Planejamento aplicado a jogos de computador: uma implementação baseada em Redes de Tarefas Hierárquicas

Marcos A. Castilho[1], **Luis Allan Künzle**[1], **Silvio A. Porto**[1], **Fabiano Silva**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19081 – 81531-980 – Curitiba – PR – Brasil

castilho@ufpr.br, kunzle@inf.ufpr.br, sap_st@yahoo.com.br, fabiano@inf.ufpr.br

Resumo. *Neste trabalho nós investigamos como planejadores são atualmente utilizados na área de jogos de computador. Analisamos os planejadores baseados em redes de tarefas hierárquicas e apresentamos uma nova implmentação que é adequada para este tipo de aplicação.*

Abstract. *In this work we investigate how planners are used nowadays in computer games. We analyse planners based on hierarquical tasks networks and show a new implementation which is adequate for this kind of aplication.*

1. Introdução

A inteligência artificial atualmente é considerada pelos desenvolvedores de jogos como parte essencial de um jogo moderno. Mas nem sempre foi assim. No princípio o tempo destinado para o desenvolvimento desta era bem limitado e a inteligência não passava de umas poucas regras ou ações baseadas em estatísticas. Estas regras ou ações geralmente eram escritas diretamente no código do jogo. Desta forma não havia como adicionar mais inteligência ao programa depois de compilado. Também havia escassez de recursos para executar os algoritmos de inteligência artificial devido a cada quadro da animação ter um tempo limite para ser montado, o que obrigava os desenvolvedores a sacrificar a IA em favor da computação gráfica.

Embora os desenvolvedores de jogos tenham usado técnicas melhores desde então, os agentes implementados em jogos são essencialmente reativos. Agentes reativos têm como características principais a facilidade em obter a próxima ação que deverá ser executada e o baixo custo de implementação e execução. Mas, entre os novos desafios, está tornar os agentes deliberativos, ou seja, inserir neles um mecanismo que lhes permita fazer uma análise mais complexa do mundo do que a feita pelos agentes reativos. Geralmente isto é conseguido com o uso de um planejador (Seção 2)

Entre os planejadores existentes, os mais adequados para esta aplicação são os semi-automáticos baseados em HTN. Dentre eles uma opção é SHOP2, que consiste de um sistema de planejamento baseado em redes de tarefas hierárquica que se diferencia dos demais planejadores HTN por fazer planejamento para frente, e assim simplifica o processo de planejamento permitindo um aumento considerável do poder de expressão.

O objetivo do nosso trabalho foi implementar o algoritmo de planejamento SHOP2 adaptando-o para que possa ser usado como parte do módulo deliberativo de um agente de jogo. A nossa implementação de SHOP2 tem a forma de uma biblioteca orientada a objetos. Desta forma é fácil para o programa usuário da biblioteca criar instâncias de

planejamento e gerencia-las usando *threads*. Assim, o planejador deve ser capaz de salvar o estado atual para que este seja retomado em um momento posterior. A especificação ou instanciação de uma arquitetura de agentes está fora do escopo deste trabalho.

Para a construção desta biblioteca foi observada a modularidade da mesma, bem como foram adotadas abordagens e ferramentas que permitam levar rapidamente o programa para outras plataformas. Basicamente a nossa implementação depende apenas de um compilador C++ com a biblioteca STL e um gerador de analisadores sintáticos para ser compilada. A linguagem C++ foi escolhida devido à sua disponibilidade nas mais diversas plataformas e a ser a mais difundida entre os desenvolvedores de jogos. Além disso, C++ permite criar bibliotecas dinâmicas que podem ser acessadas por executáveis feitos por outras linguagens, incluindo Java. Em nossos testes compilamos o programa tanto na plataforma Debian GNU/Linux quanto no Microsoft Windows.

