

UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

VISEDU-SIMULA 1.0: VISUALIZADOR DE MATERIAL
EDUCACIONAL, MÓDULO DE ANIMAÇÃO
COMPORTAMENTAL

GUSTAVO RUFINO FELTRIN

BLUMENAU
2014

2014/2-11

GUSTAVO RUFINO FELTRIN

**VISEDU-SIMULA 1.0: VISUALIZADOR DE MATERIAL
EDUCACIONAL, MÓDULO DE ANIMAÇÃO
COMPORTAMENTAL**

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Dalton Solano dos Reis , M. Sc. - Orientador

**BLUMENAU
2014**

2014/2-11

**VISEDU-SIMULA 1.0: VISUALIZADOR DE MATERIAL
EDUCACIONAL, MÓDULO DE ANIMAÇÃO
COMPORTAMENTAL**

Por

GUSTAVO RUFINO FELTRIN

Trabalho de Conclusão de Curso aprovado
para obtenção dos créditos na disciplina de
Trabalho de Conclusão de Curso II pela banca
examinadora formada por:

Presidente:	_____ Prof. Dalton Solano dos Reis, M. Sc. – Orientador, FURB
Membro:	_____ Prof. Mauro Marcelo Mattos, Dr. – FURB
Membro:	_____ Prof. Nome do professor, Titulação – FURB [@@ fazer]

Blumenau, dia de mês de ano [data da apresentação] [@@ fazer]

[@@ fazer...] Dedico este trabalho ...
[Geralmente um texto pouco extenso, onde o autor homenageia ou dedica o trabalho a alguém. Colocar a partir do meio da página.]

AGRADECIMENTOS

[@@ fazer] A Deus...

À minha família...

Aos meus amigos...

Ao meu orientador...

[Colocar menções a quem tenha contribuído, de alguma forma, para a realização do trabalho.]

[@@ fazer] [Epígrafe: frase que o estudante considera significativa para sua vida ou para o contexto do trabalho. Colocar a partir do meio da página.]

[Autor da Epígrafe]

RESUMO

[@@ fazer] O resumo é uma apresentação concisa dos pontos relevantes de um texto. Informa suficientemente ao leitor, para que este possa decidir sobre a conveniência da leitura do texto inteiro. Deve conter OBRIGATORIAMENTE o **OBJETIVO, METODOLOGIA, RESULTADOS** e **CONCLUSÃO**. O resumo deve conter de 150 a 500 palavras e deve ser composto de uma seqüência corrente de frases concisas e não de uma enumeração de tópicos. O resumo deve ser escrito em um único texto corrido (sem parágrafos). Deve-se usar a terceira pessoa do singular e verbo na voz ativa (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2003).

Palavras-chave: Animação comportamental. Simuladores. Jogos sérios. Jogos 2D. HTML5. Javascript.

ABSTRACT

[@@ do] *Abstract* é o resumo traduzido para o inglês. *Abstract* vem em uma nova folha, logo após o resumo. Escrever com letra normal (sem itálico).

Key-words: Behavioral animation. Simulators. Serious Games. 2D games. HTML5. Javascript.

LISTA DE FIGURAS

Figura 1 - Diagrama de casos de uso do motor de jogos	24
Figura 2 - Diagrama de pacotes do motor de jogos	26
Figura 3 - Diagrama de classes do pacote <code>component</code>	27
Figura 4 - Diagrama de classes do pacote <code>game</code>	28
Figura 5 - Diagrama de classes do pacote <code>gameobject</code>	28
Figura 6 - Diagrama de classes do pacote <code>perception</code>	29
Figura 7 - Diagrama de classes do pacote <code>system</code>	29
Figura 8 - Diagrama de classes do pacote <code>utils</code>	30
Figura 9 - Diagrama de casos de uso do editor de jogos	31
Figura 10 - Diagrama de pacotes do editor de jogos	32
Figura 11 - Diagrama de classes do pacote <code>br.com.furb.html5.game.editor.controller</code>	32
Figura 12 - Diagrama de classes do pacote <code>br.com.furb.html5.game.editor.utils</code>	33
Figura 13 - Diagrama de casos de uso do Raciocinador	33
Figura 14 - Diagrama de pacotes do raciocinador	35
Figura 15 - Diagrama de classes do pacote <code>br.furb.visedu.log</code>	35
Figura 16 - Diagrama de classes do pacote <code>br.furb.visedu.reasoner.jason</code>	36
Figura 17 - Diagrama de Sequencia do <i>loop</i> principal do motor de jogos	38
Figura 18 - Camadas do editor, do motor de jogos e do raciocinador	39
Figura 19 - Criação de um novo projeto	53
Figura 20 - Adicionando o componente determinado <code>RemoveGravityComponent</code>	53
Figura 21 - Adicionando o <code>Game Object</code> do tipo <code>BoxObject</code>	54
Figura 22 - Adicionando uma mente <code>Jason</code> ao editor	56
Figura 23 - Execução do protótipo da simulação	59
Figura 24 - Cenário da aplicação	61
Figura 25 - Execução do protótipo da simulação para testes de percepção	62
Figura 26 - Protótipo de simulação para Cenário A	65
Figura 27 - Ativação do recurso <code>FEATURE_WEBSOCKET_MAXCONNECTIONSPERSERVER</code> definindo 128 conexões simultâneas	67

Figura 28 - Gráfico comparativo do tempo de estabilização de conexão entre os navegadores	67
Figura 29 - Gráfico comparativo do tempo de estabilização de conexão entre os navegadores (sem Mozilla Firefox)	68
Figura 30 - Protótipo de simulação para Cenário B	69
Figura 32 - Colisão entre objeto que representa um agente e objeto que representa seu campo de visão	71
Figura 32 - Comparação da quantidade de quadros por segundo de uma mesma simulação com e sem a presença de raciocínio.....	73

LISTA DE QUADROS

Quadro 1 - Caso de uso UC20	24
Quadro 2 - Caso de uso UC21	25
Quadro 3 - Caso de uso UC22	25
Quadro 4 - Caso de uso UC23	34
Quadro 5 - Caso de uso UC24	34
Quadro 6 - Caso de uso UC25	34
Quadro 7 - Implementação da classe <code>ReasonerJasonServlet</code>	42
Quadro 8 - Implementação da classe <code>ReasonerJasonWebSocket</code>	43
Quadro 9 - Implementação do método <code>act(ActionExec, List<ActionExec>)</code> da classe <code>Agente</code>	44
Quadro 10 - Implementação de uma mente em <code>AgentSpeak(L)</code>	44
Quadro 11 - Novo eventos do objeto <code>Component</code>	45
Quadro 12 - Código fonte de classe <code>Game</code>	45
Quadro 13 - Implementação da classe <code>PerceptionSystem</code>	46
Quadro 14 - Implementação da sobrescrita do método <code>initialize</code> da classe <code>PerceptionVisionComponent</code>	47
Quadro 15 - Implementação do método <code>onPercept(gameObjectPerceived)</code> na classe <code>PerceptionVisionComponent</code>	48
Quadro 16 - Exemplo de mensagem enviada pela simulação para o raciocinador	49
Quadro 17 - Exemplo de mensagem enviada pelo raciocinador para a simulação	49
Quadro 18 - Implementação do método <code>onMessage(MessageEvent)</code> na classe <code>PerceptionVisionComponent</code>	50
Quadro 19 - Implementação de exemplo da classe <code>PerceptionVisionPerformanceComponent</code>	51
Quadro 20 - Implementação da mente da presa em <code>AgentSpeak</code>	56
Quadro 21 - Implementação da classe <code>PerceptionVisionLotkaVolterraComponent</code>	57
Quadro 22 - Estados do protótipo da simulação exemplificada	63
Quadro 23- Implementação da mente do exemplo <i>Domestic Robot</i> (JASON, 2014c)	84

LISTA DE TABELAS

Tabela 1 – Média de tempo para estabelecer conexão com o raciocinador nos principais navegadores	66
Tabela 2 – FPS versus tempo médio de raciocínio.....	70
Tabela 3 - Quantidade de quadros por segundo por objetos na cena (com detecção da percepção).....	72
Tabela 4 - Média de tempo para estabelecer conexão com o raciocinador - Google Chrome .	78
Tabela 5 - Média de tempo para estabelecer conexão com o raciocinador - Internet Explorer	78
Tabela 6 - Média de tempo para estabelecer conexão com o raciocinador - Opera	79
Tabela 7 - Média de tempo para estabelecer conexão com o raciocinador – Mozilla Firefox .	79
Tabela 8 - FPS versus tempo médio de raciocínio - Google Chrome	80
Tabela 9 - FPS versus tempo médio de raciocínio - Mozilla Firefox.....	80
Tabela 10 - FPS versus tempo médio de raciocínio - Internet Explorer.....	81
Tabela 11 - FPS versus tempo médio de raciocínio - Opera	81
Tabela 12 - Quantidade de quadros por segundo por objetos na cena (com detecção da percepção) - Google Chrome	82
Tabela 13 - Quantidade de quadros por segundo por objetos na cena (com detecção da percepção) - Mozilla Firefox	82
Tabela 14 - Quantidade de quadros por segundo por objetos na cena (com detecção da percepção) - Internet Explorer	83
Tabela 15 - Quantidade de quadros por segundo por objetos na cena (com detecção da percepção) - Opera.....	83

[@@ remover]

[@@ Dalton: algum contraindicação com nomes títulos longos?]

LISTA DE SIGLAS

[@@ fazer, o word não tem nada para isso?]

[Deve conter as siglas utilizadas mais de uma vez ao longo do texto em ordem alfabética. A seguir estão dois exemplos de forma de apresentação.]

ABNT – Associação Brasileira de Normas Técnicas

API – *Application Programming Interface*

SUMÁRIO

1 INTRODUÇÃO.....	15
1.1 OBJETIVOS.....	16
1.2 ESTRUTURA.....	16
2 FUNDAMENTAÇÃO TEÓRICA	17
2.1 ANIMAÇÃO COMPORTAMENTAL	17
2.2 SIMULADORES – JOGOS	18
2.3 HTML5 E JAVASCRIPT.....	19
2.4 TRABALHOS CORRELATOS	20
2.4.1 Unity 3D.....	20
2.4.2 Aperfeiçoamento de Reações Comportamentais de Non-Player Character (NPC) no Jogo Doom	21
2.4.3 Massive	21
3 DESENVOLVIMENTO.....	23
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO.....	23
3.2 ESPECIFICAÇÃO	23
3.2.1 Diagrama de casos de uso do motor de jogos	23
3.2.2 Diagramas de pacotes do motor de jogos.....	25
3.2.2.1 Pacote component	26
3.2.2.2 Pacote game	28
3.2.2.3 Pacote gameobject.....	28
3.2.2.4 Pacote perception.....	29
3.2.2.5 Pacote system	29
3.2.2.6 Pacote util	30
3.2.3 Diagrama de casos de uso do editor de jogos	30
3.2.4 Diagramas de classes do editor de jogos.....	31
3.2.4.1 Pacote br.com.furb.html5.game.editor.controller.....	32
3.2.4.2 Pacote br.com.furb.html5.game.editor.model.....	33
3.2.5 Diagrama de casos de uso do raciocinador	33
3.2.6 Diagrama de pacotes do raciocinador	34
3.2.6.1 Pacote br.furb.visedu.log	35

3.2.6.2 Pacote br.furb.visedu.reasoner.jason	36
3.2.7 Diagrama de Sequência.....	37
3.2.8 Diagrama de Arquitetura.....	39
3.3 IMPLEMENTAÇÃO	40
3.3.1 Técnicas e ferramentas utilizadas.....	40
3.3.2 Módulo de Animação Comportamental	41
3.3.2.1 Módulo de Raciocínio.....	41
3.3.2.2 O motor de jogos	44
3.3.2.3 O editor de jogos.....	52
3.3.2.4 Operacionalidade da implementação	52
3.4 RESULTADOS E DISCUSSÃO	59
3.4.1 Arquitetura desenvolvida	60
3.4.2 Teste de funcionalidades	61
3.4.3 Teste de desempenho	63
3.4.3.1 Cenário A.....	64
3.4.3.2 Cenário B	68
3.4.4 Comparativo entre o módulo desenvolvido e seus correlatos.....	73
4 CONCLUSÕES.....	74
4.1 EXTENSÕES	74
REFERÊNCIAS	75
Apêndice A – Tabelas com amostras do Cenário A.	78
Apêndice B – Tabelas com amostras do Cenário B.....	80
Apêndice C – Tabelas com amostras do Cenário B sem o processamento do raciocinador. 82	
ANEXO A – Exemplo de uma mente criada em AgentSpeak interpretada pelo Jason...84	

1 INTRODUÇÃO

Segundo Magnenat-Thalmann e Thalmann (2004, p. 260, tradução nossa), “a utilização de Animação Comportamental tem gerado muitas pesquisas, em especial na indústria do entretenimento”. O jogo *Creature* foi o primeiro a adicionar redes neurais aos jogos e deixá-la ensinar as suas criaturas como elas deveriam se comportar. Mais recentemente, o jogo *The Sims* focou na simulação de seres humanos virtuais em seu dia a dia, com resultados bem sucedidos e divertidos. Embora os comportamentos ainda sejam limitados, estes jogos têm demonstrado que as pessoas são atraídas para jogar com agentes autônomos e o sucesso recente do jogo *Black and White* prova isso mais uma vez (MAGNENAT-THALMANN; THALMANN, 2004, p. 260-261).

Para o desenvolvimento de Animação Comportamental, necessariamente a mesma precisa ocorrer em algum meio, que é um simulador. O simulador cria o ambiente em que o personagem dotado de uma Animação Comportamental irá existir e no mesmo são dispostos e definidos a forma que o ambiente irá coexistir com esse personagem, tendo todas as regras que o personagem irá perceber a sua volta.

Outro fator a ser observado nos jogos atuais é a adoção da plataforma Web para o desenvolvimento. Em particular usando o *Hypertext Markup Language*, versão 5 (HTML5), que pode ser considerado o mais recente padrão para HTML. Trata-se de uma cooperação entre o *World Wide Web Consortium* (W3C) e a *Web Hypertext Application Technology Working Group* (WHATWG). Foi especialmente concebido para proporcionar um conteúdo rico, sem a necessidade de *plugins* adicionais. A versão atual oferece desde animações para gráficos, músicas para filmes e também pode ser usado para construir aplicações web. Os principais navegadores (Chrome, Firefox, Internet Explorer, Safari, Opera) suportam as novas *Application Programming Interface* (APIs) e os novos elementos do HTML5 e continuam a adicionar novos recursos do HTML5 a suas últimas versões (W3SCHOOLS, 2014a).

Além do HTML5 é comum associar o uso do JavaScript para desenvolver jogos, principalmente em 2D. O JavaScript é uma linguagem de *script* de programação leve que pode ser inserido em qualquer página HTML e pode ser executado por todos os tipos de navegadores web (W3SCHOOLS, 2014b).

Diante do exposto, foram estendidos os módulos desenvolvidos por Harbs (2013), criando um simulador 2D, utilizando os conceitos de jogos, que possibilitou gerar animações comportamentais nos personagens criado pelo Motor de jogos 2D.

1.1 OBJETIVOS

O objetivo desse trabalho desenvolvido é criar um simulador 2D para geração de animações comportamentais.

Os objetivos específicos do trabalho são;

- a) estender os módulos desenvolvidos em Harbs (2013) para permitir criar a Animação Comportamental;
- b) proporcionar um controle mínimo da percepção, raciocínio e atuação do personagem de forma desacoplada a inteligência utilizada;
- c) utilizar um dos modelos clássicos da Inteligência Artificial (IA) (reativo, rede neural, sistema especialista, *Belief Desire Intention* (BDI), entre outros) para poder testar o simulador.

1.2 ESTRUTURA

O trabalho desenvolvido está organizado em quatro capítulos. O capítulo 2 apresenta a fundamentação teórica, proporcionando embasamento teórico para compreensão do trabalho. O capítulo 3 apresenta o desenvolvimento do módulo de Animação Comportamental, descrevendo as implementações do raciocinador utilizando diagramas de pacotes, diagramas de classes, diagrama de sequência e diagrama de arquitetura e descrevendo os ajustes e adaptações realizados no motor e no editor de jogos criados por Harbs (2013). No capítulo 4 são apresentadas as conclusões e as extensões sugeridas.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo dispõe de quatro seções, na seção 2.1 será abordada a Animação Comportamental, na seção 2.2 são abordados simuladores e jogos, na seção 2.3 são abordados os HTML5 e Javascript e na seção 2.4 são abordados os trabalhos correlatos a este trabalho.

2.1 ANIMAÇÃO COMPORTAMENTAL

Animação Comportamental consiste no fato de, em uma cena, animar um personagem dotado de estados, que possui comportamento próprio e restrições, capaz de realizar ações com propósito de alcançar seus objetivos. Utilizando de técnicas de Inteligência Artificial possibilita-se que em um mundo aberto tal personagem tenha a capacidade de perceber o mesmo e, a partir desse contato, agir de uma melhor maneira para cumprir um propósito (MENDONÇA JR., [1999?]).

Dessa forma, o personagem torna-se capaz de realizar improvisações quando necessário, evitando também que o animador seja obrigado a definir todos os detalhes nos movimentos do personagem (REYNOLDS, 1997).

Já Roncarelli (1989, p. 13) afirma que nas Animações Comportamentais

[...] são definidas regras para determinar as ações de certas classes de objetos no sistema. As regras de interação de uma com as outras para criar um comportamento que é um pouco, mas não totalmente, previsível. Um exemplo comum é o peixe - eles nadam dentro de parâmetros de velocidade específicos, dentro de um determinado raio, evitam objetos sólidos e possuem um instinto de ficar perto do centro do cardume.

A Animação Comportamental é uma, de uma série de abordagens de automatizadores do controle do movimento que têm aparecido nos últimos anos. Estas abordagens podem ajudar a remover (quando se desejar) o ônus da especificação explícita do movimento pelo animador (MAGNENAT-THALMANN; THALMANN, 1990, p. 94).

A Animação Comportamental possui conceitos próximos aos de agentes inteligentes. O primeiro conceito é a percepção, que é um processo de reconhecimento, uma transformação das informações coletadas por sensores externos e enviadas para si próprio (WOOLDRIDGE; JENNINGS, 1994, p. 235). O segundo conceito é o raciocínio, que se entende por qualquer coisa que é capaz de ter crenças, fatos, ou seja, que está disposta a aceitar verdades. Capacidade de auferir conclusões sobre um conjunto de hipóteses próprias e/ou alheias (MULLER et al., 1997, p. 118). O último conceito é a ação, que é um comportamento reativo para uma resposta imediata de um estímulo externo (WOOLDRIDGE; JENNINGS, 1994, p. 303).

Entre os possíveis modelos mentais utilizados para realização do raciocínio, pode-se citar o modelo *belief–desire–intention* (BDI) que é uma das abordagens mais conhecidas para o desenvolvimento de agentes cognitivos. Baseada na arquitetura BDI, uma das linguagens abstratas orientadas a agentes mais influentes é a linguagem AgentSpeak. Muitas vezes, os agentes programados com AgentSpeak são referenciados a sistemas de planejamento reativos (JASON, 2014b).

Segundo Bordini¹ et al. (2006 apud WIKIPEDIA, 2014), devido o desenvolvimento da plataforma Jason, AgentSpeak é uma das linguagens orientadas a agentes mais populares. Distribuído sob a licença GNU LGPL², Jason é um interpretador para uma versão estendida da linguagem AgentSpeak. A plataforma implementa a semântica operacional da linguagem e possibilita o desenvolvimento de sistemas multi-agente, com muitas características customizáveis pelo usuário (JASON, 2014a).

Arquivos com código AgentSpeak possuem a extensão `.asl`, nos mesmos são realizadas as definições das mentes dos agentes, através de crenças, planos, etc. Exemplos como *Airport Robots* – uma simulação de robôs trabalhando em um aeroporto (ajudando passageiros, fazendo verificações de segurança, etc) ou *Gold Miners* – uma simulação de uma equipe de agentes de mineração recuperando peças de ouro dispersas em um território, exemplificam como mentes em AgentSpeak podem se tornar complexas, podendo possuir mais de 200 linhas de definições (JASON, 2014c).

No Anexo A foi disponibilizada a implementação da mente de um robô doméstico que pega cerveja da geladeira quando ordenado pelo proprietário.

2.2 SIMULADORES – JOGOS

No desenvolvimento da Animação Comportamental, é comum associar-se a construção de simuladores. Entre os simuladores que existem, pode-se citar o *Multiple Agent Simulation System In Virtual Environment* (MASSIVE), desenvolvido pela empresa Massive Software (MASSIVE, 2014c).

