );
      }
}
```