2. Inteligência artificial em jogos.

Um dos objetivos da inteligência artificial (IA) é desenvolver programas capazes de imitar o raciocínio humano e assim reproduzir o seu comportamento. Isto tem sido conseguido com algum exito, em especial na área de jogos, onde existe a vantagem das ações executadas pelo agente não precisarem ser exatamente as mesmas que uma pessoa executaria, mas o comportamento necessita apenas ser coerente em relação ao do ser humano.

Isto possibilita ao desenvolvedor "trapacear" usando uma série de artifícios que dão ao usuário a falsa impressão de que os agentes são oponentes espertos. A trapaça é útil para simplificar a simulação a fim de economizar recursos. Entre as trapaças mais comuns estão o pré-processamento do cálculo da navegação, ataque massivo contra o usuário humano e configuração dos parâmetros de resistência e força dos agentes.

A maioria dos jogos atuais podem ser vistos como sistemas multi-agentes com ambientes dinâmicos. Agentes são entidades que "percebem seu ambiente através de sensores e agem sobre o mesmo utilizando atuadores" [Russell and Norvig 2003]. Os agentes também podem ser classificados quanto ao modo usado para gerar seu comportamento, sendo assim classificados como reativos, deliberativos ou híbridos [Hawes 2003].

Jogos podem ser vistos como uma maneira não linear de representar uma história, assim a cada decisão do jogador o fluxo da mesma toma um rumo diferente. Para que este fluxo seja contínuo é necessário observar a jogabilidade, ou seja, o grau de interatividade do jogo [Porto 2006]. Existem diversos tipos de jogos sendo que os mesmos podem basicamente ser divididos em de simulação e em de tabuleiro. Cada tipo tem vários sub tipos que apresentam diferentes requisitos, dentre os principais requisitos contrapostos estão a qualidade da ação contra o tempo de espera pela próxima jogada [Pedersen 2003].

O motor de IA pode ser visto, assim como o motor gráfico utilizado tradicionalmente pelos jogos, como uma coleção de técnicas não organizadas. Como a lógica de cada jogo deve ser construída com as ferramentas de IA que melhor se adequem aos subproblemas que devem ser resolvidos, então a escolha da ferramenta correta para cada subproblema específico é a chave para assegurar o bom desempenho do sistema por inteiro. O motor oferece uma implementação para as técnicas que devem ser utilizadas junto com as técnicas de gerência para atingir aos objetivos de um agente de jogo em particular.

Além das técnicas de IA propriamente ditas, também são consideradas em jo-

gos técnicas de gerência do motor de IA. Um dos aspectos mais relevantes da gerência é o modo de funcionamento do laço que atualiza os agentes. Este laço pode ser implementado usando *pooling* ou utilizando tratamento de eventos. Utilizando *pooling* todos os agentes são atualizados a cada ciclo de animação, ao contrário da orientada a eventos onde são atualizados apenas os que foram afetados pelos eventos que foram gerados no ciclo anterior. O tratamento de eventos tem se mostrado mais adequado por apenas analisar as entidades que sofreram alguma mudança no mundo.

Outra técnica de gerenciamento que reduz em muito o número de agentes a serem atualizados é o nível de detalhe. Esta em geral usa uma função heurística para definir a importância de cada entidade tendo como ponto de referência o jogador humano. Desta forma os agentes com menor importância no contexto do jogador podem ser eliminados da atualização e assim reservando o tempo de processamento para os demais. A técnica de balanceamento de carga por sua vez procura distribuir o ciclo de simulação entre os quadros de animação a fim de assegurar a continuidade da animação entre os quadros montados em determinado ciclo de simulação.

Para que a implementação do motor de IA seja útil, as técnicas que fazem parte deste devem ser implementadas de uma forma que se permita a interação entre elas. Além do que, também é necessário que sejam escaláveis, ou seja, possam trabalhar tanto com problemas simples e pequenos quanto com grandes e complexos. Dentre as técnicas implementadas nos motores de IA estão as máquinas de estado [de Bly 2004], sistemas de regras [Christian 2002], árvores de decisão [Buckland 2005], robótica (movimento) [LaMothe 1999], lógica *fuzzy* [McCuskey 2000] e planejamento.