Segundo Cornélio Filho (1998 apud SCHULTER, 2007, p. 15),

historicamente a simulação, como técnica, originou-se dos estudos de Von Neumann e Ulan. Estes estudos ficaram conhecidos como análise ou técnica de Monte Carlo. A simulação começou a ser mais utilizada como técnica para solução de problemas, principalmente para o tratamento dos problemas eminentemente probabilísticos, cuja solução analítica é, geralmente, muito mais árdua e difícil, senão impossível.

¹ Um dos desenvolvedores do Jason.

² Informações sobre a licença GNU LGPL disponíveis em <http://www.gnu.org/licenses/lgpl.html>

No desenvolvimento dos simuladores, é comum utilizar-se todo o conhecimento ou rotinas gráficas que são utilizadas em jogos. No caso em particular, pensando em jogos 2D, que em geral precisam ter um grafo de cena, necessitam de componentes, *assets*, entre outros recursos, sendo um deles o próprio personagem.

Outra abordagem de simuladores seria no caráter de jogos sérios, que por definição, referem-se a aplicações desenvolvidas utilizando tecnologias de jogos de computador que servem a propósitos que não seja puro entretenimento. O termo tem sido usado para descrever uma variedade de tipos de jogos, particularmente aqueles associados com *e-learning*³, simulação militar e treinamento médico (ARNAB et al., 2013).

Essa abordagem se dá devido os simuladores possuírem muitas características dos jogos (ambiente fechado, possuir personagens, qualidades gráficas, etc.), porém detendo propósitos profissionais como treinamentos, obtenção de dados, simulações de fenômeno, entre outros.

2.3 HTML5 E JAVASCRIPT

O HTML5 é um padrão para estruturação e apresentação de conteúdo na web. Ele incorpora recursos como geolocalização, reprodução de vídeo e *drag-and-drop*. HTML5 permite aos desenvolvedores criar aplicações ricas para internet sem a necessidade de APIs e *plug-ins* de terceiros. Muitos aspectos do HTML5 já estão estáveis e podem ser implementados nos navegadores (SUGRUE, 2010, p. 1).

Outro recurso incorporado a especificação do HTML5 na parte de comunicação é a *WebSocket*, que define um canal de comunicação *full-duplex* (bidirecional), que opera através da Web através de um único *socket*. *WebSocket* não é apenas mais um aprimoramento gradual para comunicações convencionais sob HTTP; ela representa um grande avanço, especialmente para aplicações web de tempo real, orientadas a eventos (LUBBERS, 2011, p.1).

O principal arquiteto da especificação do HTML5, Ian Hickson (W3C, 2014b), dentre as vantagens da *WebSocket*, em carta aberta afirma:

Reduzindo kilobytes de dados a 2 bytes é mais do que "um pouco mais eficiente" e reduzindo a latência de 150ms (ida e volta TCP para configurar a conexão somada ao pacote para a mensagem) para 50 ms (apenas o pacote para a mensagem) é muito mais do que marginal. Na verdade, esses dois fatores por si só são suficiente para fazer da *WebSocket* seriamente interessante para o Google. (HICKSON, 2009, tradução nossa).

³ Modalidade de ensino à distância, utilizada para definir aprendizagem por meio de mídia eletrônica.

Em comentário no blog oficial da W3C, o CEO Jeff Jaffe informou que agora com o HTML5 pronto, a W3C deve se concentrar no fortalecimento de uma plataforma Web aberta (W3C, 2014a).

Ao abordar a linguagem de marcação HTML5 é comum associar-se a linguagem JavaScript. Originalmente a linguagem se chamava LiveScript, porém mais tarde resolveram trocar o nome para mostrar sua proximidade com a linguagem Java, mesmo se tratando de linguagens totalmente distintas. O JavaScript foi desenvolvido pela Netscape e por ser uma linguagem interpretada seus códigos são escritos em forma de texto e na linguagem (em inglês) compreensível por nós humanos. Mais tarde, um interpretador, disponível em todos os navegadores mais populares transforma essa linguagem humana em linguagem de máquina (SILVA, 2003, p. 21-22).

Segundo Levent (2013, p. 1, tradução nossa), “assim como a maturidade dos padrões do HTML5, o JavaScript tornou-se de fato a linguagem de programação da web para ambientes do lado do cliente e rapidamente está ganhando terreno no lado do servidor.” Tal como as bases de código do JavaScript crescem amplamente e complexas, as capacidades de orientação a objetos da linguagem tornam-se cada vez mais cruciais. Embora as capacidades da orientação a objetos do JavaScript não serem tão poderosas como os das linguagens de servidor como Java, o modelo atual orientado a objetos pode ser utilizado para escrever um código melhor (LEVENT, 2013, p. 1).

2.4 TRABALHOS CORRELATOS

Como trabalhos correlatos a este, foram selecionados, o motor de desenvolvimento Unity 3D (UNITY, 2014a), desenvolvida pela empresa Unity Technologies, o trabalho de Mattedi (2007) e o simulador MASSIVE, desenvolvido pela Massive Software (MASSIVE, 2014a).

2.4.1 Unity 3D

A Unity 3D é um motor de desenvolvimento de jogos para criação de jogos 2D, 3D e conteúdo interativo (UNITY, 2014b). Trata-se de um ambiente flexível e intuitivo que possibilita realizar a criação, comercialização e a implantação em diversas plataformas. A mesma se adapta as necessidades dos utilizadores através de uma rica linha de ferramentas (*assets*) que podem ser incorporadas ao ambiente de desenvolvimento (UNITY, 2014a).

A *Asset Store* é um repositório de recursos que podem ser adicionados ao ambiente, existindo hoje centenas de recursos disponíveis que podem ser incorporados ao ambiente,

alguns com custos e outros não. Dentre os *assets* disponíveis existe uma gama que permite explorar o uso de IA como o *Neural Network AI* e o *AI Behavior*, por exemplo, que respectivamente possuem o propósito de criação de uma inteligência artificial de jogo e comportamentos dinâmicos dos personagens.

A Unity 3D possui três linhas de distribuição, a mesma pode ser utilizada de forma gratuita para empresas de baixo faturamento (menor que \$100.000). A Versão Pro pode ser utilizada pelo período de um mês e nessa versão os recursos (*Memory Profiler*, efeitos na tela pós-processamento, *Occlusion Culling*, entre outros) do sistema estão todos disponíveis. A licença da versão paga possui um custo de USD de \$1.500 ou \$75/mês (plano de 12 meses) (UNITY, 2014c).

2.4.2 Aperfeiçoamento de Reações Comportamentais de Non-Player Character (NPC) no Jogo Doom

Mattedi (2007) realizou um aprimoramento do comportamento dos oponentes enfrentados no jogo Doom, com o propósito de incrementar a dificuldade vivenciada na experiência de jogo pelo jogador. Foram realizadas modificações que alteraram a jogabilidade do mesmo, que minimizaram o problema dos tiros cruzados entre os NPCs.

Inicialmente foram constatadas algumas situações do jogo em que alterações de código poderiam alterar a jogabilidade do jogo Doom de forma que dificultasse o mesmo. Dentre as situações identificadas, duas foram selecionadas:

- a) o NPC nunca foge do jogador;
- b) os NPCs não verificam a presença de outro NPC na linha de fogo.

Realizadas as alterações, constatou-se que o jogo passou a fornecer NPCs que não apenas atiravam de forma cega no oponente (jogador), mas também tentavam se preservar, resultando em NPCs que demoravam mais para morrer que no original. Ao final, para o jogador não houve mudanças significativas na jogabilidade, mas as alterações ampliaram as possibilidades na forma de jogar.

[@@ Dalton: esse cara ainda é um bom correlato?]

2.4.3 Massive

O *Multiple Agent Simulation System In Virtual Environment* (MASSIVE) é um simulador de multi-agentes que possui Animação Comportamental e pode produzir uma simulação com a quantidade de agentes que for necessária. Com agentes bem simples, milhões podem ser executados em apenas um passo. Um grande número de agentes

complexos tais como típicos agentes humanoides, podem ser feitos em múltiplos passos, em grupos de cerca de 100.000 agentes por vez, com cada grupo subsequente capaz de ver e reagir com os grupos previamente simulados (MASSIVE, 2014b).

Originalmente o MASSIVE foi desenvolvido para o uso de Peter Jackson⁴ na trilogia O Senhor dos Anéis, porém posteriormente, a empresa Massive Software foi criada para trazer esta tecnologia para o cinema e produções de televisão ao redor do mundo. Desde então, o MASSIVE tornou-se o líder de software para multidão, relacionando efeitos visuais e animação de personagens autônomos.

Em 2004, Stephen Regelous (fundador e CEO da Massive Software) recebeu o prêmio *Academy Award for Scientific and Engineering Achievement*, pela concepção e desenvolvimento do MASSIVE, um sistema inovador utilizando agentes autônomos conduzidos por inteligência artificial para gerar animação. Stephen concebeu os conceitos básicos do software no início dos anos 90, depois de estudar sistemas de agentes baseado em Vida Artificial e desenvolveu o MASSIVE anos mais tarde para trilogia do O Senhor dos Anéis. Na produção desta trilogia o uso do MASSIVE transformou a expectativa da audiência em épicas multidões e cenas de batalhas no filme e na televisão (MASSIVE, 2014a).

⁴ Premiado roteirista, cineasta e produtor de filmes neozelandês.

3 DESENVOLVIMENTO

São abordadas nesse capítulo as etapas do desenvolvimento do módulo de Animação Comportamental, a extensão do editor e do motor de jogos desenvolvido por Harbs (2013), assim como as simulações de testes e de demonstração. São abordados os principais requisitos, a especificação, a implementação e os resultados e discussão.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O módulo de Animação Comportamental deverá:

- a) adicionar funcionalidades que permitam gerar animações comportamentais para os personagens (Requisito Funcional - RF):
 - percepção do ambiente,
 - raciocínio baseado nas percepções,
 - execução da ação determinada pelo raciocínio;
- b) ser desenvolvido de forma fortemente desacoplada (Requisito Não Funcional - RNF);
- c) proporcionar o controle mínimo da percepção, raciocínio e atuação do personagem (RNF);
- d) utilizar um dos modelos clássicos da IA (reativo, rede neural, entre outros) para o raciocínio dos personagens, utilizando um módulo de terceiros (RNF);
- e) ser compatível com os mesmos navegadores que o motor de jogos desenvolvido por Harbs (2013) (RNF);
- f) ser desenvolvido com HTML5 e JavaScript (RNF).

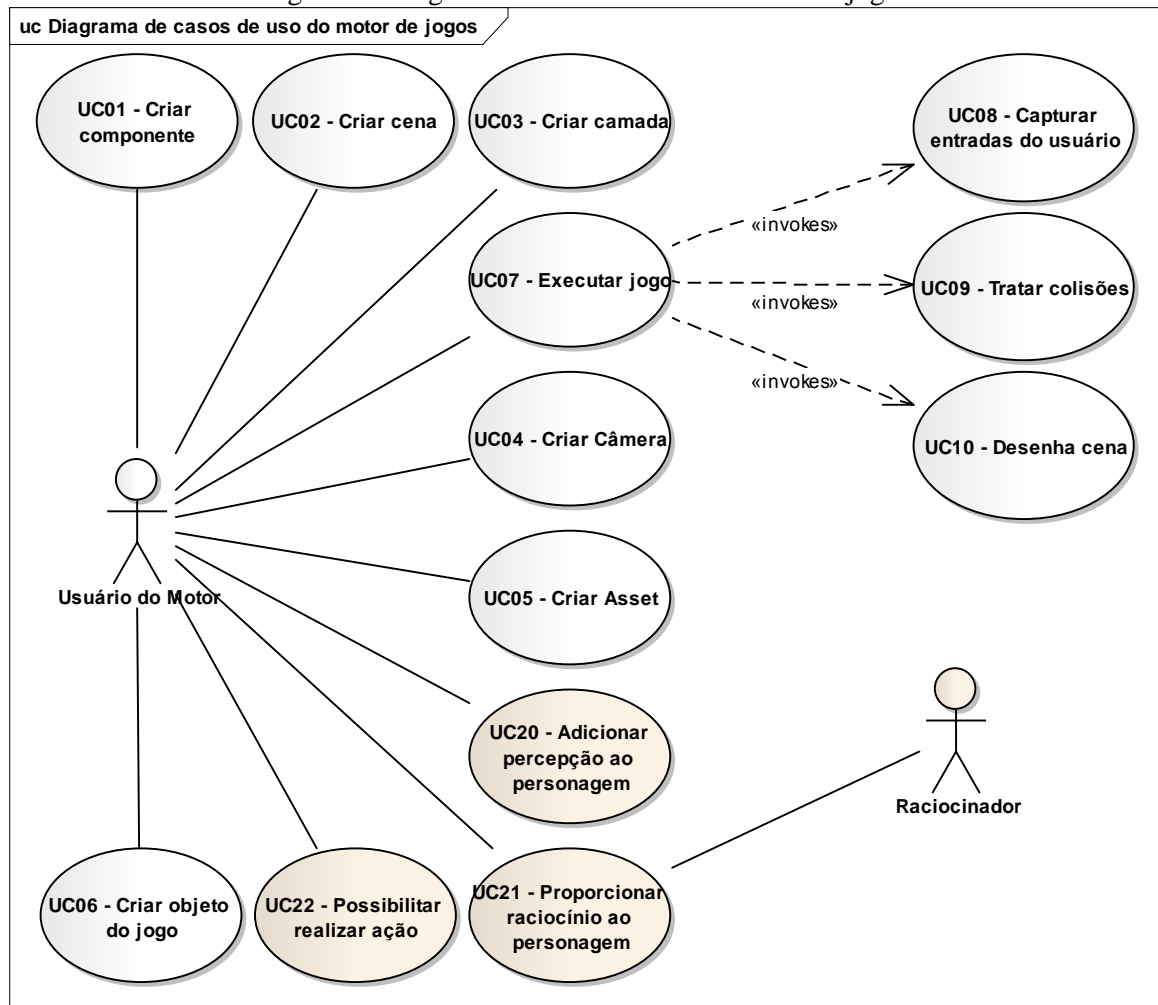
3.2 ESPECIFICAÇÃO

Para realizar a especificação do desenvolvimento do módulo de Animação Comportamental, foram criados diagramas de caso de uso, de pacote, de classe e de sequencia, com a ferramenta Enterprise Architect e utilizado diagramas da *Unified Modeling Language* (UML).

3.2.1 Diagrama de casos de uso do motor de jogos

Conforme pode ser visto na Figura 1, com base no requisito funcional do módulo desenvolvido, três novos casos de uso foram criados para o ator *Usuário do motor* (UC20, UC21 e UC22) e nenhum dos casos de uso desenvolvidos por Harbs (2013) teve a necessidade de serem modificados.

Figura 1 - Diagrama de casos de uso do motor de jogos



Fonte: estendido de Harbs (2013, p.25).

No Quadro 1 é apresentado o detalhamento do primeiro caso de uso operado pelo ator *Usuário do Motor* em virtude do módulo desenvolvido.

Quadro 1 - Caso de uso UC20

UC20 - Adicionar percepção ao personagem	
Descrição	Permite que o <i>Usuário do Motor</i> proporcione uma percepção do ambiente a um objeto do jogo.
Cenário Principal	<ol style="list-style-type: none"> 1. <i>Usuário do Motor</i> cria a definição de um componente que estende do componente <i>PerceptionVisionComponent</i>. 2. O <i>Usuário do Motor</i> sobrescreve os métodos padrões do componente <i>PerceptionVisionComponent</i>. 3. O sistema permite a utilização do componente para proporcionar percepção da simulação a um objeto do jogo.
Pós-Condição	O <i>Usuário do Motor</i> possui um componente que associado a um objeto do jogo proporciona percepção da simulação.

No Quadro 2 é apresentado o detalhamento do segundo caso de uso operado pelo ator *Usuário do Motor* em virtude do módulo desenvolvido.

Quadro 2 - Caso de uso UC21

UC21 - Proporcionar raciocínio ao personagem	
Descrição	Permite que o Usuário do Motor identifique as percepções.
Cenário Principal	1. O objeto do jogo observando o UC20 recebe as percepções. 2. Usuário do Motor constrói mensagem enviada para Raciocinador. 3. Simulação invoca o UC24 do Raciocinador.
Pré-Condição	O objeto do jogo deve possuir uma instância de um componente que estende PerceptionVisionComponent.
Pós-Condição	Usuário do Motor capaz de construir uma mensagem para interpretação no Raciocinador baseada na percepção recebida.

No Quadro 3 é apresentado o detalhamento do último caso de uso operado pelo ator Usuário do Motor em virtude do módulo desenvolvido.

Quadro 3 - Caso de uso UC22

UC22 - Possibilitar realizar ação	
Descrição	Permite que o Usuário do Motor execute ações raciocinadas.
Cenário Principal	1. O Raciocinador executa o UC24. 2. Objeto do jogo recebe a mensagem com ação determinada. 3. Usuário do motor identifica ação determinada.
Pré-Condição	O objeto do jogo deve possuir uma instância de um componente que estende PerceptionVisionComponent.
Pós-Condição	Usuário do Motor capaz de interpretar e executar a ação determinada.

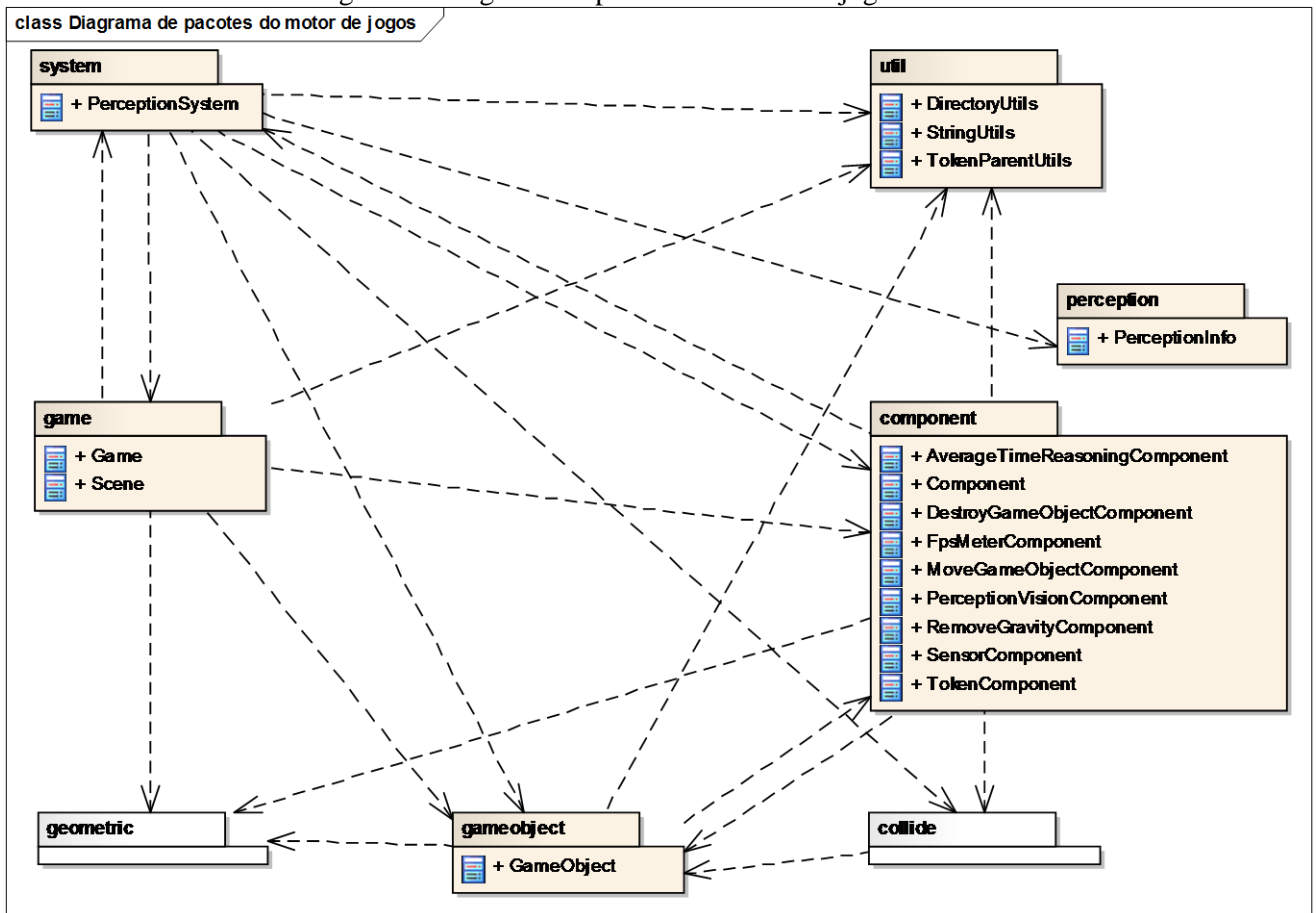
3.2.2 Diagramas de pacotes do motor de jogos

Nessa seção são descritos os novos pacotes e as novas classes adicionadas ao motor de jogos. Na Figura 2 além dos pacotes e das classes criadas, são evidenciadas as classes desenvolvidas por Harbs (2013) que sofreram alterações durante a criação do módulo. Vale ressaltar que as classes não mencionadas foram mantidas e não sofreram nenhuma alteração.

[@@ Dalton: criar anexo com os principais diagramas do Harbs assim como Nunes fez com Montibeler?]

[@@ annot: revisar todas as chamadas dos diagramas, onde todos devem ser evidencias/deve-se deixar claro que estão sendo apresentadas apenas as novas classes/métodos ou os editados...]

Figura 2 - Diagrama de pacotes do motor de jogos



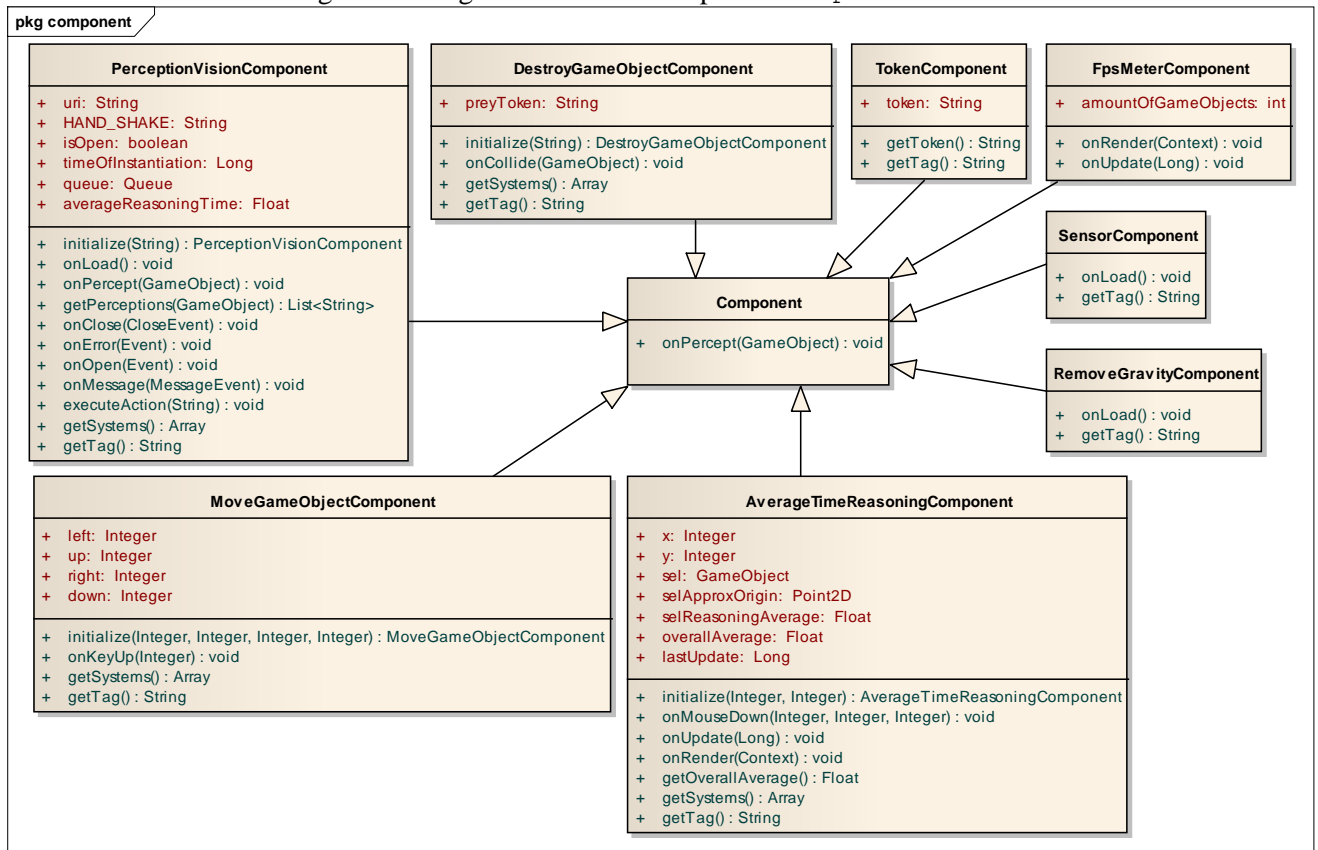
Fonte: estendido de Harbs (2013, p.26).

3.2.2.1 Pacote `component`

Na Figura 3 é apresentado o pacote `component`, composto por componentes que já existiam e tiveram modificações e por novos componentes desenvolvidos.

Na classe `Component`, foi adicionado o método `onPercept(GameObject)`, possibilitando que ao sobrescrever esse evento um componente possa tratar as percepções do objeto a qual o mesmo está associado. A classe `FpsMeterComponent`, além de medir a quantidade de vezes que o jogo está sendo desenhado, também está apresentando a quantidade de objetos existentes na tela do jogo. A classe `RemoveGravityComponent` define um componente que altera o comportamento da camada a qual a mesma está associada, removendo a gravidade existente ao ser carregada.

A classe `AverageTimeReasoningComponent` define um componente que apresenta em tela o tempo médio de processamento do raciocínio dos agentes do jogo. A classe `TokenComponent` define um componente que permite relacionar um token de identificação para a instância (`Scene`, `Layer`, `GameObject`) a qual está relacionada.

Figura 3 - Diagrama de classes do pacote `component`

Fonte: estendido de Harbs (2013, p.27).

A classe `DestroyGameObjectComponent` define um componente que adiciona o comportamento de destruição do objeto colidido, ao objeto que o componente está relacionado, quando o objeto com que o mesmo colidiu possuir um token de identificação igual ao recebido no método `initialize(String)`. A classe `MoveGameObjectComponent` define um componente que permite movimentar um personagem na tela através de *callback* de teclado. A classe `SensorComponent` define um componente que designa o comportamento de sensor ao objeto a qual está relacionado. Esse componente, muda o comportamento do objeto em que está relacionado, fazendo com que o mesmo passe a propagar percepções do mundo para o motor de jogos, ao invés de colisões físicas.

A classe `PerceptionVisionComponent` define um componente que utilizando *WebSocket*, adiciona uma estrutura de comunicação com o raciocinador que realizará o processamento da mente do agente. O componente adiciona o comportamento de sensor ao objeto em que está relacionado, assim como o componente `SensorComponent` sendo responsável por transmitir e receber as mensagens para o raciocinador. Para utilizá-lo é necessário criar um novo componente que estenda de `PerceptionVisionComponent` e sobrescrever os métodos `getPerceptions(GameObject)` e `executeAction(String)`. O primeiro método é responsável por recuperar e retornar ao componente todas as percepções

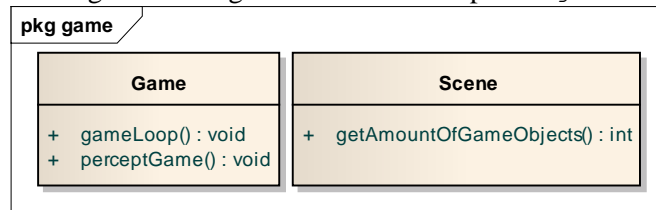
que serão enviadas para o raciocinador e o segundo para interpretar e executar a ação determinada pelo mesmo.

3.2.2.2 Pacote `game`

A Figura 4 ilustra as classes do pacote `game` que tiveram métodos criados ou que sofreram modificações.

[@@ Daton: capobianco não gostou dessa disposição, de demonstrar apenas as novas classes métodos. Sugeriu adicionar o original, fazendo alguma marcação (talvez um circulo) no que foi adicionado/alterado. Ou deixar mais clara descrição por diagrama demonstrado.]
 [@@ annot: uma sugestão foi alterar a legenda para “métodos inseridos ou modificados na classe do pacote `game`”]

Figura 4 - Diagrama de classes do pacote `game`



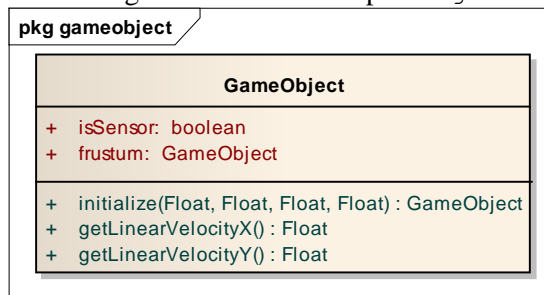
Fonte: estendido de Harbs (2013, p.29).

Na classe `Game` foi criado o método `perceptGame()`, responsável por disparar o resolvidor das percepções. No método `gameLoop()` foi adicionada uma chamada para o método `perceptGame()` fazendo com que durante o *loop* do jogo as percepções sejam disparadas. Na classe `Scene` foi adicionado o método `getAmountOfGameObjects()` que retorna a quantidade de objetos existentes na cena.

3.2.2.3 Pacote `gameobject`

No pacote `gameobject` apenas a classe `GameObject` sofreu alterações, conforme visto na Figura 5.

Figura 5 - Diagrama de classes do pacote `gameobject`



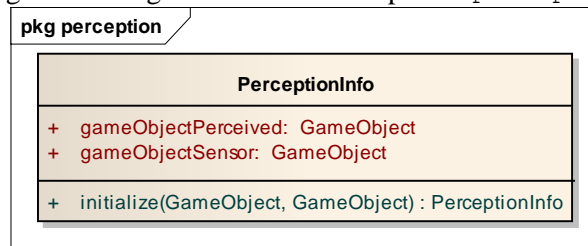
Fonte: estendido de Harbs (2013, p.28).

Na classe `GameObject` foram adicionadas duas novas propriedades, `isSensor` responsável por identificar se um objeto é um sensor e `frustum`, propriedade responsável por referenciar o objeto que representa a objeto do sensor responsável por propagar suas percepções. Também foram adicionados dois novos métodos, `getLinearVelocityX()` e `getLinearVelocityY()` responsáveis por recuperar a velocidade do objeto respectivamente no eixo X (horizontal) e no eixo Y (vertical).

3.2.2.4 Pacote `perception`

A Figura 6 demonstra o novo pacote adicionado ao motor de jogos, nomeado de `perception`, o mesmo possui a classe `PerceptionInfo` responsável por armazenar a ocorrência de uma percepção, registrando o objeto sensor e o objeto percebido.

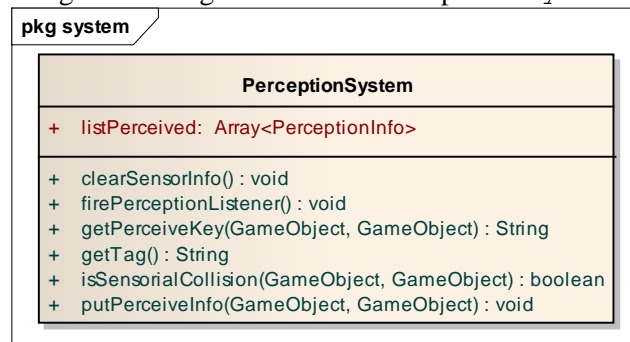
Figura 6 - Diagrama de classes do pacote `perception`



3.2.2.5 Pacote `system`

A Figura 7 demonstra a nova classe disparadora de eventos do motor de jogos adicionada ao pacote `system`.

Figura 7 - Diagrama de classes do pacote `system`



Fonte: estendido de Harbs (2013, p.30).

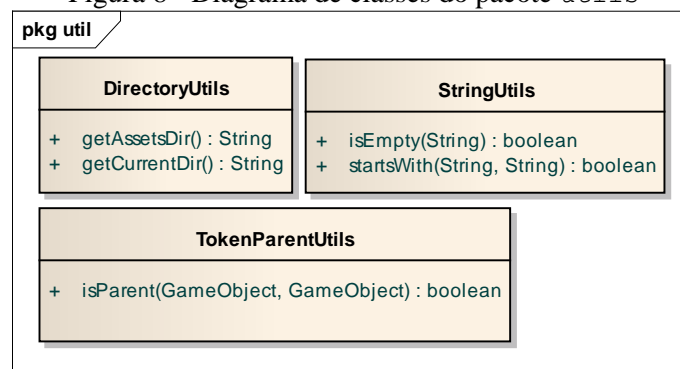
Compete a classe `PerceptionSystem`, identificar se em uma colisão entre objetos ocorreu alguma percepção. No método `isSensorialCollision(GameObject, GameObject)` são identificados os objetos sensores da colisão e dependendo da quantidade de sensores da colisão três caminhos pode considerados. No caso de nenhum objeto ser identificado como um sensor, essa colisão é ignorada para o contexto de percepção, porém se um dos objetos

identificados for um sensor, a percepção é registrada e posteriormente através do método `firePerceptionListener()` todos os eventos de percepção são disparados. A última possibilidade é a identificação de dois sensores, nessa situação o evento é considerado uma percepção, porém não é feito nenhum registro devido o fato de nenhum dos objetos serem perceptíveis uns aos outros.

3.2.2.6 Pacote `util`

Conforme demonstrado na Figura 8, pode-se identificar que duas novas classes foram adicionadas ao pacote `utils` e dois novos métodos foram adicionados a classe `StringUtils`.

Figura 8 - Diagrama de classes do pacote `utils`



Fonte: estendido de Harbs (2013, p.31).

Foram adicionados na classe `StringUtils` os métodos `isEmpty(String)` e `startsWith(String, String)`, o primeiro método é responsável por identificar se uma *String* está vazia e segundo se uma *String* é iniciada com outra.

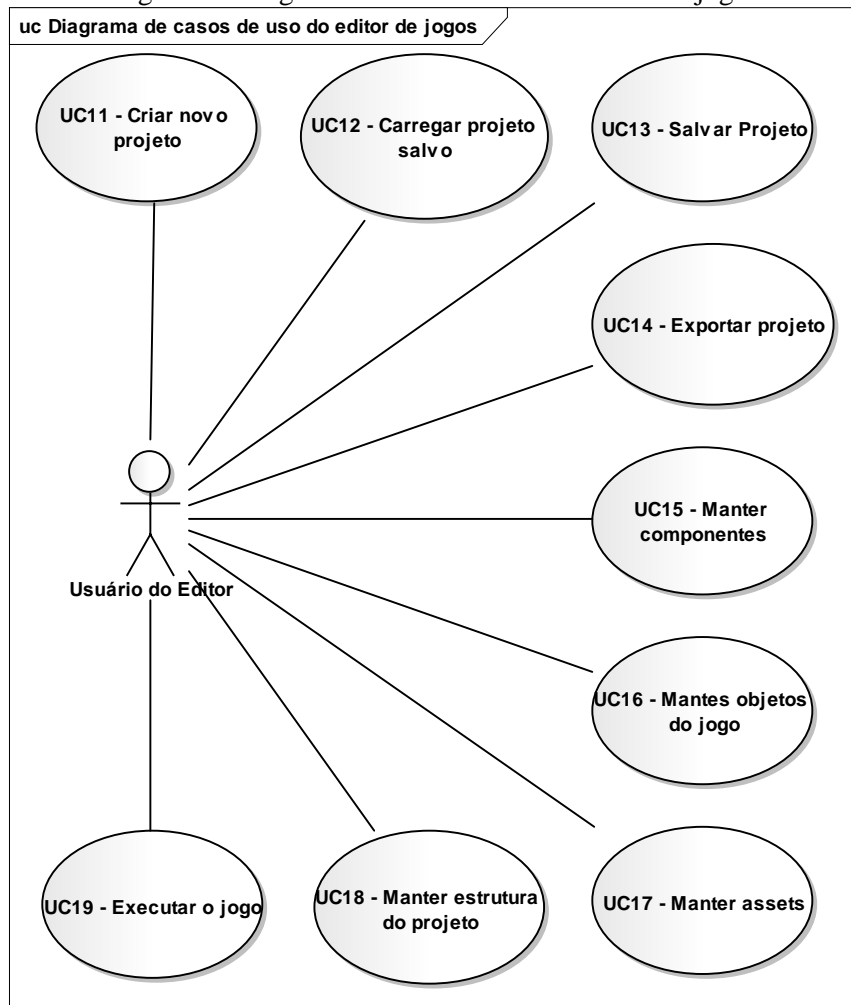
Na classe `TokenParentUtils`, o método `isParent(GameObject, GameObject)` identifica se ambos objetos possuem o adicionado o componente `TokenComponent` e se os mesmo possuem o mesmo inicio. A classe `DirectoryUtils` é responsável por retornar a URL atual do jogo através do método `getCurrentDir()` e o diretório dos *assets* disponíveis através do método `getAssetsDir()`.

3.2.3 Diagrama de casos de uso do editor de jogos

Nessa seção são descritas as alterações realizadas no editor de jogos desenvolvido por Harbs (2013). As classes não mencionadas foram mantidas e não sofreram nenhuma alteração. Conforme pode ser visto na Figura 9, todas as funcionalidades do editor foram mantidas, nenhuma funcionalidade foi adicionada ao editor e nenhum dos casos de uso existentes sofreram alterações.

[@@ annot: cuidar com o presente e o futuro... futuro apenas nas extensões]

Figura 9 - Diagrama de casos de uso do editor de jogos

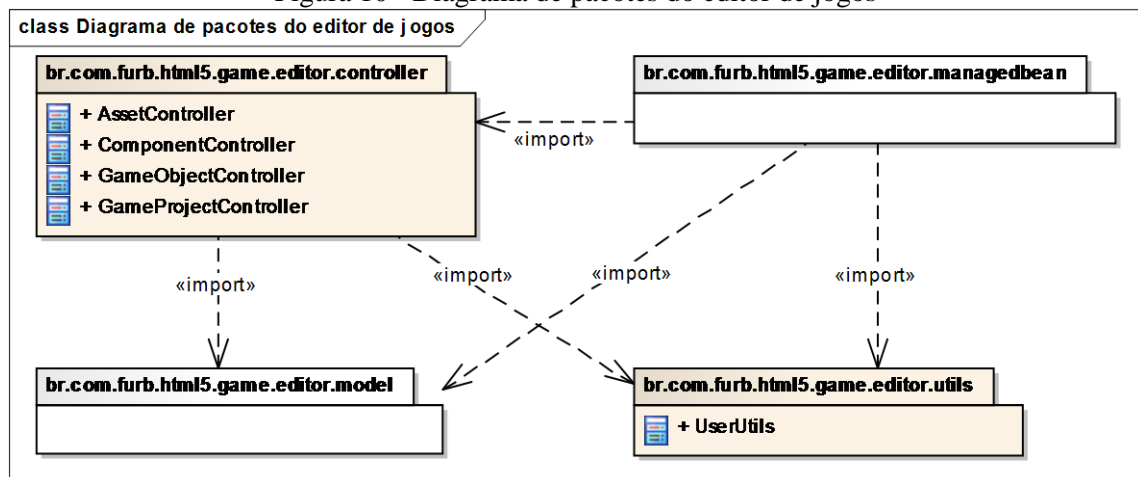


Fonte: estendido de Harbs (2013, p.32).

3.2.4 Diagramas de classes do editor de jogos

Nessa seção será descrita a nova classe adicionada ao editor de jogos e as classes do mesmo que sofreram alterações. Como pode ser visto na Figura 10, nenhum novo pacote foi adicionado ao editor.

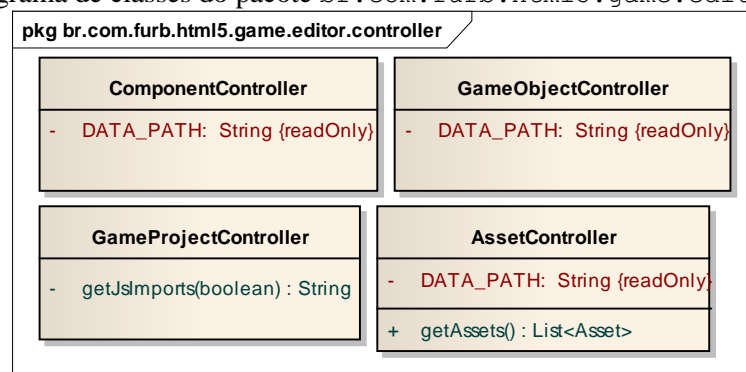
Figura 10 - Diagrama de pacotes do editor de jogos



Fonte: estendido de Harbs (2013, p.33).

3.2.4.1 Pacote `br.com.furb.html5.game.editor.controller`

Na Figura 11 são demonstrados os ajustes realizados no pacote `br.com.furb.html5.game.editor.controller` para funcionamento do editor em qualquer plataforma e da inclusão dos novos *imports* ao jogo exportado.

Figura 11 - Diagrama de classes do pacote `br.com.furb.html5.game.editor.controller`

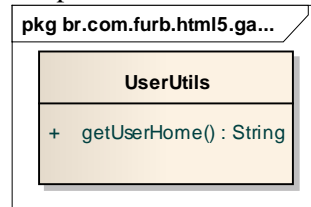
Fonte: estendido de Harbs (2013, p.34).