Como os agentes de jogos em geral tomam decisões sobre dados numéricos, tais como distâncias entre objetos, valores destes, probabilidades, decisões através de variáveis nebulosas e avaliação de recursos baseada na teoria dos jogos é desejável que as técnicas utilizadas nas tomadas de decisão possam avaliar expressões. Por outro lado estas técnicas devem poder diferenciar os objetos do mundo e os contextos ao qual se inserem, assim também é necessária a capacidade de avaliação simbólica.

Do ponto de vista de agentes, a maioria dos jogos atuais utiliza apenas agentes reativos e uma pequena minoria tem algum mecanismo de predição. Quando são utilizados agentes reativos as decisões desses dependem apenas do estado atual, ao contrário dos deliberativos que procuram antecipar passos futuros do agente fazendo planos. A especificação de agentes deliberativos é possível através do uso de um planejador.

Planejamento consiste em construir sequências de ações simples que formem um caminho entre o estado atual e o objetivo. Essa sequência de ações é denominada plano e pode ser processada pelo executor de planos que aplica essas operações no mundo a fim de atingir o efeito desejado. Isto é uma boa opção para situações em jogos que exijam IA para tomar decisões de alto nível, como no gerenciamento de recursos, em especial aquelas situações onde o agente tenha muitas opções para desempenhar o mesmo papel.

O operador é uma forma de representação das ações que podem ser instanciadas por um planejador. Este foi introduzido pelo planejador STRIPS (*Stanford Research Institute Problem Solver*) [Fikes and Nilsson 1971] e é adotado na representação no domínio na maioria dos planejadores atuais. Um operador é dividido em três partes, são essas a pré-condição, a lista de atômos que deverão ser adicionados no mundo e a lista dos que

deverão ser excluídos.

Os planejadores também podem ser classificados como clássicos ou hierárquicos. Os planejadores clássicos utilizam busca no espaço de estados associada com análise meio-fim para construir um plano que ligue o estado inicial ao objetivo. A análise meio-fim verifica quais ações devem ser adicionadas ao plano, em ordem, para que depois da execução do mesmo seja obtido o efeito desejado. Por outro lado, os hierárquicos, fazem decomposição de tarefas, ou seja, partem de uma tarefa complexa que é reduzida em uma ou mais tarefas com menor grau.

Nos planejadores hierárquicos são utilizados métodos para reduzir as tarefas de maior nível de abstração. Os métodos especificam as possíveis reduções destas tarefas para conjuntos de zero ou mais tarefas abstratas ou primitivas. Cada redução tem uma condição associada que define se ela é aplicável ou não em determinada situação. As tarefas primitivas são realizadas por operadores que efetivam as mudanças no estado do mundo. Os operadores utilizados para realizar as tarefas primitivas têm uma estrutura parecida com os operadores de STRIPS e cada um deles está associado a uma determinada tarefa primitiva. A realização das tarefas através dos operadores tem o mesmo efeito da instanciação das ações de um plano na abordagem clássica. Tanto nos planejadores clássicos quanto nos hierárquicos, o processo termina quando existir no plano apenas ações ou tarefas primitivas.

Um plano é uma lista ordenada de ações que leva o mundo do estado inicial até o estado desejado. O estado inicial de um problema consiste em um conjunto de átomos que devem ser verdadeiros no mundo no começo do processo de planejamento. O objetivo, no planejamento clássico, é um conjunto de átomos que devem ser verdadeiros no mundo após a execução do plano. Utilizando a abordagem hierárquica o objetivo é dado por uma lista de tarefas que devem ser realizadas.

Os planejadores hierárquicos mais conhecidos são os baseados em redes de tarefas hierárquica (HTN). Nesta abordagem a dependência entre as ações é dada por uma rede de tarefas que contém tarefas primitivas e compostas. Assim, as tarefas compostas que estão na rede devem ser reduzidas para tarefas menos abstratas até que o planejamento chegue às folhas, que são tarefas primitivas, e podem ser adicionadas ao plano.