Nas classes `AssetController`, `ComponentController` e `GameObjectController` ajustado o caminho das pastas do usuário que armazenam as classes dos novos componentes, objetos do jogo e dos recursos (áudio, imagem, mente) utilizados pelo motor e adicionados pelo usuário através do editor de jogos. Anteriormente os nomes estavam fixos a um usuário específico e foi alterado para sempre utilizar a pasta de usuário do usuário corrente. Alterada classe `GameProjectController` para concatenar os *imports* das novas classes adicionadas ao motor de jogos ao realizar a exportação do HTML necessário para a execução do jogo.

3.2.4.2 Pacote `br.com.furb.html5.game.editor.model`

Conforme visto na Figura 12, a classe utilitária `UserUtils` foi adicionada ao pacote `br.com.furb.html5.game.editor.model`. Compete ao método `getUserHome()` recuperar o diretório da pasta do usuário.

Figura 12 - Diagrama de classes do pacote `br.com.furb.html5.game.editor.utils`

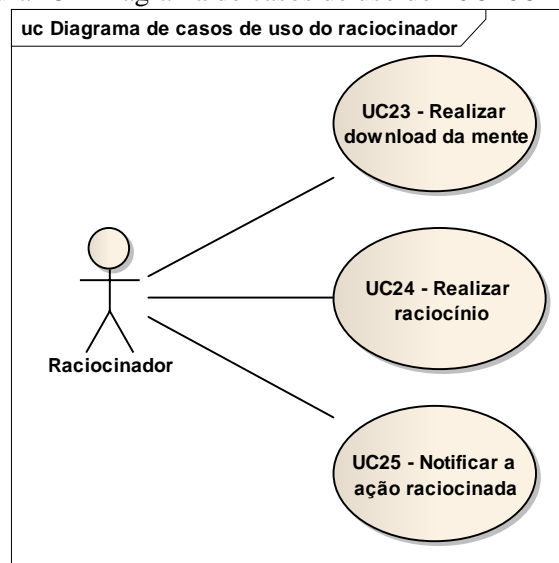


Fonte: estendido de Harbs (2013, p.37).

3.2.5 Diagrama de casos de uso do raciocinador

A Figura 13 demonstra os três casos de uso criados para o ator `Raciocinador` (UC23, UC24 e UC25).

Figura 13 - Diagrama de casos de uso do `Raciocinador`



No Quadro 4 é apresentado o detalhamento do primeiro caso de uso operado pelo ator `Raciocinador`.

Quadro 4 - Caso de uso UC23

UC23 - Realizar download da mente	
Descrição	Permite que o <i>Raciocinador</i> realize download da mente do agente.
Cenário Principal	<ol style="list-style-type: none"> 1. Servidor de Aplicação que disponibiliza o <i>Raciocinador</i> recebe conexão da <i>WebSocket</i> criada pelo motor de jogos. 2. O sistema realiza <i>download</i> do arquivo que representa uma mente para a simulação. 3. O sistema prepara a mente para realização de raciocínio da simulação. 4. O Servidor de aplicação retorna a definição de conexão estabelecida para o motor de jogos.
Pós-Condição	<i>Raciocinador</i> preparado para executar o UC24.

No Quadro 5 é apresentado o detalhamento do segundo caso de uso operado pelo ator *Raciocinador*.

Quadro 5 - Caso de uso UC24

UC24 - Realizar raciocínio	
Descrição	Permite que o <i>Raciocinador</i> realize o processamento do raciocínio.
Cenário Principal	<ol style="list-style-type: none"> 1. A simulação envia mensagem com as percepções para o Servidor de aplicação. 2. Servidor de Aplicação transmite a mensagem para o <i>Raciocinador</i>. 3. Utilizando a mente preparada no UC23, o <i>Raciocinador</i> interpreta a mente com base nas percepções recebidas. 4. <i>Raciocinador</i> invoca o UC25 para retorna a ação determinada.
Pós-Condição	<i>Raciocinador</i> realiza raciocínio da mente do agente com base nas percepções recebidas.

No Quadro 6 é apresentado o detalhamento do último caso de uso operado pelo ator *Raciocinador*.

Quadro 6 - Caso de uso UC25

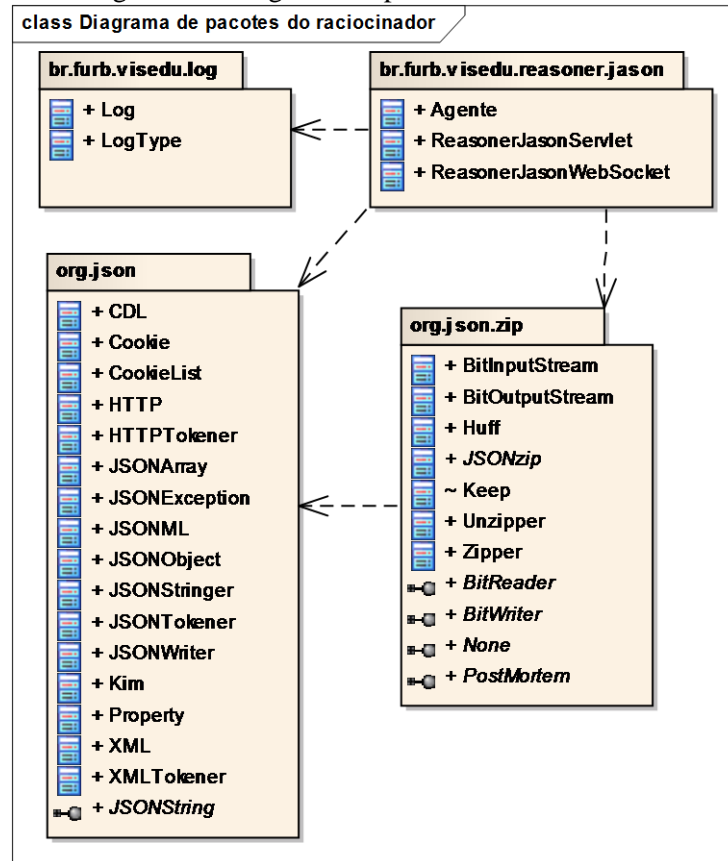
UC25 - Notificar a ação raciocinada	
Descrição	<i>Raciocinador</i> notifica a simulação com a ação determinada.
Cenário Principal	<ol style="list-style-type: none"> 1. <i>Raciocinador</i> invoca UC24. 2. <i>Raciocinador</i> transmite a ação determinada para o servidor de aplicação. 3. Servidor de Aplicação transmite a ação raciocinada para simulação.
Pós-Condição	<i>Raciocinador</i> realiza a notificação da ação raciocinada para simulação.

3.2.6 Diagrama de pacotes do raciocinador

Nessa seção, são descritos os pacotes e as classes criadas no desenvolvimento do raciocinador do módulo de Animação Comportamental, para o motor de jogos. Na Figura 14 pode-se observar que o raciocinador é composto por quatro pacotes, cujos pacotes

denominados `org.json` e `org.json.zip` tratam-se de uma implementação de referência de um pacote JSON⁵ em Java.

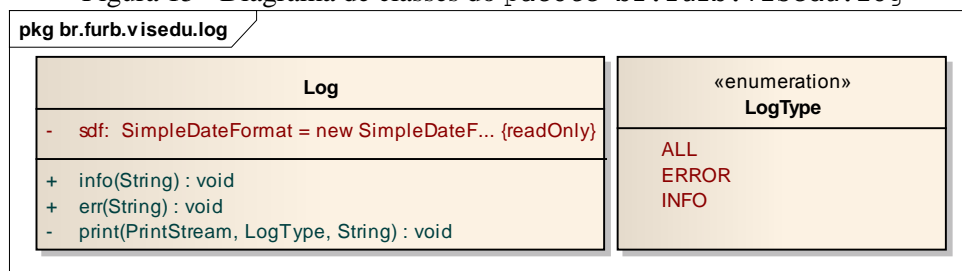
Figura 14 - Diagrama de pacotes do raciocinador



3.2.6.1 Pacote `br.furb.visedu.log`

O primeiro pacote do raciocinador é denominado `br.furb.visedu.log` (Figura 15) e possui as classes responsáveis por gerenciar a parte de logs do raciocinador.

Figura 15 - Diagrama de classes do pacote `br.furb.visedu.log`



A classe `Log` é responsável por gerenciar a disposição dos logs gerado pelo raciocinador, através de argumentos da máquina virtual Java (*VM arguments*), é possível demonstrar todos os tipos de logs (`-Dprintall=true`), demonstrar apenas as informações básicas (`-Dprintinf=true`) ou somente os erros (`-Dprinterr=true`). O enumerador

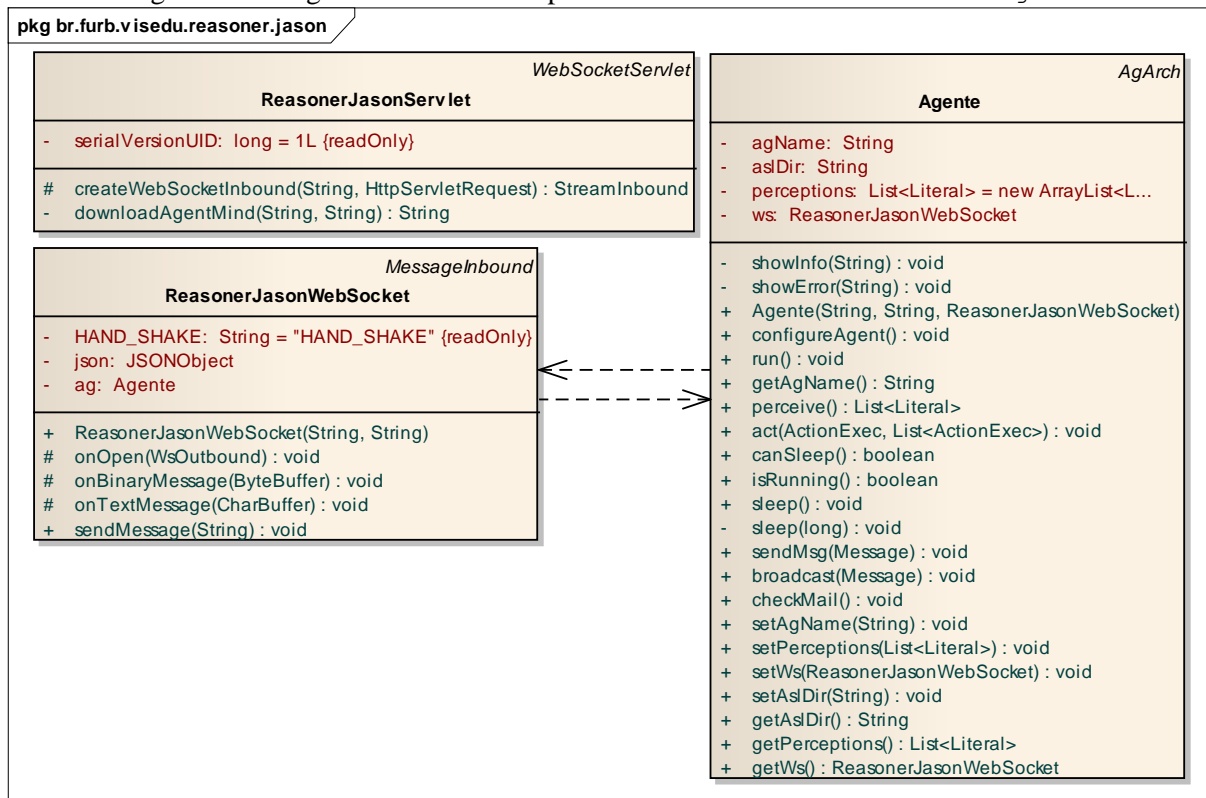
⁵ Informações sobre a licença disponíveis em <http://www.json.org/license.html>

LogType é responsável por definir os tipos de log registrados pelo raciocinador e proporcionar o gerenciamento dos mesmos pela classe Log.

3.2.6.2 Pacote br.furb.visedu.reasoner.jason

O segundo pacote do raciocinador é denominado br.furb.visedu.reasoner.jason e possui as classes responsáveis por gerenciar o raciocinador Jason, conforme pode ser visto na Figura 16.

Figura 16 - Diagrama de classes do pacote br.furb.visedu.reasoner.jason



A classe ReasonerJasonServlet estende a classe WebSocketServlet e ao estabelecer a conexão via socket com a simulação, tem por responsabilidade realizar o download do asset relativo a mente do agente. A classe ReasonerJasonWebSocket estende a classe MessageInbound e tem por responsabilidade gerenciar o canal de comunicação com o agente em execução na simulação. Além dessa funcionalidade, compete a classe ReasonerJasonWebSocket gerenciar a instância da classe Agente, realizando a configuração, o registro das percepções identificadas na simulação e realizando a execução da mente do agente no raciocinador Jason. A classe Agente estende a classe AgArch e compete a ela ser a camada entre as informações coletadas na simulação (mente, percepções) e o interpretador Jason. Quando uma instancia da classe Agente identifica que uma ação foi determinada durante um ciclo de raciocínio (com base na mente e nas percepções recuperadas de um

agente da simulação), utilizando a classe `ReasonerJasonWebSocket` é enviada para a simulação uma notificação com a ação determinada.

3.2.7 Diagrama de Sequência

Nessa seção será apresentado um diagrama de sequência que descreve o *pipeline* de percepção proporcionado pelo módulo de Animação Comportamental desenvolvido, ao *loop* principal de um jogo gerenciado pelo motor de jogos desenvolvido por Harbs (2013).

Conforme disposto no diagrama da Figura 17, após realizar o carregamento de todos os componentes associados ao jogo e realizar a configuração da cena a ser apresentada, é iniciado o *loop* do jogo. Anteriormente a *loop* do jogo era dividido em três passos:

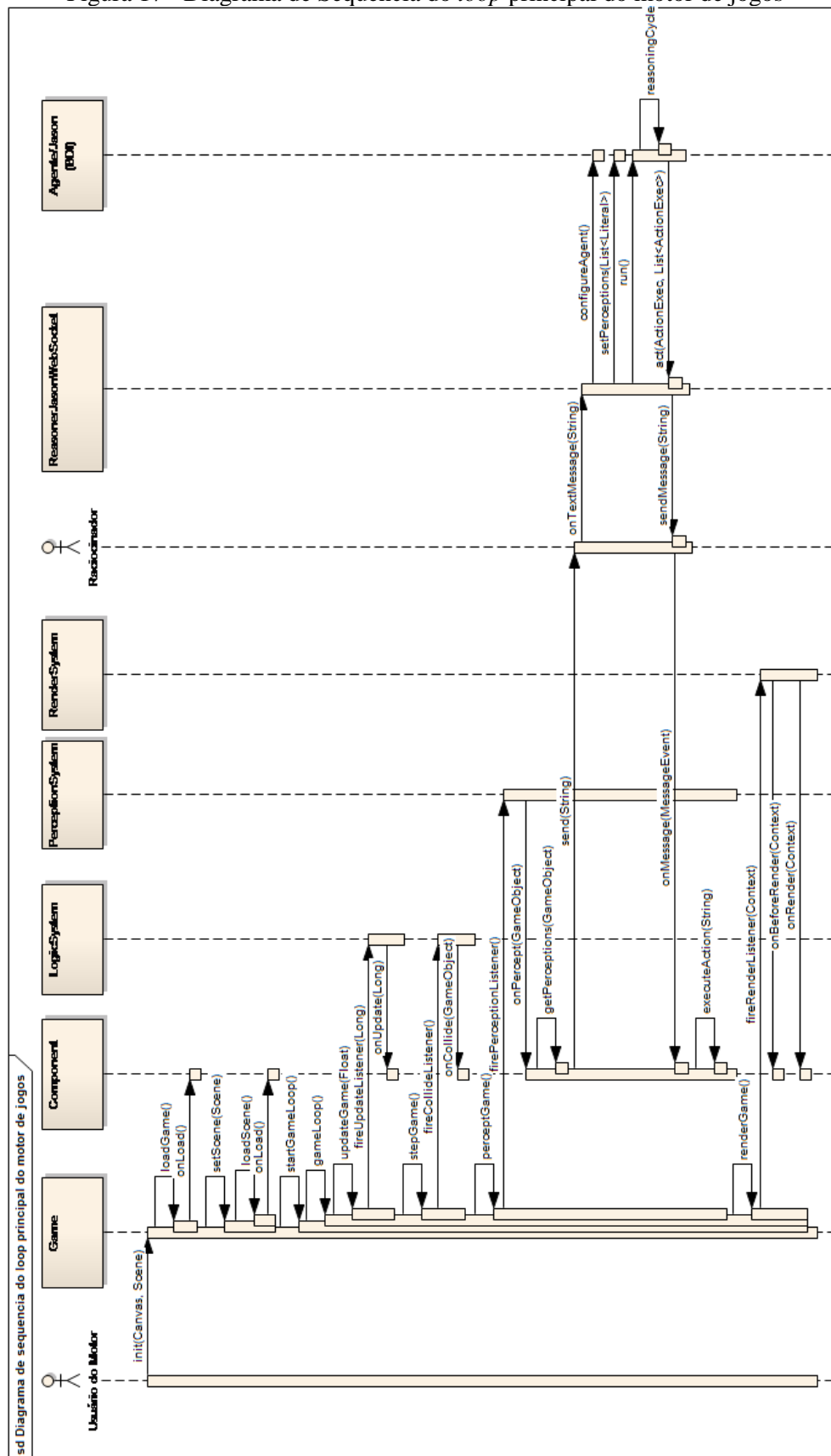
- a) atualização dos componentes do jogo;
- b) verificações e propagação das colisões do jogo;
- c) atualização do desenho da cena.

Com o desenvolvimento do módulo um novo passo foi adicionado no *loop* do jogo, responsável por verificar e propagar as percepções da simulação. O passo é executado entre as verificações e propagação das colisões da simulação e a atualização do desenho da cena assumindo assim o terceiro passo de um *loop* agora formado por quatro passos.

No novo estágio do *loop*, através do método `firePerceptionListener()` é disparado o evento de percepção para os objetos que tiveram percepções registradas pelo método `isSensorialCollision()`. É importante salientar que diferentemente dos outros três estágios do *loop*, o processo de propagação das percepções para o raciocinador, até a eventual resposta do mesmo com a determinação da ação a ser executada, é um processo assíncrono.

Fazendo um contraponto ao ciclo originalmente desenvolvido por Harbs (2013), o diagrama demonstra a lógica interna da execução dos métodos para realização de um ciclo de percepção, raciocínio e atuação no ambiente. Demonstrando todas as classes e seus respectivos papéis na geração da animação comportamental.

Figura 17 - Diagrama de Sequencia do *loop* principal do motor de jogos



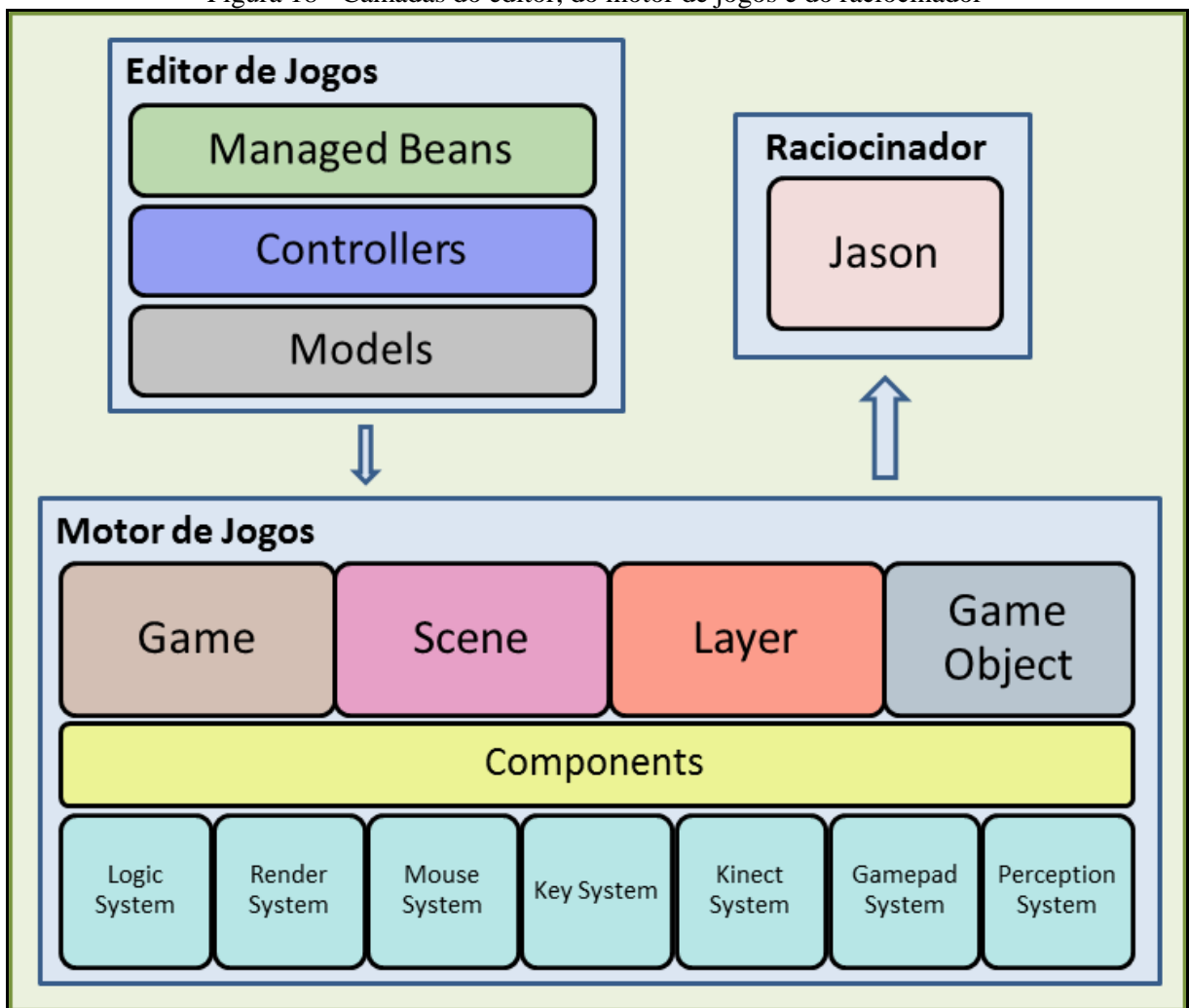
Fonte: estendido de Harbs (2013, p.38).

Outra observação do sobre o diagrama de sequencia é método `send(String)`, o mesmo pertence a `WebSocket` gerenciada pelo componente `PerceptionVisionComponent`. O método é invocado ao final da execução do método `onPercept(gameObjectPerceived)` do mesmo componente.

3.2.8 Diagrama de Arquitetura

A Figura 18 apresenta o diagrama das camadas da arquitetura do editor, do motor de jogos e do raciocinador.

Figura 18 - Camadas do editor, do motor de jogos e do raciocinador



Fonte: estendido de Harbs (2013, p.38).

A especificação da arquitetura do editor e do motor de jogos foram mantidas, conforme a proposta de manter o desacoplamento desenvolvido.

A arquitetura do motor de jogos foi especificada de maneira orientada a componentes. A camada **GAME** se comunica com a camada abaixo dela chamada de **SCENE**. A camada **SCENE** se comunica com a camada **LAYERS**, que por sua vez, se comunica com a camada **GAME OBJECTS**. As camadas **LOGIC SYSTEM**, **RENDER SYSTEM**, **MOUSE SYSTEM**, **KEY SYSTEM**, **KINECT SYSTEM** e **GAMEPAD SYSTEM** fazem interfaces com dispositivos externos e se comunicam

com as demais camadas [sic] através da camada COMPONENTS.

As camadas do editor de jogos estão divididas em MANAGED BEANS, CONTROLLERS e MODELS. Esta divisão foi adotada seguindo o padrão *Model View Controller* (MVC). (Harbs, 2013, p 39-40).

Ao implementar o módulo de Animação Comportamental, foi adicionada uma nova camada a estrutura do motor de jogos, denominada “PERCEPTION SYSTEM”, com o propósito de atuar como interface de comunicação com o raciocinador. O raciocinador, por sua vez, trata-se de um adaptador responsável por preparar as informações do ambiente externo para interpretação de um modelo mental, no caso o Jason.

3.3 IMPLEMENTAÇÃO

Nesta seção são apresentadas as técnicas e as ferramentas utilizadas na implementação do módulo de Animação Comportamental e na edição do editor e do motor de jogos do trabalho estendido. Também são abordadas as principais classes e rotinas desenvolvidas.

3.3.1 Técnicas e ferramentas utilizadas

Para o desenvolvimento do módulo de Animação Comportamental, foi utilizada a linguagem de programação Java na versão 7 e como ambiente foi utilizado o Eclipse IDE for Java EE Developers utilizando os *plug-ins* JBoss Tools (JBOSS, 2014) e Jasonide (JASON, 2014d). Para criação do raciocinador foi utilizado o interpretador Jason 1.4.1 (JASON, 2014a) e como servidor de aplicação para o mesmo foi utilizado o Apache Tomcat 7.0.55 (APACHE TOMCAT, 2014).

Para edição e execução do editor de jogos desenvolvido por Harbs (2013) também foram utilizados a linguagem de programação Java e o Eclipse IDE for Java EE Developers como ambiente de desenvolvimento e como servidor de aplicação o JBoss 6.1.0 (JBOSS, 2014b). Para edição do motor de jogos foi utilizado o ambiente Sublime Text 2 (SUBLIME, 2014).

Para realização do desenvolvimento do módulo de Animação Comportamental foram utilizados os navegadores Google Chrome 38.0.2125.104m, Internet Explorer 11.0.9600.17351, Mozilla Firefox 33.0 e Opera 25.0.1614.50, em um notebook Dell Vostro 3500 com sistema operacional Windows 8.1 64 bits, processador Intel Core I5 M480 2.66 Giga Hertz (Ghz) e 4 Gigabytes (GB) de memória RAM. Todo o gerenciamento dos códigos fonte foi realizado através do repositório remoto Git do Bitbucket e como ferramenta para realização das submissões foi utilizado o próprio ambiente de desenvolvimento Eclipse.

3.3.2 Módulo de Animação Comportamental

Nesta seção será descrita a implementação do módulo de Animação Comportamental, demonstrando suas rotinas e as principais alterações necessárias ao editor e ao motor de jogos.

3.3.2.1 Módulo de Raciocínio

O módulo de raciocínio, neste conceito será denominado de raciocinador. O raciocinador tem o objetivo ser um motor de raciocínio, através de um modelo mental ele determinará ações para personagens de uma simulação. Para criação do raciocinador optou-se por utilizar o interpretador Jason (JASON, 2014), pois além de ser uma ferramenta *Open Source* o interpretador proporciona integração dos agentes de seu Sistema Multiagente (SMA) a ambientes externos.

O Quadro 7 apresenta a classe responsável pela criação da *WebSocket* no raciocinador. Na linha 1, é definida a anotação `@WebServlet` contendo o valor da localização da *WebSocket*. Nas linhas 8 e 9, pode-se observar que ao criar a *WebSocket* o raciocinador necessariamente aguarda dois parâmetro, `agent` e `assetsDir`. O primeiro identifica o nome do agente que a *WebSocket* se encarregará de processar e o segundo o caminho (*path*) do arquivo que contem as definições da mente do agente.

Para criar um raciocinador que funciona-se forma independente da mente e das percepções a serem interpretadas, optou-se por realizar o *download* da mente do agente ao estabelecer a conexão. Na linha 15, ao invocar o método `downloadAgentMind(String, String)` a classe realiza o *download* do arquivo com as definições da mente do agente, persiste as definições em um arquivo temporário e retorna o diretório do mesmo. Posteriormente o arquivo relativo às definições da mente será interpretado pelo Jason com bases nas percepções recebidas.

Quadro 7 - Implementação da classe ReasonerJasonServlet

```

1  @WebServlet("/jason")
2  public class ReasonerJasonServlet extends WebSocketServlet {
3
4      private static final long serialVersionUID = 1L;
5
6      protected StreamInbound createWebSocketInbound(String
7          subProtocol, HttpServletRequest request) {
8          String agent = request.getParameter("agent");
9          String assetsDir = request.getParameter("assetsDir");
10         String aslDir = null;
11         if ( !assetsDir.contains( "html5-2d-game-editor" ) ) {
12             try {
13                 Log.info( String.format("Start download of"
14                     + " \"%s\" @ %s", agent, assetsDir) );
15                 aslDir = downloadAgentMind(agent, assetsDir);
16                 Log.info( String.format("Download complete of"
17                     + " \"%s\" @ %s", agent, assetsDir) );
18             } catch (IOException e) {
19                 Log.err( String.format("Download error. File:"
20                     + " \"%s\" @ %s", agent, assetsDir) );
21                 e.printStackTrace();
22             }
23         }
24         return new ReasonerJasonWebSocket( agent, aslDir );
25     }
26
27     private static String downloadAgentMind(String name, String dir)
28         throws IOException {
29         File tmpAsl = File.createTempFile(name, ".asl");
30         URL urlFile = new URL( String.format("%s%s.asl", dir,
31             name) );
32         ReadableByteChannel rbc =
33             Channels.newChannel(urlFile.openStream());
34         FileOutputStream fos = new FileOutputStream( tmpAsl );
35         fos.getChannel().transferFrom(rbc, 0, Long.MAX_VALUE);
36         return tmpAsl.getAbsolutePath();
37     }
38 }
39

```

O Quadro 8 apresenta a classe responsável por recuperar as percepções enviadas pelo ambiente e enviar para o SMA do Jason.

Na linha 8, pode-se observar que ao construir a classe `ReasonerJasonWebSocket`, o raciocinador cria uma instância de `Agente` armazenando referência do nome e do diretório da mente do agente para utilização futura ao executar o raciocínio. Também é armazenada a referência da `WebSocket` para proporcionar maior independência da parte do agente quando enviar a mensagem de resposta com a ação determinada a ser executada.

Quadro 8 - Implementação da classe ReasonerJasonWebSocket

```

1  public class ReasonerJasonWebSocket extends MessageInbound {
2
3      private static final String HAND_SHAKE = "HAND_SHAKE";
4      private JSONObject json;
5
6      private Agente ag;
7
8      public ReasonerJasonWebSocket(String agentName, String aslDir) {
9          Log.info("new Agent: " + agentName + " @ " + aslDir);
10         ag = new Agente(agentName, aslDir, this);
11     }
12
13     ...
14
15     @Override
16     protected void onTextMessage(CharBuffer msg) throws IOException {
17         Log.info("onTextMessage: " + msg);
18
19         if ( HAND_SHAKE.equalsIgnoreCase(msg.toString()) ) {
20             sendMessage( HAND_SHAKE );
21             return;
22         }
23
24         json = new JSONObject(msg.toString());
25         String sPerceptions = (String) json.get("perceptions");
26         JSONArray ja = new JSONArray(sPerceptions);
27
28         List<Literal> perceptions = new ArrayList<Literal>();
29         for (int i = 0; i < ja.length(); i++) {
30             perceptions.add( Literal.parseLiteral(
31                 ja.getString(i) ) );
32         }
33         ag.configureAgent();
34         ag.setPerceptions(perceptions);
35         ag.run();
36     }
37
38     public void sendMessage(String message) {
39         if ( !HAND_SHAKE.equalsIgnoreCase(message) ) {
40             json.put("action", message);
41             message = json.toString();
42         }
43         Log.info("sendMessage: " + message);
44         try {
45             getWsOutbound().writeTextMessage(
46                 CharBuffer.wrap(message) );
47         } catch (IOException e) {
48             e.printStackTrace();
49         }
50     }
51 }

```

Nas linhas 24, 25 e 26 pode-se identificar que o raciocinador traduz a mensagem recebida no formato texto para uma estrutura gerenciadora de informações no formato *JavaScript Object Notation* (JSON), possibilitando assim recuperar as percepções da mensagem enviada pelo ambiente. As percepções são iteradas, transformadas em literais compreensíveis para o interpretador Jason (`jason.asSyntax.Literal`). Em seguida (linha

33) é realiza a configuração do agente através do método `configureAgent()`, relacionando assim um agente do SMA com a mente que teve seu *download* realizado anteriormente quando a *WebSocket* foi estabelecida (linha 15 do Quadro 7). Na linha 37 a lista de percepções (em forma de `Literal`) é definida no agente, possibilitando que sejam interpretadas na execução do ciclo de raciocínio do Jason (linha 35).

Na linha 41, é demonstrado o método `sendMessage(String)` que se encarrega de encapsular a ação determinada no arquivo JSON (recebido anteriormente no método `onTextMessage(CharBuffer)`), atribuindo a mesma sob a *tag* `action`. Em seguida envia para simulação uma mensagem de uma representação em texto da estrutura JSON contendo a ação determinada.

No Quadro 9 é demonstrado o método invocado quando uma ação é determinada pelo Jason.

Quadro 9 - Implementação do método `act(ActionExec, List<ActionExec>)` da classe `Agente`

```
1  @Override
2  public void act(ActionExec action, List<ActionExec> feedback) {
3      showInfo("Agent " + getAgName() + ": doing: " +
4          action.getActionTerm());
5      getWs().sendMessage( action.getActionTerm().toString() );
6      action.setResult(true);
7      feedback.add(action);
8  }
```

Na linha 5, nota-se que a classe `Agente` recupera a instância de `ReasonerJasonWebSocket` (classe responsável pela comunicação com a simulação) e transmite uma representação em texto da ação e dos termos determinados pelo Jason.

No Quadro 10 é demonstrado um exemplo de arquivo com as definições feitas na linguagem `AgentSpeak(L)` que representam a mente de um agente.

Quadro 10 - Implementação de uma mente em `AgentSpeak(L)`

```
1  +onPercept(C)
2      <- changeMyFillStyle(C).
```

Na linha 1, pode-se observar que é adicionado o fato gerador `onPercept(C)` que quando alcançado através de percepções na simulação, dispara para o mundo externo a ação externa `changeMyFillStyle(C)` disposta na linha 2.

3.3.2.2 O motor de jogos

O viés de estender o motor de jogos desenvolvido por Harbs (2013), adicionando funcionalidades que proporcionem a geração de animação comportamental, visa possibilitar que personagens possam interpretar a simulação que estão presentes e agir com base no que é observado.

Com o intuito de possibilitar que personagens sejam dotados de animação comportamental, foi adicionado ao motor de jogos um *pipeline* de percepção, possibilitando que cada personagem (representado por um objeto do jogo) possa interagir com o objeto percebido. Conforme visto no Quadro 11 foi adicionado um novo evento a classe `Component`, distinguindo assim esse novo comportamento dos demais.

Quadro 11 - Novo eventos do objeto `Component`

```
1 Component.prototype.onPercept = function(gameObject){}
```

Fonte: estendido de Harbs (2013, p.45).

Na linha 1, nota-se a adição do evento `onPercept()` à classe `Component`, possibilitando que componentes que sobrescrevam tal método realizem suas tratativas quando interceptarem o disparo do *pipeline* de percepção.

O *pipeline* de percepção é formado pelo disparo dos eventos de percepção, fazendo com que os objetos detentores de um sensor de percepção realizem a transmissão das percepções obtidas para o raciocinador, aguardando a determinação da ação a ser executada.

No Quadro 12 é demonstrado a disposição do *loop* do jogo após a adição do *pipeline* de percepção.

Quadro 12 - Código fonte de classe `Game`

```
1 var Game = new function(){
2   ...
3   this.gameLoop = function(){
4     var deltaTime = (Date.now() - this.lastUpdateTime) / 1000;
5     if(!Game.paused){
6       this.updateGame(deltaTime);
7       this.stepGame();
8       this.perceptGame();
9       this.renderGame();
10    }
11    this.lastUpdateTime = Date.now();
12  }
13  ...
14  this.perceptGame = function() {
15    PerceptionSystem.firePerceptionListener();
16  }
17  ...
18 }
```

Fonte: estendido de Harbs (2013, p.42).

Na linha 8, pode ser observado que dentro do *loop* do jogo, foi adicionada uma chamada para o método `perceptGame()` da própria classe `Game`, realizando assim a execução do *pipeline* de percepção. No método `perceptGame()` é realizado o disparo dos eventos de percepção ao invocar o método `firePerceptionListener()` da classe `PerceptionSystem`.

No Quadro 13 é apresentado o disparo dos eventos de percepção nos componentes e a identificação das percepções no motor de jogos.

Quadro 13 - Implementação da classe PerceptionSystem

```

1  var PerceptionSystem = new function(){
2
3      this.listPerceived = new Array();
4
5      this.isSensorialCollision = function(gameObject1, gameObject2){
6          if ( gameObject1 instanceof GameObject &&
7              gameObject2 instanceof GameObject) {
8              if ( gameObject1.isSensor || gameObject2.isSensor ) {
9                  if ( gameObject1.isSensor && !gameObject2.isSensor ) {
10                     this.putPerceiveInfo(gameObject1, gameObject2);
11                 } else if ( !gameObject1.isSensor && gameObject2.isSensor ) {
12                     this.putPerceiveInfo(gameObject2, gameObject1);
13                 }
14                 return true;
15             }
16         }
17         return false;
18     }
19
20     this.firePerceptionListener = function(){
21         for( var i in this.listPerceived ){
22             var info = this.listPerceived[i];
23             if( info instanceof PerceptionInfo ){
24                 for( var j in info.gameObjectSensor.listComponents ){
25                     var component = info.gameObjectSensor.listComponents[j];
26                     if( component instanceof Component ){
27                         component.onPercept( info.gameObjectPerceived )
28                     }
29                 }
30             }
31         }
32         this.clearSensorInfo();
33     }
34     ...
35 }

```

Conforme pode ser visto na linha 20, o disparo dos eventos de percepção nos componentes, ocorre através da invocação do método `onPercept(GameObject)` (linha 27), durante a iteração das percepções registradas pelo motor de jogos. O registro das percepções (linhas 10 e 12) ocorre quando a Box2DJS identifica uma colisão entre dois objetos e o motor de jogos identifica que pelo menos dos objetos colididos possui o comportamento de sensor (linha 8). Quando o motor identifica a colisão de um sensor, é realizado o registro dos objetos de forma que seja possível identificar o objeto sensor como o realizador da percepção e o outro objeto como um objeto percebido (linhas 9 e 11).

Em toda colisão entre objetos, a Box2DJS possui o comportamento de realizar a repulsão entre os objetos colididos. Os objetos são projetados em direção contrária a que estavam realizando o deslocamento, de forma que a colisão pare de ocorrer, evitando que os mesmos ocupem o mesmo lugar no ambiente. Porém no caso da colisão identificada como uma percepção, o comportamento de repulsão entre os objetos não deve ocorrer. Para isso o

método `isSensorialCollision(GameObject, GameObject)` retorna o valor lógico verdadeiro, sinalizando para a Box2DJS que a repulsão dentre os objetos não deve ser resolvida. Também é assumido que a repulsão não deve ser resolvida quando ambos os objetos gráficos são identificados como sensores. Nessa situação o método `isSensorialCollision(GameObject, GameObject)` também retornará o valor lógico verdadeiro, porém sem efetuar o registro da colisão como uma percepção.

Para realizar a comunicação entre a simulação e o raciocinador, foi criado um componente denominado `PerceptionVisionComponent`, responsável por abstrair a criação de uma *WebSocket*, conforme pode ser visto no Quadro 14.

Quadro 14 - Implementação da sobrescrita do método `initialize` da classe `PerceptionVisionComponent`

```

1 JSUtils.addMethod(PerceptionVisionComponent.prototype, "initialize",
2   function(uri) {
3     this.initialize();
4     var pvc = this;
5     if ('WebSocket' in window || 'MozWebSocket' in window) {
6       this.webSocket = new WebSocket(uri);
7       this.timeOfInstantiation = Date.now();
8     } else {
9       alert("Browser não suporta WebSocket");
10      return this;
11    }
12    this.webSocket.onmessage = function(evt) {
13      pvc.onMessage(evt)
14    };
15    this.webSocket.onopen = function(evt) {
16      pvc.onOpen(evt)
17    };
18    this.webSocket.onclose = function(evt) {
19      pvc.onClose(evt)
20    };
21    this.webSocket.onerror = function(evt) {
22      pvc.onError(evt)
23    };
24    return this;
25  }
26 );

```

Nas linhas 1 e 2, pode-se observar que a através do método `addMethod(Object, String, Function)` da classe `JSUtils` é sobrescrito o método `initialize` na classe `PerceptionVisionComponent`. Ao sobrescrever tal método, é atribuído ao mesmo a competência de criar a *WebSocket* que será utilizada na comunicação com o raciocinador e retornar uma instância da classe criada, conforme pode ser visto na linha 6. Aos eventos `onmessage(MessageEvent)`, `onopen(Event)`, `onerror(Event)` e `onclose(CloseEvent)` da *WebSocket* são atribuídos, respectivamente, chamadas para os métodos `onMessage(MessageEvent)`, `onOpen(Event)`, `onError(Event)` e `onClose(CloseEvent)` da

classe `PerceptionVisionComponent` (linhas 12 a 23), possibilitando assim interceptar os eventos da *WebSocket* para criação de tratativas específicas caso sejam necessárias.

Quando um evento de percepção é disparado, os componentes responsáveis por tratar o mesmo realizam a comunicação com o raciocinador através do método `send(String)` da *WebSocket*, conforme pode ser visto no Quadro 15.