Entre os algoritmos que utilizam as redes de tarefas hierárquicas como base está SHOP [Nau et al. 1999]. Este algoritmo faz encadeamento para frente e ordenação total das tarefas. SHOP foi a generalização do trabalho de [Smith et al. 1998] cuja abordagem foi utilizada com sucesso na construção do jogo *Bridge Baron*. Em [Nau et al. 2001] foi apresentada uma revisão do algoritmo SHOP denominada SHOP2 cujo principal diferencial era aliviar um pouco a restrição de ordenação total de tarefas, transferindo a responsabilidade da escolha de onde deve ou não deve haver ordenação total de tarefas para o autor do domínio. Esta simples modificação permitiu uma simplificação de até 90% na especificação de alguns domínios.

SHOP e SHOP2 diferem de outros planejadores baseados em HTN por gerarem cada passo do plano na ordem em que esse será executado. Desta maneira, o algoritmo conhece o estado atual do mundo a cada passo do processo de construção do plano. Isto simplifica o planejamento e evita interações indesejáveis entre tarefas. Assim é possível aumentar o poder de expressão do modelo tornando possível incorporar à SHOP computação

numérica e simbólica, inferência utilizando axiomas, interações com agentes externos e outras fontes de informação [Nau et al. 1999].

Segundo [Nau et al. 1999] SHOP é polinomial para a maior parte dos domínios e tanto ele quanto SHOP2, aparentemente, pertencem à mesma classe de complexidade do algoritmo de planejamento TLPLAN [Bacchus and Kabanza 2000], seu concorrente nas competições. Tanto SHOP quanto SHOP2 são completos e corretos [Nau et al. 2001].

O planejamento até hoje é usado timidamente em jogos. A maior parte das implementações utiliza planos prontos associados a estímulos ou faz alguma análise do estado atual utilizando regras que implicam em planos prontos ou regras associadas a arvores de decisão. Alguns jogos mais modernos utilizam planejamento hierárquico a fim de construir agentes orientados a objetivos. Estes agentes são definidos em termos de tarefas que devem ser executadas para atingir o objeto de sua ação.

O uso de planejadores de uso geral em jogos é algo ainda experimental devido principalmente à demora na formulação dos planos ultrapassar em muito os limites de tempo disponíveis para a produçao desses. Mas acreditamos que a escolha de um planejador hierárquico semi-automático é uma solução viável para resolver este problema. Em uma comparação feita em [Long and Fox 2003] os planejadores semi-automáticos tiveram um desempenho maior do que o dos automáticos devido à suas especificações de domínio serem mais descritivas. Isto exige em contra-partida que o engenheiro de conhecimento responsável pelo domínio insira mais detalhes sobre o funcionamento desse na especificação a fim de conduzir o algoritmo mais facilmente ao plano.

Existem duas boas implementações de SHOP, ambas inapropriadas para o uso em jogos. A maior deficiência da abordagem usada em JSHOP2 na implementação de jogos está na compilação dos domínios. JSHOP2 depende da geração do domínio e do problema em código Java e posterior compilação em código objeto para cada par de domínio e de problema antes de iniciar o processo de planejamento. A outra implementação, feita em LISP, acarretaria problemas de acoplamento com o código do motor de jogos devido a dificuldades em implementar técnicas de gerência como balanceamento de carga.

3. Arquitetura

Neste trabalho apresentamos *SHOP2 4 Games* [Porto 2006] que é uma implementação do algoritmo SHOP2 com características desejáveis para seu uso em jogos de computador. O projeto de arquitetura deste sistema busca uma boa relação entre o cumprimento dos requisitos de desempenho exigidos pelos jogos e a qualidade do planejamento. A implementação, embora baseada em JSHOP2, diferencia-se dessa principalmente por ser interpretada e por permitir que vários problemas e domínios sejam explorados paralelamente pelo agente, cabendo a esse escalonar os recursos de processamento.

Ao contrário de JSHOP2 que gera um código java à partir da especificação do domínio e da problema, *SHOP2 4 Games* cria moldes para os domínios e problemas com os quais é possível criar uma infinidade de instancias de planejamento. O processo de criação da instância em nossa abordagem é infinitamente mais rápido do que o de compilação de JSHOP2, o que torna seu uso viável para jogos.

Outra vantagem oferecida por nossa implementação é a possibilidade de formulação dos problemas durante o tempo de simulação. Na prática, em jogos com am-

bientes dinâmicos é impossível prever quais serão as necessidades de um agente. Desta forma o agente pode criar um molde de problema e descrever, de acordo com sua situação atual, o problema que deve ser resolvido utilizando determinado domínio. O agente para isso deverá descrever o estado inicial e atribuir os objetivos que devem ser alcançados através de uma lista de tarefas que será reduzida pelo algoritmo de planejamento.

A Figura 1 mostra a arquitetura da nossa implementação. Esta permite que várias instâncias de problemas sejam resolvidas simultaneamente pelo agente com o uso de *threads*. Isto traz vantagens, pois a arquitetura de agentes pode gerenciar vários pedidos de resolução de problemas e colocá-los para rodar com prioridades diferentes. Há duas restrições para o uso de paralelismo nesta implementação e são essas o *parser* e o mecanismo de instanciação. Para assegurar que o código de leitura e instanciação de domínios e problemas não seja executado simultaneamente por duas *threads* foram utilizados semáforos de proteção para estas duas seções críticas. Mas como a leitura dos arquivos acontece fora do tempo de simulação do jogo e o processo de instanciação é rápido isto não irá acarretar problemas.

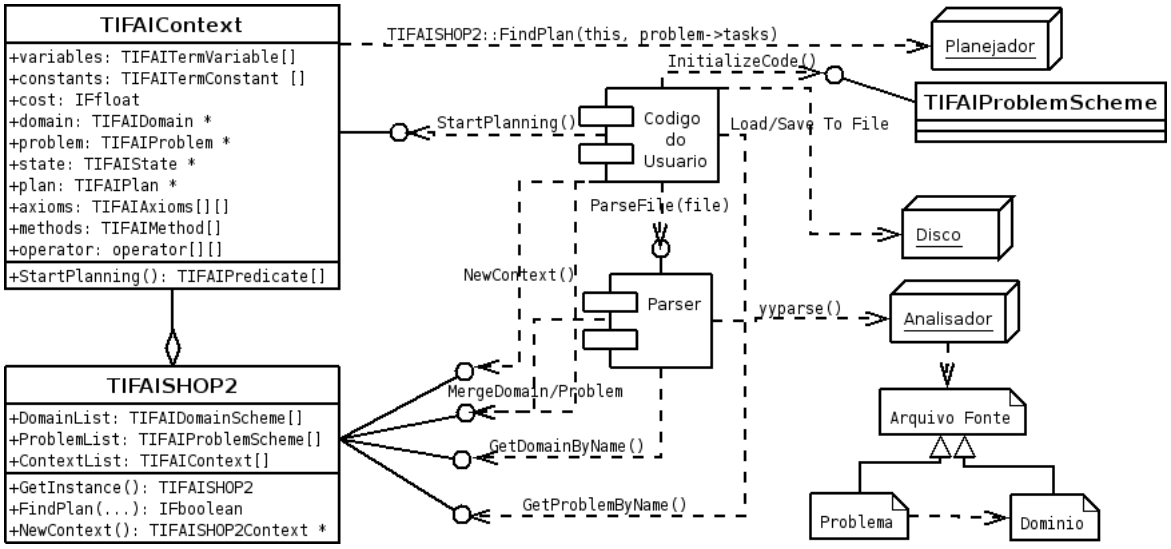

Figura 1. Diagrama de componentes com a visão do usuário.