Quadro 15 - Implementação do método `onPercept(gameObjectPerceived)` na classe `PerceptionVisionComponent`

```

1 PerceptionVisionComponent.prototype.onPercept = function(
2   gameObjectPerceived ) {
3     if (this.websocket!=undefined && this.isOpen) {
4       var percepts = this.getPerceptions(gameObjectPerceived);
5
6       if ( percepts.length > 0 ) {
7         var perceptions = [];
8         for (var i = 0; i < percepts.length; i++) {
9           perceptions.push( percepts[i] );
10        }
11
12        var obj = new Object();
13        obj.origin = this.owner.id;
14        obj.target = gameObjectPerceived.id;
15        obj.perceptions = perceptions;
16        var message = JSON.stringify(obj);
17
18        var now = new Date();
19        this.queue.push(now);
20
21        this.websocket.send( message );
22      } else {
23        console.warn("Mensagem não enviada: Nenhuma percepção
24 identificada!");
25      }
26    } else {
27      console.warn("Mensagem não enviada: Socket não está
28 definida/aberta!");
29    }
30  }

```

O exemplo acima demonstra a implementação padrão do método `onPercept(gameObjectPerceived)` na classe `PerceptionVisionComponent`.

Na linha 4, pode-se observar que o componente recupera as percepções através do método `getPerceptions(gameObjectPerceived)`. Caso existam percepções, as mesmas são iteradas e em seguida é definido um novo objeto contendo o identificador do objeto sensor, o identificador do objeto percebido e as percepções iteradas (linhas 7 a 15). Desse novo objeto é criada uma representação em texto no formato JSON (linha 16), conforme exemplo demonstrado no quadro Quadro 16 e a representação é transmitida via *WebSocket* para o raciocinador (linha 21).

Quadro 16 - Exemplo de mensagem enviada pela simulação para o raciocinador

```

1 {
2   "origin": "1d252436-04b8-4756-b1fd-510d11ea15fe",
3   "target": "467de1a7-a811-4bb2-9ccd-c86003b9408a",
4   "perceptions": "[\"onPercept(\\\"#673dd7\\\")\"]"
5 }

```

Conforme visto nas linhas de 2 a 5, a mensagem é composta pelo identificador do objeto que realizou a percepção no ambiente (*tag origin*), o identificador do objeto identificado na percepção (*tag target*) e uma lista de mensagens com as percepções identificadas (*tag perceptions*).

Após realizar um ciclo de raciocínio, caso o raciocinador determine uma ação a ser executada pelo agente na simulação, é criada uma mensagem contendo a descrição da ação (formato JSON) e posteriormente a mensagem é enviada para simulação. A composição dessa mensagem pode ser vista no Quadro 17.

Quadro 17 - Exemplo de mensagem enviada pelo raciocinador para a simulação

```

1 {
2   "origin": "1d252436-04b8-4756-b1fd-510d11ea15fe",
3   "target": "467de1a7-a811-4bb2-9ccd-c86003b9408a",
4   "perceptions": "[\"onPercept(\\\"#673dd7\\\")\"]",
5   "action": "changeMyFillStyle(\\\"#673dd7\\\")"
6 }

```

Pode-se observar nas linhas 2 a 4, que a mensagem de retorno é parcialmente idêntica à mensagem enviada anteriormente pela simulação para o raciocinador. A diferença entre as mensagens é a presença da descrição da ação determinada pelo raciocinador, sob a *tag action* (linha 5).

Após ser realizada a transmissão da mensagem do raciocinador para a simulação, na *WebSocket* o é disparado o evento `onmessage(MessageEvent)`, que realiza a chamada para o método `onMessage(MessageEvent)` da classe `PerceptionVisionComponent` (Quadro 14, linhas 12 a 14), responsável por encaminhar a execução da ação, conforme visto no Quadro 18.

Quadro 18 - Implementação do método `onMessage (MessageEvent)` na classe `PerceptionVisionComponent`

```

1 PerceptionVisionComponent.prototype.onMessage = function (evt) {
2   if ( this.HAND_SHAKE == evt.data ) {
3     this.isOpen = true;
4   } else {
5     var now = new Date();
6     var sendDate = this.queue.shift();
7     var reasoningTime = Math.abs(now-sendDate)/1000;
8     if ( this.averageReasoningTime==0 ) {
9       this.averageReasoningTime = reasoningTime;
10    } else {
11      this.averageReasoningTime =
12        (this.averageReasoningTime+reasoningTime)/2;
13    }
14    var action = JSON.parse(evt.data).action;
15    this.executeAction( action );
16  }
17 }

```

Na linha 2, pode-se observar que a primeira ação a ser realizada no método `onMessage (MessageEvent)` é a comparação com o literal “HAND_SHAKE”. A comparação é realizada devido o comportamento do componente `PerceptionVisionComponent` de considerar a *WebSocket* aberta (estado) apenas após o retorno do envio da mensagem de HAND_SHAKE (envio ocorre quando é disparado o evento `onopen (Event)` na *WebSocket*). Quando o método identifica o retorno dessa mensagem, é atribuído o lógico verdadeiro a *flag* `isOpen` da classe `PerceptionVisionComponent`. Para possibilitar um gerenciamento de tempo de raciocínio do raciocinador (considerando transmissão, raciocínio e recepção) é realizada a diferença entre a hora de recepção da mensagem (linha 5 a 7) e a hora da transmissão da mensagem (ver Quadro 15, linhas 18 e 19) obtendo assim o tempo de raciocínio. Com o tempo calculado, é realizada uma nova operação envolvendo os tempos calculados nas mensagens anteriores, resultando assim em uma média geral de tempo de raciocínio (linhas 8 a 13).

Ao final, a mensagem no formato JSON é convertida para um objeto, do objeto é obtida a ação determinada pelo raciocinador (linha 14), em seguida é disparada a execução da ação invocando o método `executeAction (String)`.

No Quadro 19 é apresentado um exemplo de implementação utilizando o novo recurso de percepção adicionado ao motor de jogos, estendendo a classe `PerceptionVisionPerformanceComponent` o comportamento de comunicação com o raciocinador é obtido através da classe estendida. Sobrescrevendo os métodos `getPerceptions (gameObjectPerceived)` e `executeAction (action)` é possível determinar um comportamento específico ao objeto que se deseja adicionar o componente.

Quadro 19 - Implementação de exemplo da classe PerceptionVisionPerformanceComponent

```

1  function PerceptionVisionPerformanceComponent() {}
2
3  PerceptionVisionPerformanceComponent.prototype =
4      new PerceptionVisionComponent();
5
6  PerceptionVisionPerformanceComponent.prototype.getPerceptions =
7      function( gameObjectPerceived ) {
8      var render = null;
9      if ( gameObjectPerceived instanceof BoxObject ) {
10         render = ComponentUtils.getComponent(gameObjectPerceived,
11             "BOX_RENDER_COMPONENT");
12     } else if ( gameObjectPerceived instanceof CircleObject ) {
13         render = ComponentUtils.getComponent(gameObjectPerceived,
14             "CIRCLE_RENDER_COMPONENT");
15     } else if ( gameObjectPerceived instanceof PolygonObject ) {
16         render = ComponentUtils.getComponent(gameObjectPerceived,
17             "POLYGON_RENDER_COMPONENT");
18     }
19     var perceptions = [];
20     if (render) {
21         perceptions.push( "onPercept(\"" + render.fillStyle + "\")" );
22         return perceptions;
23     }
24     return perceptions;
25 }
26
27 PerceptionVisionPerformanceComponent.prototype.executeAction =
28 function( action ) {
29     if ( this.owner.parent ) {
30         var arrAction = action.split("(");
31         arrAction = arrAction[1].split(")")[0];
32         arrAction = StringUtils.replaceAll(arrAction, "\"", "");
33         var render = null;
34         if ( this.owner.parent instanceof BoxObject ) {
35             render = ComponentUtils.getComponent(this.owner.parent,
36                 "BOX_RENDER_COMPONENT");
37         } else if ( this.owner.parent instanceof CircleObject ) {
38             render = ComponentUtils.getComponent(this.owner.parent,
39                 "CIRCLE_RENDER_COMPONENT");
40         } else if ( this.owner.parent instanceof PolygonObject ) {
41             render = ComponentUtils.getComponent(this.owner.parent,
42                 "POLYGON_RENDER_COMPONENT");
43         }
44         if (render) {
45             render.fillStyle = arrAction;
46         }
47     }
48 }

```

Como pode ser visto nas linhas 9 a 18, o componente de percepção recupera o componente responsável pela renderização do objeto percebido. Quando se tratar de um objeto conhecido pelo componente, é criada uma representação em texto de uma percepção com a cor de preenchimento do objeto percebido (linha 21) e retornada em forma de lista para o raciocinador, conforme visto na linha 22. Entre as linhas 34 e 43 pode-se observar que para executar a ação, o componente de percepção recupera o componente responsável pela

renderização do objeto que representa o agente e em seguida define como cor de preenchimento a cor proveniente da ação determinada pelo raciocinador (linhas 30 e 32).

3.3.2.3 O editor de jogos

Esta seção descreve os ajustes realizados no editor de jogos durante a implementação do raciocinador. Também são descritos os passos para a criação de um exemplo de utilização de percepção que interaja com a simulação.

Além do motor de jogos descrito na seção 3.3.2.2, também houve a necessidades de realizar adaptações no editor de jogos desenvolvido por Harbs (2013), em virtude do novo recurso adicionado ao motor.

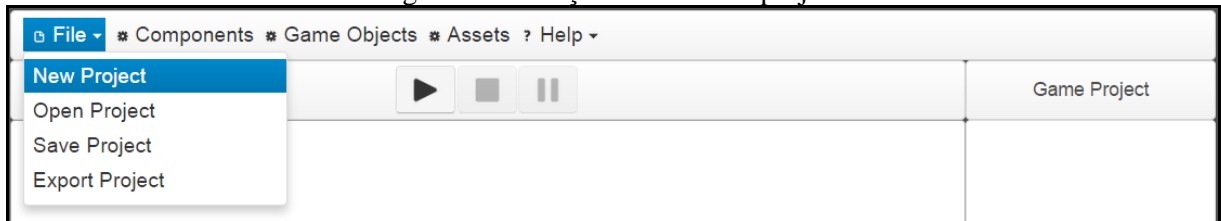
Conforme descrito na seção 3.2.4.1, para realização da exportar o jogo pelo editor de jogos, foram concatenados novos arquivos ao criar os *imports* necessários para execução do jogo. Foi adicionado o arquivo `PerceptionSystem.js` do pacote `js.engine.src.system`, o arquivo `PerceptionInfo.js` do novo pacote `js.engine.src.perception` e os arquivos `DirectoryUtils.js` e `TokenParentUtils.js` do pacote `js.engine.src.utils`.

Também foi criada a classe `UserUtils`, cuja função é através do método `getUserHome()` recuperar o diretório do usuário independente da plataforma utilizada. Possibilitando assim que o servidor de aplicação que disponibiliza o editor de jogos possa recuperar de uma única forma o caminho dos *assets* (`/editor_data/assets`), dos componentes (`/editor_data/components`) e dos objetos do jogo (`/editor_data/game_objects`) adicionados pelo usuário ao editor.

3.3.2.4 Operacionalidade da implementação

Como exemplo de percepção interagindo com o ambiente foi desenvolvido um protótipo de simulação de presa versus predador e para a criação da simulação, na tela inicial da ferramenta web para edição de jogos deve ser acessada a opção `File > New Project`, para adicionar a estrutura básica do jogo ao `Game Project`.

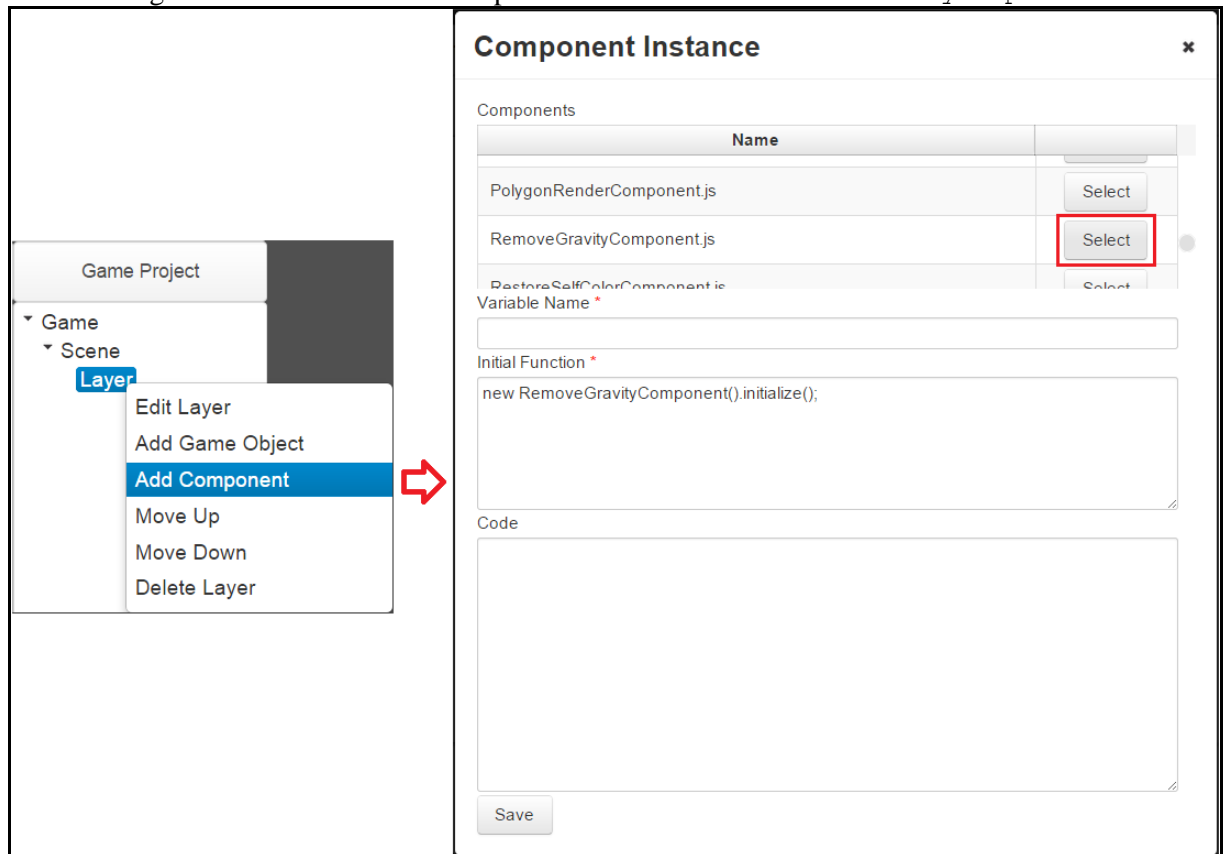
Figura 19 - Criação de um novo projeto



Fonte: estendido de Harbs (2013, p.50).

O protótipo de simulação de presa versus predador necessita que os personagens sofram ação da física ao colidirem entre si, porém não necessitam ser afetados pela força da gravidade enquanto estiverem presentes na simulação. Para tal, deve ser adicionada uma instância de um componente a camada do jogo (*layer*) que remova o comportamento padrão da gravidade, conforme pode ser visto na Figura 20.

Figura 20 - Adicionando o componente determinado RemoveGravityComponent

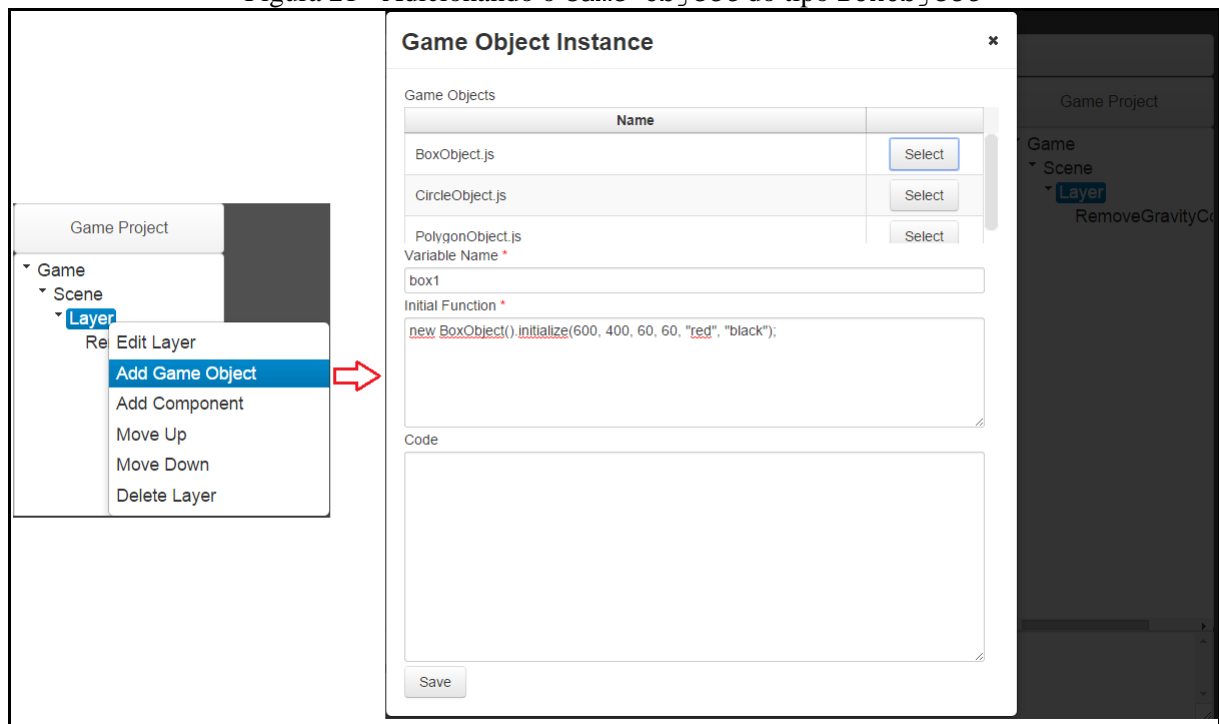


Para realizar a remoção da gravidade deve ser relacionada à camada (*layer*) do jogo, o componente `RemoveGravityComponent`, removendo assim a força de gravidade nessa camada da simulação. No editor, com o botão direito do *mouse*, sob o elemento *Layer* na representação em árvore do jogo (*Game Project*), deve ser selecionada a opção *Add Component*, em seguida selecionar o botão *Select* da instância de `RemoveGravityComponent`. Ainda na tela de seleção de instâncias de componentes, deve ser

atribuído um nome de variável a instância no campo `Variable Name` e em seguida a definição deve ser salva no botão `Save`.

Como prova de conceito, os personagens da simulação são representados por formas geométricas, o corpo do predador será representado por um quadrado (`BoxObject`), o campo de visão da presa será representado por um polígono em forma de triângulo (`PolygonObject`) e o corpo da presa será representado por um círculo (`CircleObject`). A adição da instância de `Game Object` que representará o predador pode ser vista na Figura 21.

Figura 21 - Adicionando o `Game Object` do tipo `BoxObject`



Para adicionar um `Game Object` à camada desejada, com o botão direito do *mouse* no editor, sob o elemento `Layer` na representação em árvore do jogo (`Game Project`), deve ser selecionada a opção `Add Game Object`, em seguida selecionar no botão `Select` da instância de `Game Object` desejada. No campo `Variable Name` deve ser atribuído um nome de variável a instância selecionada, no campo `Initial Function` devem ser definidas as propriedades da instância e ao final a definição do `Game Object` deve ser salva no botão `Save`.

Devido a necessidade das instâncias de `Game Object`, que representarão os personagens na cena, necessitam sofrer com a ação da física, uma instância do componente `RigidBodyComponent` deve ser atribuída a cada uma das representações, fazendo com que tal comportamento seja atribuído os corpos dos mesmo.

Em virtude da ausência do grafo de cena, foi necessária a criação de tratativas para identificação de relações entre objetos de uma cena. Foi definida uma estrutura de relação baseada na composição inicial dos *tokens*, ou seja, ambos os *tokens* devem possuir o mesmo literal inicial para serem considerados relacionados. Por exemplo, um objeto possui uma instância de componente do tipo `TokenComponent` que define um *token* “XPTO” e um segundo objeto também possui uma instância de componente do tipo `TokenComponent` definindo o *token* “XPTO_XPTO>”, esses dois objetos seriam considerados relacionados devido o literal inicial “XPTO” ser idêntico em ambos os *tokens*. Partindo dessa premissa, componentes podem definir comportamentos específicos ao identificarem o relacionamento entre determinados objetos. No protótipo da simulação, a relação entre o objeto que representa a presa e o objeto que representa o campo de visão da presa, foi criada através da adição de instâncias do componente `TokenComponent`, atribuindo respectivamente os *tokens* “PREY” e “PREY_VISION”.