Para utilizar o planejador o programa que usa a biblioteca deve instanciar um determinado problema de um domínio. Um molde de problema pré-existente é escolhido ou gerado pela aplicação e em seguida é chamada a função "InitializeCode()" que retorna um objeto da classe "TIFAIProblem" que representa o problema instanciado.

Após a instanciação, deve ser criado um contexto (classe "TIFAISHOP2Context") que servirá de plataforma de ligação entre os vários elementos do planejador tais como, estado, domínio, problema, variáveis, etc. Chamando a função membro do contexto "StartPlanning()" o problema será processado e um plano será retornado, ou nulo se não existir plano para aquele problema. O plano é representado por uma sequência de cabeças de tarefas primitivas. Essas cabeças representam ações que podem ser aplicadas diretamente na solução do problema pelo executor de planos ou mesmo podem representar comandos para a solução de problemas, menos abstratos do que o problema original, utilizando outras técnicas de IA para refinamento.

4. Resultados

Em nossos testes usamos o domínio *travel* [Nau et al. 1999] que resolve problemas ligados à escolha de um meio de transporte por um agente que não é motorista. Problemas de gerência de recursos como os resolvíveis por este domínio são comuns em jogos e justificam a utilização de um planejador. Aqui também fica evidente a eficiência de SHOP2 para solucionar problemas em tempo hábil quando aplicado em jogos. Por fim são comparados nossos resultados aos de JSHOP2 [Ilghami 2005] nesta aplicação.

Neste domínio, o ator principal pode escolher entre ir à pé, de táxi ou de ônibus ao centro, ao subúrbio ou ao parque. A decisão de "como" ir é tomada pelo agente baseando-se nas tarefas que devem ser realizadas por ele. Alguns fatos também influenciam na decisão do agente, entre eles estão o clima (se está bom ou ruim) e a quantia de dinheiro que o ator tem para gastar na viagem.

O custo da viagem de ônibus é de $1.00 e a de táxi custa $1.50, no segundo caso acrescido de $1.00 por quilômetro rodado. Embora a viagem à pé seja gratuita, esta é condicionada ao clima. Se o clima estiver ruim, então a distância que ele pode caminhar é bem menor do que a que poderia num dia de sol.

O agente interage com outros atores no mundo a fim de realizar seus objetivos. Esta interação é realizada em nível de arquitetura de agentes e não é abordada neste trabalho. Cada agente segue os planos especificados pelos domínios que ditam seu comportamento. Dois casos de interação são evidentes neste domínio.

O primeiro caso é quando o agente toma um táxi. Neste o motorista automático do táxi está em um estado de espera por um plano que deve ser entregue pelo passageiro. A partir do recebimento deste plano o motorista deve aplicar uma técnica de busca de caminhos [Buckland 2005] para escolher a melhor rota a ser seguida para fazer a ligação entre os dois pontos. A espera pelo passageiro e o comportamento do motorista em trafego podem ser implementados utilizando máquinas de estado.

O segundo caso é quando o veículo escolhido é o ônibus. Neste, o motorista automático de ônibus segue um plano pré-fixado, ou seja, independente da vontade do ator principal os ônibus vão circular, cabendo a esse apenas a decisão de usar o transporte ou não. Neste caso a rota do veículo é pré-definida e muda somente por motivo de força maior, quando deve haver um re-planejamento do caminho com o algoritmo de busca de caminhos. O tráfego deste motorista também é governado por uma máquina de estados.

A metodologia escolhida para comparar nossa abordagem com JSHOP2 é considerar o caso em que vários problemas são compilados sequencialmente utilizando o mesmo domínio. Neste caso, utilizando JSHOP2, seria necessário fazer n compilações do código do domínio ligado ao código de cada um dos n problemas. Isto causa uma sobrecarga muito grande na linha de montagem de planos, tomando na maioria das vezes mais tempo para fazer a compilação do planejador do que esse utiliza para fazer o planejamento. O tempo total usado pelas implementações que compreende a leitura do domínio e do problema, a carga destes e a execução do planejamento também é considerada para fins de comparação entre as abordagens.

De posse de um problema e do domínio relacionado a esse é possível efetuar o planejamento em ambas as abordagens. Em JSHOP2 o problema e o domínio são com-

pilados e deste é gerado um código objeto que pode ser executado. Isto se diferencia de *SHOP2 4 Games* que é interpretado e permite que múltiplos arquivos contendo descrições de domínios e problemas sejam carregados e eventualmente sejam eleitos para serem instanciados e sirvam de subsídio para uma busca por planos.

Na Tabela 1 aparece o resultado de um confronto entre as duas abordagens. Neste confronto foram utilizadas 10 variações do problema solucionável pelo domínio *travel*. Para gerar estas variações foram modificados o estado inicial e as tarefas a serem realizadas pelo agente. Basicamente no estado inicial foi modificado o lugar onde o agente se encontra no início da execução do planejamento. Já na reformulação de tarefas foram escolhidos novos destinos para o agente.

Tabela 1. Desempenho *SHOP2 4 Games* x JSHOP2 no domínio *travel*.

Problemas	JSHOP2			*SHOP2 4 Games*		
	Compilação	Carga	Planejamento	Compilação	Carga	Planejamento
01	2,844	0,350	0,050	0,035	0,0003	0,00158
02	2,794	0,370	0,040	0,005	0,0003	0,00062
03	2,794	0,360	0,040	0,005	0,0003	0,00030
04	2,784	0,360	0,040	0,005	0,0003	0,00030
05	2,744	0,360	0,040	0,005	0,0003	0,00030
06	2,744	0,350	0,040	0,005	0,0003	0,00124
07	2,744	0,360	0,040	0,005	0,0003	0,00030
08	2,734	0,360	0,040	0,005	0,0003	0,00030
09	2,734	0,350	0,050	0,005	0,0003	0,00030
10	2,734	0,360	0,040	0,005	0,0003	0,00126

Para fazer os testes apresentados na Tabela 1 o tempo total de planejamento foi dividido em três partes, são essas:

- Compilação - O tempo que o sistema de planejamento leva para ler os arquivos fonte com o domínio e o problema que deve ser resolvido. Neste processo em JSHOP2 é feito um programa executável na linguagem Java e em *SHOP2 4 Games* a descrição dos domínios e problemas são colocadas em estruturas na memória do computador denominadas moldes. Estes moldes são utilizados na fase de carga para gerar a estrutura instanciada de planejamento.
- Carga - Este é o tempo que o sistema de planejamento leva para estar pronto para iniciar o planejamento propriamente dito. Em JSHOP2 este consiste no tempo que a máquina virtual demora em carregar o executável gerado pelo processo de compilação. Em *SHOP2 4 Games* este é o tempo levado pelo sistema para instanciar a estrutura de moldes para que o processo de planejamento possa começar.
- Planejamento - É o tempo que o algoritmo SHOP2 leva para planejar em cada uma das abordagens.

Embora todos os dez problemas tenham propositalmente o mesmo custo de compilação, carga e planejamento, JSHOP2 tem um custo de compilação constante enquanto *SHOP2 4 Games* tem o custo de compilação do domínio no início e um custo para

cada problema. Na Tabela 1 o custo da compilação do domínio foi agrupado no custo da compilação do primeiro problema, sendo que nos próximos problemas foi computado apenas o custo da leitura do problema. Mesmo assim os custos de JSHOP2 para está fase foram no mínimo 81 vezes mais altos [Porto 2006].

Acreditamos que estas diferenças nos tempos de compilação e carga apresentadas na Tabela 1 entre as duas abordagens são dadas pela necessidade de utilizar um compilador durante o processo e os tempos de carga de executáveis do sistema operacional e da máquina Java. Também tivemos alguns problemas ao tentar cancelar a produção de um plano ou parar o planejador quando usado JSHOP2, o que às vezes é útil pelo fato de, nem sempre, os planos saírem em um tempo viável nesta aplicação.