Para proporcionar individualidade a um personagem, ou para um conjunto de personagens da cena, novas instâncias de componentes devem ser atribuídas aos mesmos. No caso do protótipo da simulação exemplificada, um comportamento comum aos personagens, é a livre movimentação pela cena. Para realizar a adição desse comportamento foi criado especificamente para o exemplo a classe `MoveGameObjectComponent`, responsável por definir um componente que proporcione o comportamento desejado aos objetos em estiver relacionado. Para possibilitar uma movimentação independente entre os personagens, conjuntos de propriedades distintos (grupo de identificadores de teclas) são atribuídos aos mesmos e devido a natureza da relação entre o objeto que representa a presa e o objeto que representa o campo de visão da presa, o mesmo conjunto de propriedades é atribuído para ambos os objetos.

A adição do comportamento de predador ao objeto que o representa, é realizada através da atribuição da instância do componente `DestroyGameObjectComponent`. Ao criar a instância desse componente são atribuídas duas propriedades, um *token* que permite realizar a identificação de uma presa e uma cor (nesse caso vermelha) que representará o estado ativo do predador. Ao objeto que representa o predador também é definido uma instância de `RestoreSelfColorComponent`, componente que através de execuções periódicas restaurar o estado ativo do predador, atribuindo a cor vermelha ao próprio corpo.

No Quadro 20 pode-se observar a codificação da mente que será atribuída a presa para interpretação no raciocinador.

Quadro 20 - Implementação da mente da presa em AgentSpeak

```

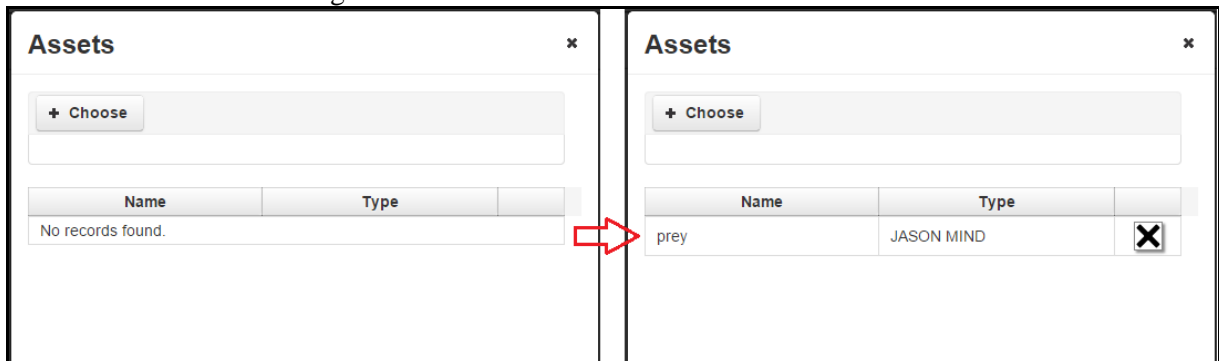
1 +onPercept(X, Y)
2   <- changeColor(X, Y, green, black).

```

Pode-se observar que quando o fato gerador `onPercept(X, Y)` (linha 1) for alcançado através das percepções identificadas no protótipo de simulação, a ação externa `changeColor(X, Y, green, black)` (linha 2) será disparada para a simulação contendo os termos `X` e `Y` obtidas do fato gerador da linha 1, somado aos termos fixos `green` e `black` da ação externa.

Para o editor de jogos, a mente da presa é tratada como um recurso da aplicação. Para realizar a disponibilização da mesma, no editor, deve ser acessado o menu `Assets`, conforme pode ser visto na Figura 22.

Figura 22 - Adicionando uma mente Jason ao editor



Com o arquivo da mente Jason criada e disponível sob a extensão `ASL`, na tela de adição de *asset*, basta pressionar o botão `+Choose`, localizar o arquivo, selecionar e confirmar a operação.

A adição da percepção ao protótipo de simulação exemplificada, ocorre através da atribuição de uma instancia do componente denominado `PerceptionVisionLotkaVolterraComponent` ao objeto que representa o campo de visão da presa. A comunicação entre a simulação gerada pelo motor de jogos e o raciocinador é realizada através de uma *WebSocket* gerenciada internamente pelo componente. O acesso ao raciocinador se dá através da URL de acesso definida na criação da instancia do componente. A implementação desse componente pode ser vista no Quadro 21.

Quadro 21 - Implementação da classe PerceptionVisionLotkaVolterraComponent

```

1  function PerceptionVisionLotkaVolterraComponent() {}
2
3  PerceptionVisionLotkaVolterraComponent.prototype = new
4      PerceptionVisionComponent();
5
6  PerceptionVisionLotkaVolterraComponent.prototype.getPerceptions =
7      function( gameObjectPerceived ) {
8      gameObjectPerceived.lastPerceived = Date.now();
9      var parent = null;
10     if (Game.scene) {
11         for (var i in Game.scene.listLayers) {
12             var layer = Game.scene.listLayers[i];
13             var foundInLayer = false;
14             for (var j in layer.listGameObjects) {
15                 var go = layer.listGameObjects[j];
16                 if (TokenParentUtils.isParent(this.owner, go)) {
17                     parent = go;
18                     foundInLayer = true;
19                     break;
20                 }
21             }
22             if (foundInLayer) { break; }
23         }
24     }
25     var perceptions = [];
26     if (parent!=null) {
27         perceptions.push( "onPercept(\"" + parent.id + "\",\"" +
28                             gameObjectPerceived.id + "\" )");
29     }
30     return perceptions;
31 }
32
33 PerceptionVisionLotkaVolterraComponent.prototype.executeAction =
34     function( action ) {
35     var arrAction = action.split("(");
36     arrAction = arrAction[1].split(")");
37     arrAction = arrAction[0].split(",");
38     var objId = StringUtils.replaceAll(arrAction[1], "\"", "");
39     if (Game.scene) {
40         for (var i in Game.scene.listLayers) {
41             var layer = Game.scene.listLayers[i];
42             var foundInLayer = false;
43             for (var j in layer.listGameObjects) {
44                 var go = layer.listGameObjects[j];
45                 if(go.id == objId && go instanceof BoxObject){
46                     var render = ComponentUtils.getComponent(go,
47                         "BOX_RENDER_COMPONENT");
48                     if (render) {
49                         render.fillStyle = arrAction[2];
50                         render.strokeStyle = arrAction[3];
51                     }
52                     foundInLayer = true;
53                     break;
54                 }
55             }
56             if (foundInLayer) { break; }
57         }
58     }
59 }

```

Conforme visto nas linhas 3 e 4, o componente criado especificamente para a simulação estende o componente `PerceptionVisionComponent`, obtendo o gerenciamento da *WebSocket* e dos demais métodos para intermédio da comunicação com o raciocinador. Nas linhas 6 e 7 ocorre a sobrescrita do método `getPerceptions(GameObject)`, possibilitando a buscar do objeto que representa o personagem (linhas 9 a 24) e a posterior criação de uma lista contendo uma mensagem (linhas 27 e 28) representando a percepção identificada, para envio ao raciocinador.

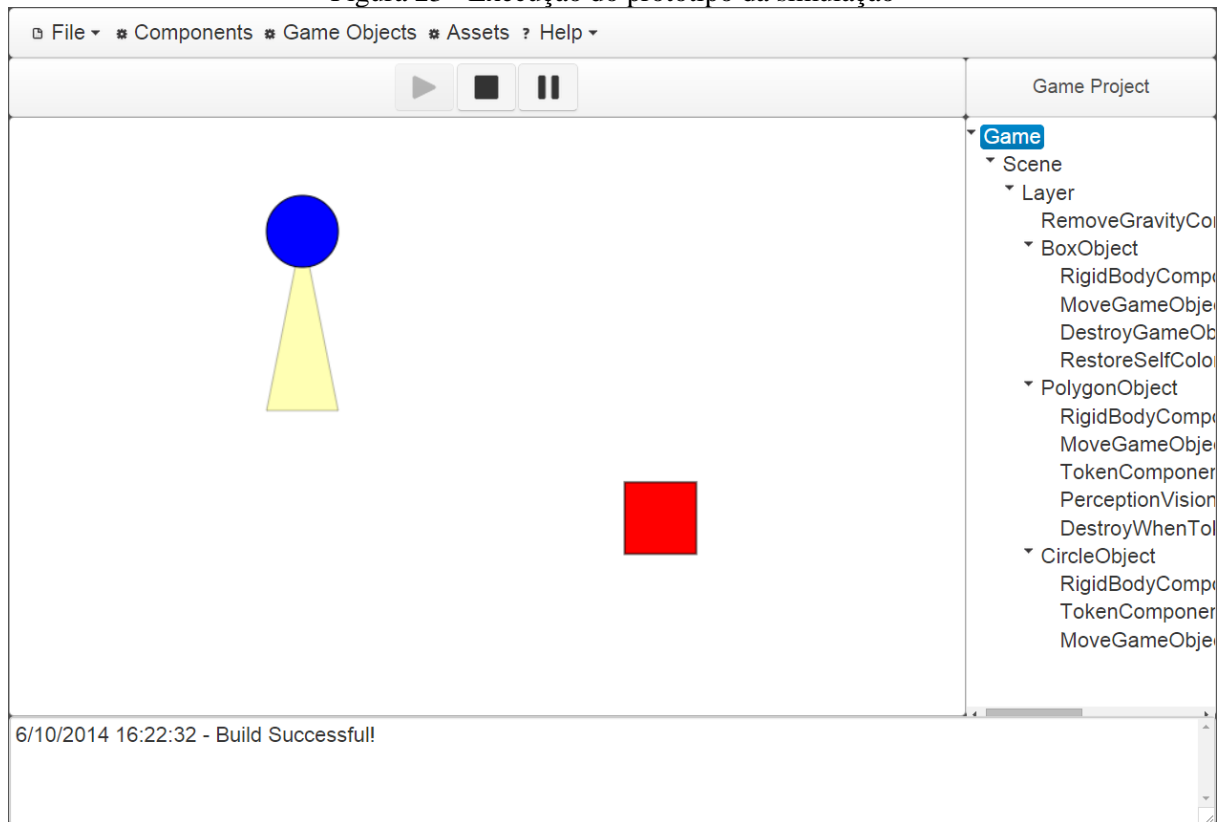
Na linha 8, o componente `PerceptionVisionLotkaVolterraComponent` define a propriedade `lastPerceived` no objeto percebido (predador), registrado no mesmo a última hora que houve uma percepção do mesmo. Essa atualização é realizada devido o comportamento do componente `RestoreSelfColorComponent`, pois o mesmo irá restaurar o estado ativo do predador (representado pela cor vermelha) apenas quando a diferença entre a hora da atualização e a hora da última percepção for maior que tempo mínimo sem percepção definido em sua instanciação.

Nas linhas 33 e 34 pode-se observar a sobrescrita do método `executeAction(String)`, viabilizando a customização da execução da ação determinada pelo raciocinador. A ação determinada está disposta no formato da ação externa da mente da presa (Quadro 20, linha 2). Pode-se observar nas linhas 35 a 37 que a ação é interpretada de forma que seja possível extrair os quatro termos presentes na composição da mesma. Através do segundo termo (identificador do predador reconhecido anteriormente na percepção) é identificado o objeto que sofrerá a ação determinada (linha 45), com o terceiro termo (`green`) é alterada a cor de preenchimento do objeto para cor verde e com o quarto termo (`black`) é alterada a cor da borda para preto.

Devido a ausência de uma relação direta entre o objetos que representam a presa e seu campo de visão, ao objeto que representa o campo de visão da presa também deve ser adicionada uma instância do componente `DestroyWhenTokenParentNotFoundComponent`, responsável por a cada atualização do jogo identificar se o objeto relacionado (presa) está presente na cena e ao perceber a ausência do mesmo destruir o objeto a qual está relacionado (campo de visão).

A representação em forma de árvore do protótipo de simulação criada, junto da execução do mesmo pode ser visto na Figura 23.

Figura 23 - Execução do protótipo da simulação



Para eventuais dúvida do fluxo de funcionamento do editor ver Operacionalidade da implementação disposto por Harbs (2013).

[@@ Dalton: Tela de documentação do motor ? carece?]

3.4 RESULTADOS E DISCUSSÃO

Este trabalho denominado de VISEDU-SIMULA, apresenta o desenvolvimento do módulo de Animação Comportamental adicionado ao visualizador de material educacional denominado VISEDU. Foram contemplados os objetivos propostos de adicionar funcionalidades que possibilitem a geração de Animações Comportamentais, proporcionando controle das percepções, raciocínio e atuação dos personagens. O raciocínio proporcionado ao criar o raciocinador, foi desenvolvido utilizando um dos modelos clássicos de IA, o modelo BDI, mostrando-se altamente escalável e possível de extensão. Vale salientar que o estrutura de disponibilização do raciocinador, denota o quão desacoplado é sua utilização, pois independente da técnica utilizada para a realização do raciocínio do lado do servidor, desde que a simulação receba as ações determinadas, a mesma é auto suficiente para executa-las.

Ao estender o motor de jogos, também se cumpriu o objetivo de manter a compatibilidade com os principais navegadores do mercado, junto da utilização das linguagens HTML5 e Javascript para seus devidos propósitos. A parte comportamental das

classes/componentes utilizando Javascript, o elemento canvas do HTML5 como um contêiner para renderização de formas em 2D e a *WebSocket*, outro elemento do HTML5, utilizado como meio de comunicação assíncrono de mão dupla entre simulação (cliente) e o raciocinador (servidor).

[@@ Dalton: usar Módulo de Animação Comportamental em iniciais maiúsculas ou apenas Animação Comportamental com iniciais maiúsculas?]

Inicialmente o desenvolvimento do módulo de Animação Comportamental demonstrou ser de um caráter exploratório, devido à indefinição do modelo de IA que seria utilizado para criação do raciocinador, o desconhecimento de como o mesmo seria utilizado e como se daria a troca de mensagens (percepções e ações) entre a simulação e o raciocinador. Após realizar a escolha pelo modelo BDI utilizando Jason, decidiu-se que o raciocinador seria disponibilizado em um servidor de aplicação (optou-se pelo Apache Tomcat) e a comunicação entre o mesmo e a simulação se daria através de *WebSocket*.

3.4.1 Arquitetura desenvolvida

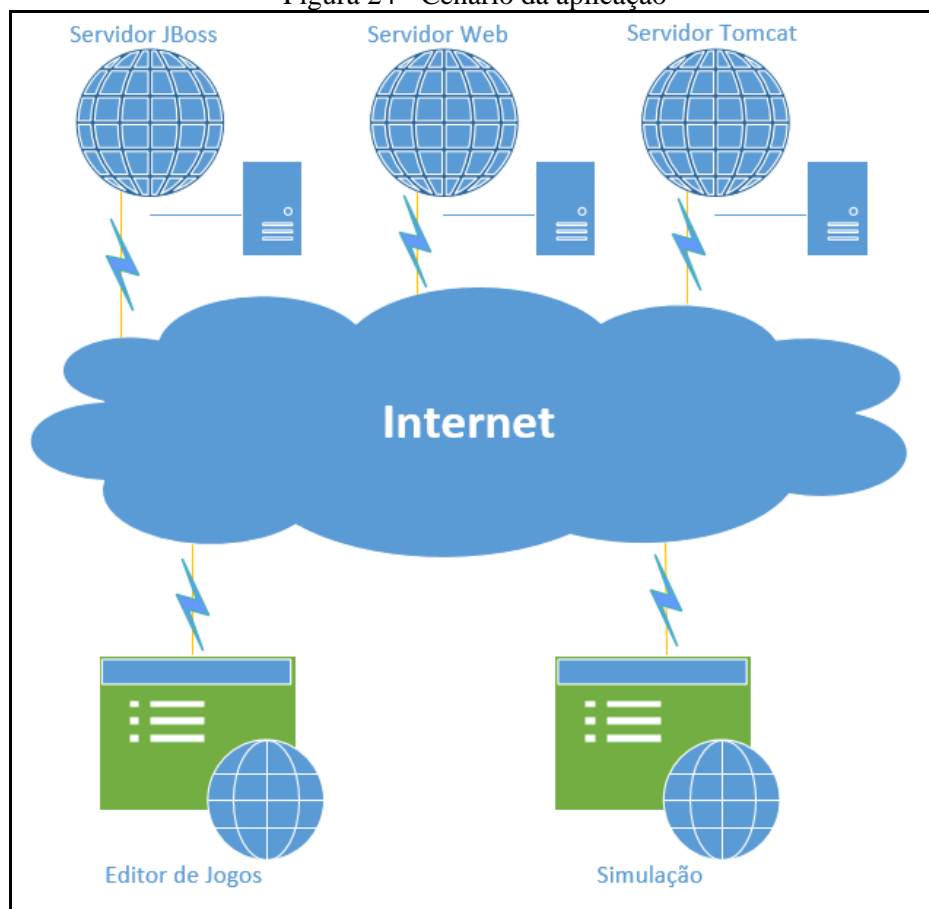
A módulo de Animação Comportamental foi concebido sob arquitetura cliente/servidor, conforme demonstrado na Figura 24.

O Servidor JBoss disponibiliza ambiente para criação e edição de jogos, possibilitando também salvar e exportar os jogos para posterior implantação manual no Servidor Web. O Servidor Web por sua vez disponibiliza acesso a execução da simulação. Ao ser acessado, os dados necessários para execução são carregados no navegador do usuário possibilitando que o mesmo tenha como interagir, ou apenas acompanhar a simulação. O Servidor Tomcat disponibiliza ambiente para realização do raciocínio dos personagens da simulação de forma transparente ao usuário.

[@@ Dalton: desmiuçar mais esse paragrafo?]

[@@ annot: como é cliente/server faz um texto grande, “a arquitetura desenvolvida/ a aplicação foi desenvolvida usando cliente/servidor, o cliente faz X e o servidor faz isso Y”. Demonstrar o menor primeiro (geralmente é o servidor).]

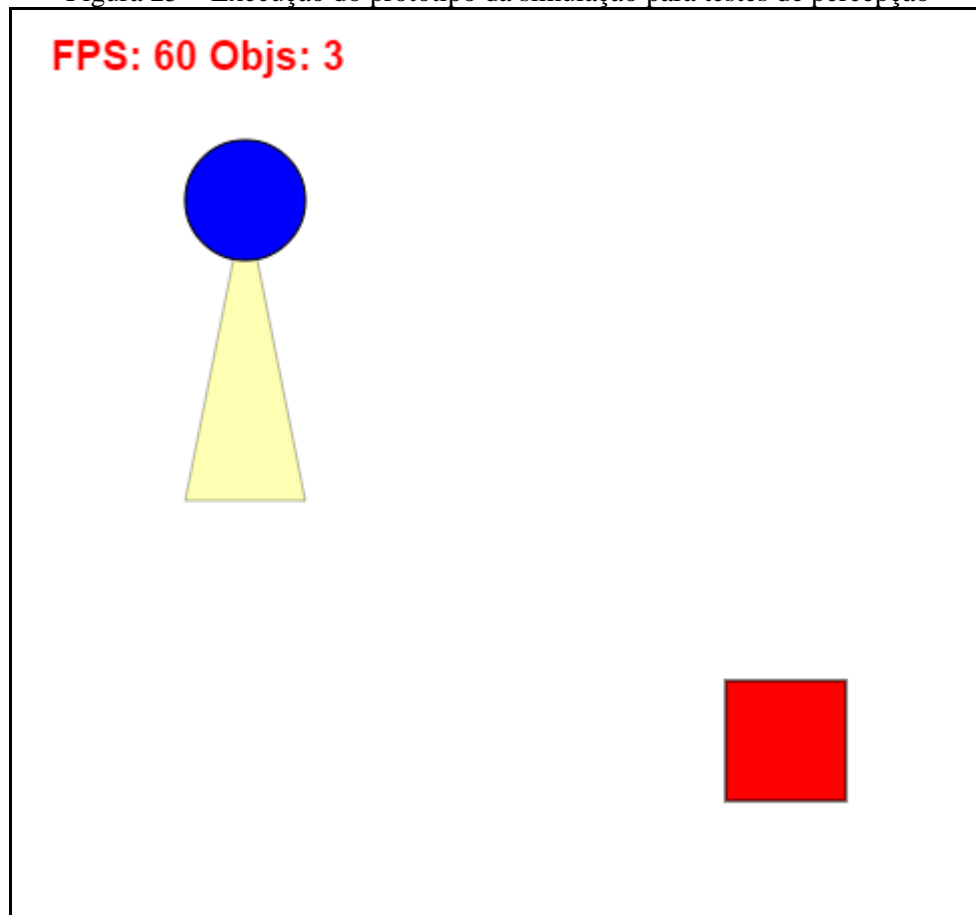
Figura 24 - Cenário da aplicação



3.4.2 Teste de funcionalidades

Para realizar verificações no módulo de Animação Comportamental, foi criado um experimento que proporcionasse a utilização dos novos recursos adicionados ao motor de jogos. O experimento é um protótipo que simula o ambiente de uma presa versus seu predador, conforme pode ser visto na Figura 25.