O tempo total de processamento do domínio *travel* e de seus dez problemas foi de cerca de 31 segundos para JSHOP2 e cerca de 9 milissegundos utilizando a nossa abordagem. Acreditamos que isto justificou o esforço despendido na construção do planejador para jogos [Porto 2006].

Durante o tempo de execução do jogo os agentes deverão fazer chamadas ao planejador de acordo com suas necessidades de obtenção de planos de mais alto nível que ditam as estratégias que cada um deles deve seguir.

Existe dezenas de pontos no código fonte de um jogo onde seria interessante solicitar um plano de ação para que o agente realize seus objetivos. Tendo um planejador em seu conjunto de ferramentas, cabe ao desenvolvedor do jogo utilizá-lo racionalmente no desenvolvimento da aplicação juntamente com as outras técnicas de IA.

5. Conclusão

Jogos de computador são sistemas de computação que ao longo de sua história sempre demandaram muito esforço de desenvolvimento e sempre que há disponibilidade de computadores mais potentes para uso doméstico existe o interesse de experimentar o uso de novas ferramentas que aumentem o realismo dos jogos.

Um sistema planejador é uma dessas ferramentas. Este aspecto vem sendo discutido pelos desenvolvedores de jogos há bem pouco tempo, mas acredita-se que pode propiciar aos agentes de jogos maior qualidade nas decisões de mais alto nível. Embora o desenvolvimento de um planejador para jogos seja uma tarefa bastante cara e difícil de ser realizada, este tem-se mostrado cada vez mais viável.

Neste contexto apresentamos *SHOP2 4 Games*, uma biblioteca de *software* que consiste em uma implementação do sistema de planejamento adaptada para ser anexada em um motor de IA para jogos.

Nossos testes mostraram que *SHOP2 4 Games* está apta a ser utilizada pelos agentes de um jogo com um tempo de execução razoável e produzindo planos de boa qualidade. Vários trabalhos futuros podem prosseguir: a formulação dos objetivos e do estado inicial de planejamento em tempo de simulação e também a questão da serialização da execução de planos. Finalmente deve ser feita a integração da biblioteca com o motor de IA e este com o motor de jogos.

Referências

Bacchus, F. and Kabanza, F. (2000). Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116(1-2):123–191.

Buckland, M. (2005). *Programming Game AI by Example*. Wordware Publishing Inc, 1ª edition.

Christian, M. (2002). A Simple Inference Engine for a Rule-Based Architecture. *AI Game Programming Wisdom*, pages 305–320.

de Bly, P. B. (2004). *Programming Believable Characters for Computer Games*. Charles River Media, 1ª edition.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence 2*, pages 189–208.

Hawes, N. (2003). *Anytime Deliberation for Computer Games Agents*. PhD thesis, The University of Birminghan.

Ilghami, O. (2005). Documentation for JSHOP2. Technical report, University of Maryland. CS-TR-4694.

LaMothe, A. (1999). *Things of Windows Game Programming Gurus*. Sams, 1ª edition.

Long, D. and Fox, M. (2003). The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research 20*. 3rd International Planning Competition.

McCuskey, M. (2000). Fuzzy Logic and Video Games. *Game Programming Gems*, pages 319–329.

Nau, D. S., AU, T. C., Ilghami, O., Kuter, U., Murdock, W., Wu, D., and Yaman, F. (2001). Total-Order Planning with Partially Ordered Subtasks. *In IJCAI-2001*, pages 968–973.

Nau, D. S., Muñoz-Avila, H., and Lotem, A. (1999). SHOP: Simple Hierarchical Ordered Planner. *In IJCAI-99*, pages 968–973.

Pedersen, R. E. (2003). *Game Design Foundations*. Wordware Publishing, 1ª edition.

Porto, S. A. (2006). Planejamento em redes de tarefas hierárquicas com aplicação em jogos. Master's thesis, Universidade Federal do Paraná.

Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2ª edition.

Smith, S., Nau, D., and Throop, T. (1998). Computer Bridge: A Big Win for AI Planning. *AI Magazine*, 19(2):93–106.