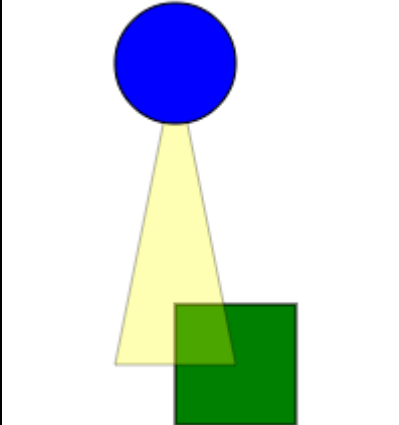
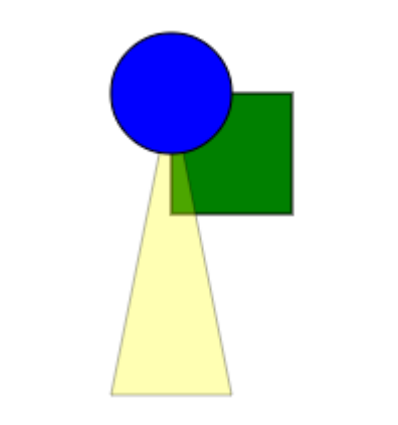
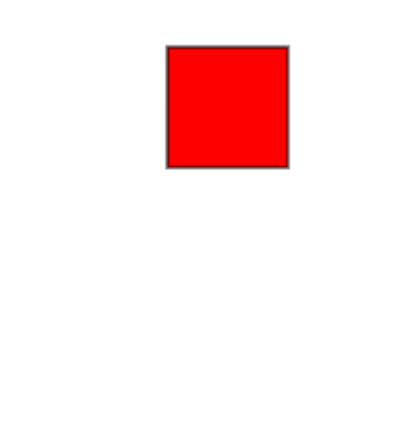
Figura 25 - Execução do protótipo da simulação para testes de percepção



No protótipo da simulação de teste, o quadrado vermelho representa o predador do ambiente, o círculo vermelho representa a presa e seu campo de visão é representado por um polígono triangular amarelo. Partindo o estado inicial, a simulação pode alcançar três estados possíveis, conforme visto no Quadro 22:

- a) a presa pode perceber o predador sem colidir com ele;
- b) a presa pode perceber o predador e estar colidindo com ele;
- c) o predador pode colidir com a presa sem ser percebido pelo campo de visão da mesma.

Quadro 22 - Estados do protótipo da simulação exemplificada

a) Percepção sem colisão	b) Percepção com colisão	c) Colisão sem Percepção
		

No primeiro estado, enquanto o predador estiver no campo de visão da presa, a cada ciclo de percepção, a percepção `onPercept(<identificador presa>, <identificador predador>)` é enviada para o raciocinador. Após o raciocinador interpretar a mente da presa, considerando a percepção recebida, é determinada a ação `changeColor(<identificador presa>, <identificador predador>, green, black)` e a cor do predador é alterada de vermelho para verde.

No segundo estado ocorrem as mesmas execuções descritas no primeiro passo, somada aos eventos propagados pela colisão entre a presa e o predador. Devido o predador estar com preenchimento da cor verde (estado desativado) o mesmo não consome a presa, estado que irá permanecer inalterado enquanto estiver no campo de visão da presa.

No terceiro estado, com o predador saindo do campo visual da presa, a instancia do componente `RestoreSelfColorComponent` ativa o estado do predador (representado pela cor vermelha) e ao colidir com a mesma a presa consumida pela predador.

3.4.3 Teste de desempenho

Para realização dos testes de performance da simulação, foram trabalhados dois possíveis cenários para coleta de dados, assumiu-se uma amostra de 5 execuções por navegador para calcular a média da performance.

Para realização da coleta das amostras foram utilizados os navegadores Google Chrome 38.0.2125.104m, Mozilla Firefox 33.0, Internet Explorer 11.0.9600.17351 e Opera 25.0.1614.50. Além dos navegadores citados, os testes também seriam realizados no navegador Safari 5.1.7 (versão atual do Safari para Windows), porém o navegador possui suporte completo a especificação da *WebSocket* a partir das versões 6.1 no iOS Safari e 7 no Safari (CANIUSE, 2014).

As figuras que demonstram as simulações nas seções 3.4.3.1 e 3.4.3.2, representam simulações com 50 agentes explorando o ambiente. Essas demonstrações diferem do situações em que os ambientes foram efetivamente testados, pois ao executar as simulações com mais de 100 agentes alinhados, explorando o ambiente com velocidade fixa, a compreensão das cores dos objetos não é clara. Essa problemática ocorre devido a necessidade de reajustar o tamanho dos corpos dos agentes e de seus obstáculos (no caso do cenário que possui), fazendo com que todos sejam apresentados na simulação, garantindo assim a renderização de todos os objetos.

No Cenário A foi definida uma simulação com a quantidade variável de agentes, que exploram horizontalmente o ambiente. Cada agente possui como representação do corpo em forma de quadrado e um polígono triangular representando seu campo de visão.

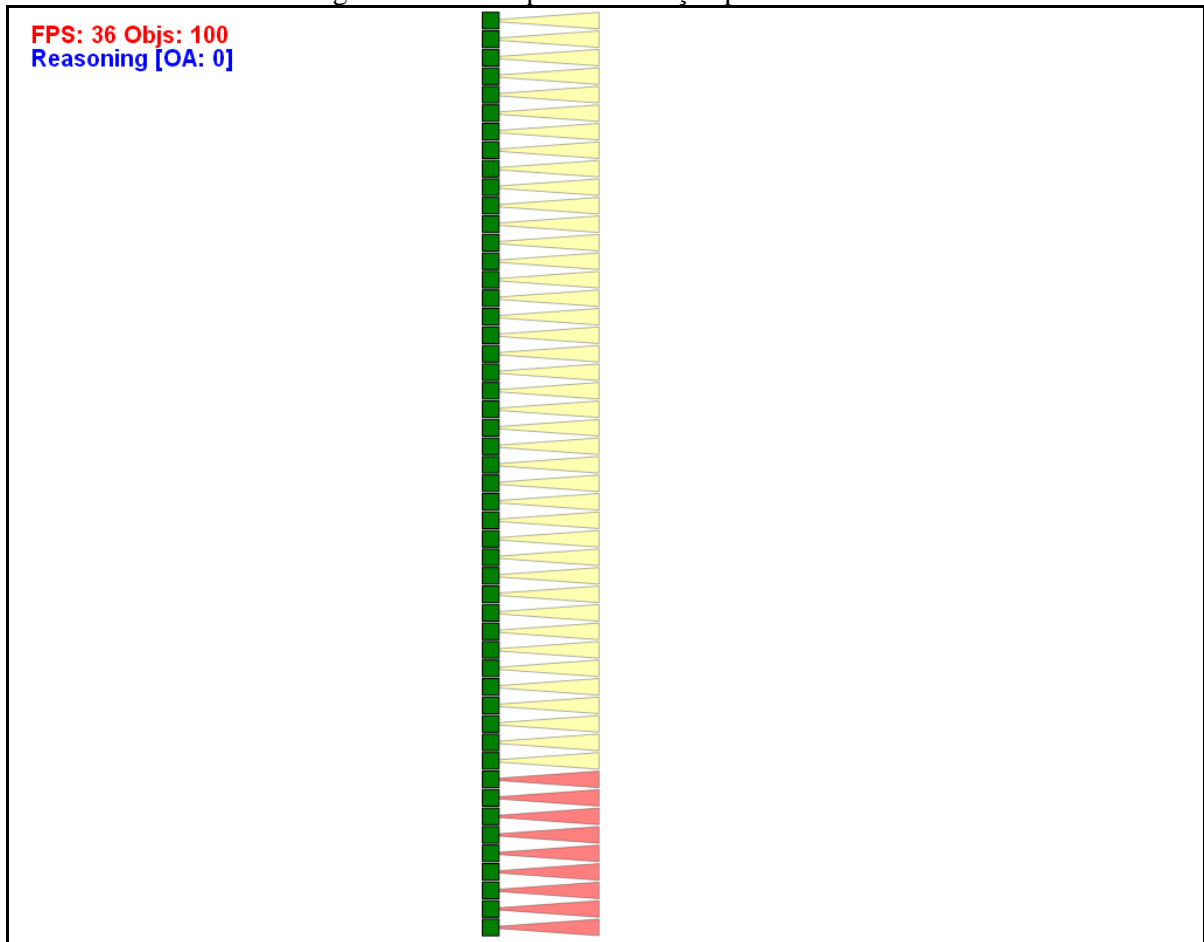
No Cenário B estende o Cenário A. Porém nesse cenário, toda vez que o agente inicia uma nova exploração, um obstáculo é criado no caminho de exploração, quando o agente identifica que o obstáculo entra em seu campo de visão, é enviada uma mensagem ao raciocinador com uma percepção contendo a cor de preenchimento reconhecida do objeto percebido. Quando o corpo do agente colide com o corpo do obstáculo, o último é destruído. Após a execução do ciclo de raciocínio, o raciocinador determina que o agente atribua para o próprio corpo a cor identificada do objeto percebido.

3.4.3.1 Cenário A

A Figura 26 apresenta uma execução com 50 agentes da simulação do Cenário A. Compete a este cenário realizar os teste desempenho do tempo de estabilização da conexão entre a simulação e o raciocinador. A simulação consiste na criação dos agentes no ambiente (variando de 1 a 200 conforme a amostragem) e após todos os agentes estabelecerem suas conexões é realizada a coleta dos dados.

Para representar visualmente o estabelecimento da comunicação com o raciocinador, ao criar o objeto que representa o campo de visão do agente (polígono triangular), é atribuída a cor vermelha para mesmo e ao identificar que a *WebSocket* está estabelecida, a cor amarela é atribuída ao corpo do campo de visão do agente.

Figura 26 - Protótipo de simulação para Cenário A



Devido a proposta de realizar o raciocínio de forma desacoplada a simulação exportada pelo editor, o raciocinador não possui nenhum vínculo direto com a mente do agente que terá suas percepções interpretadas. Para realização desse vínculo, é feito o *download* da mente no ato da conexão. Outra característica a ser levada em consideração é o comportamento da classe `PerceptionVisionComponent`. Conforme descrito no capítulo 3.3.2.2, compete a mesma realizar o gerenciamento da *WebSocket* utilizada na comunicação e devido seu comportamento de apenas considerar a comunicação estabilizada após o retorno do envio da mensagem de `HAND_SHAKE`, essa troca de mensagens também é considerada. Os tempos apresentados na Tabela 1, refletem o tempo de *download* do arquivo que representa a mente do agente, somado ao tempo da troca de mensagens de `HAND_SHAKE` e o tempo efetivo de estabelecimento da conexão da *WebSocket*. No cenário A foi utilizado como mente para os agentes, a definição disposta no Quadro 10, com tamanho de 41 *bytes*. As amostras que proporcionaram a criação da Tabela 1 estão disponíveis nos Apêndice A.

Tabela 1 – Média de tempo para estabelecer conexão com o raciocinador nos principais navegadores

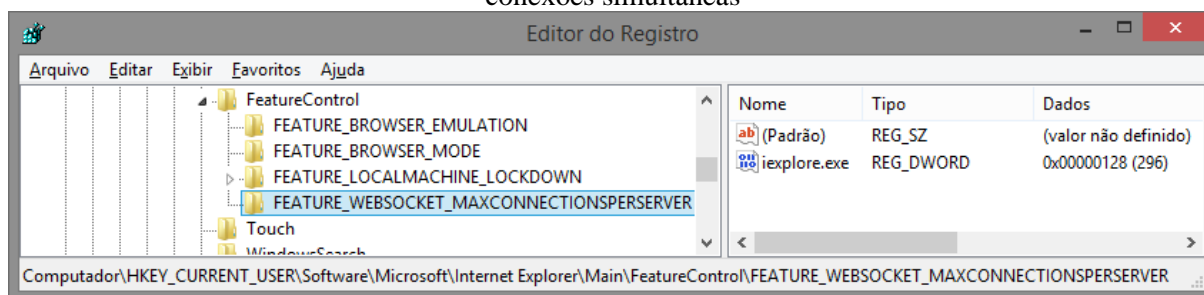
Quantidade de Agentes	Google Chrome	Mozilla Firefox	Internet Explorer	Opera
1	0,070	0,364	0,013	0,083
5	0,090	0,635	0,036	0,140
10	0,145	1,206	0,064	0,150
15	0,172	1,872	0,093	0,187
20	0,213	2,483	0,119	0,198
25	0,227	3,053	0,155	0,229
30	0,278	3,941	0,169	0,270
40	0,329	5,661	0,206	0,342
50	0,355	7,704	0,279	0,374
60	0,448	10,198	0,319	0,452
80	0,552	17,670	0,367	0,597
100	0,693	25,373	0,488	0,731
128	0,752	37,414	0,599	0,869
150	1,027	46,563		1,070
200	1,262	76,835		1,500

[@@ Dalton: para falar das limitações do IE eu adicionei a tabela aqui, pois ela de certa forma ilustra a limitação de 128 websockets. Mais a frente, depois de esclarecida a ideia eu apresento o diagrama. Mesmo nessa situação essa tabela deve virar um apêndice?]

Durante a execução do cenário A, limitações no navegador Internet Explorer foram identificadas. Ao captar dados para a amostra com 10 agentes identificou-se que, ao criar a sétima conexão a simulação lançava *SecurityError*. Posteriormente verificou-se que o número máximo permitido de conexões simultâneas de conexões (*WebSocket*) para um único *host* seria 6. Porém ativando o recurso `FEATURE_WEBSOCKET_MAXCONNECTIONSPERSERVER` esse número poderia ser modificado a um valor mínimo de 2 e máximo de 128. Por padrão esse recurso está desabilitado para o Internet Explorer, porém para habilitar o mesmo basta adicionar nome do executável ao recurso nos registro do sistema (*regedit*) (MICROSOFT, 2014).

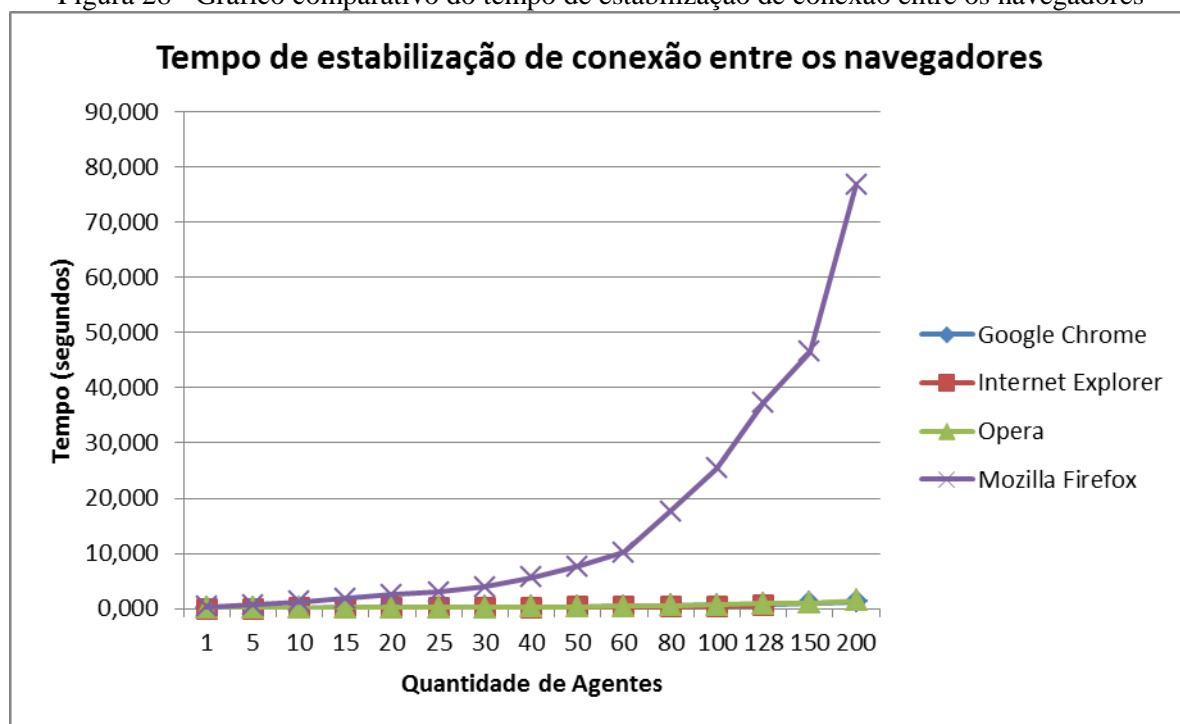
Conforme pode ser visto na Figura 27, foi realizada a ativação do recurso `FEATURE_WEBSOCKET_MAXCONNECTIONSPERSERVER`, permitindo a utilização do número máximo (128) possível de conexões simultâneas no Internet Explorer.

Figura 27 - Ativação do recurso `FEATURE_WEBSOCKET_MAXCONNECTIONSPERSERVER` definindo 128 conexões simultâneas



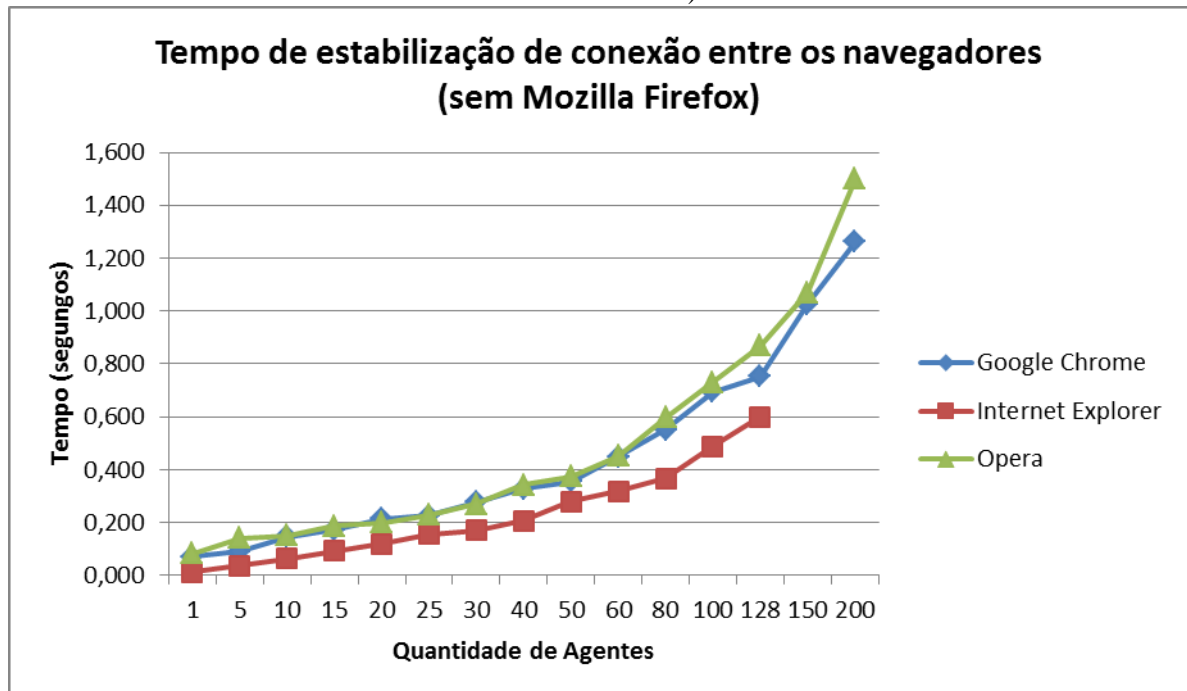
Definido o número máximo de conexões simultâneas para o Internet Explorer, foi adicionada uma nova amostra para todos os navegadores, no valor de 128 agentes, permitindo assim realizar uma comparação justa para Internet Explorer com os demais navegadores, a Figura 28 exibe um gráfico com os dados obtidos no Cenário A.

Figura 28 - Gráfico comparativo do tempo de estabilização de conexão entre os navegadores



A partir das médias proporcionadas pelas amostras do cenário A, com uma melhor disposição das informações no gráfico comparativo (Figura 28), percebe-se um baixo desempenho no navegador Mozilla Firefox em estabelecer a conexão das *WebSockets* com o raciocinador. Nota-se que o navegador foi o único a ultrapassar o tempo de 10 segundos para estabelecer as conexões nas amostras com mais de 50 agentes. Em virtude do baixo desempenho do Mozilla Firefox, foi disposto na Figura 29 um novo gráfico comparativo apenas com os demais navegadores, possibilitando realizar uma análise comparativa do desempenho dos mesmos.

Figura 29 - Gráfico comparativo do tempo de estabilização de conexão entre os navegadores (sem Mozilla Firefox)

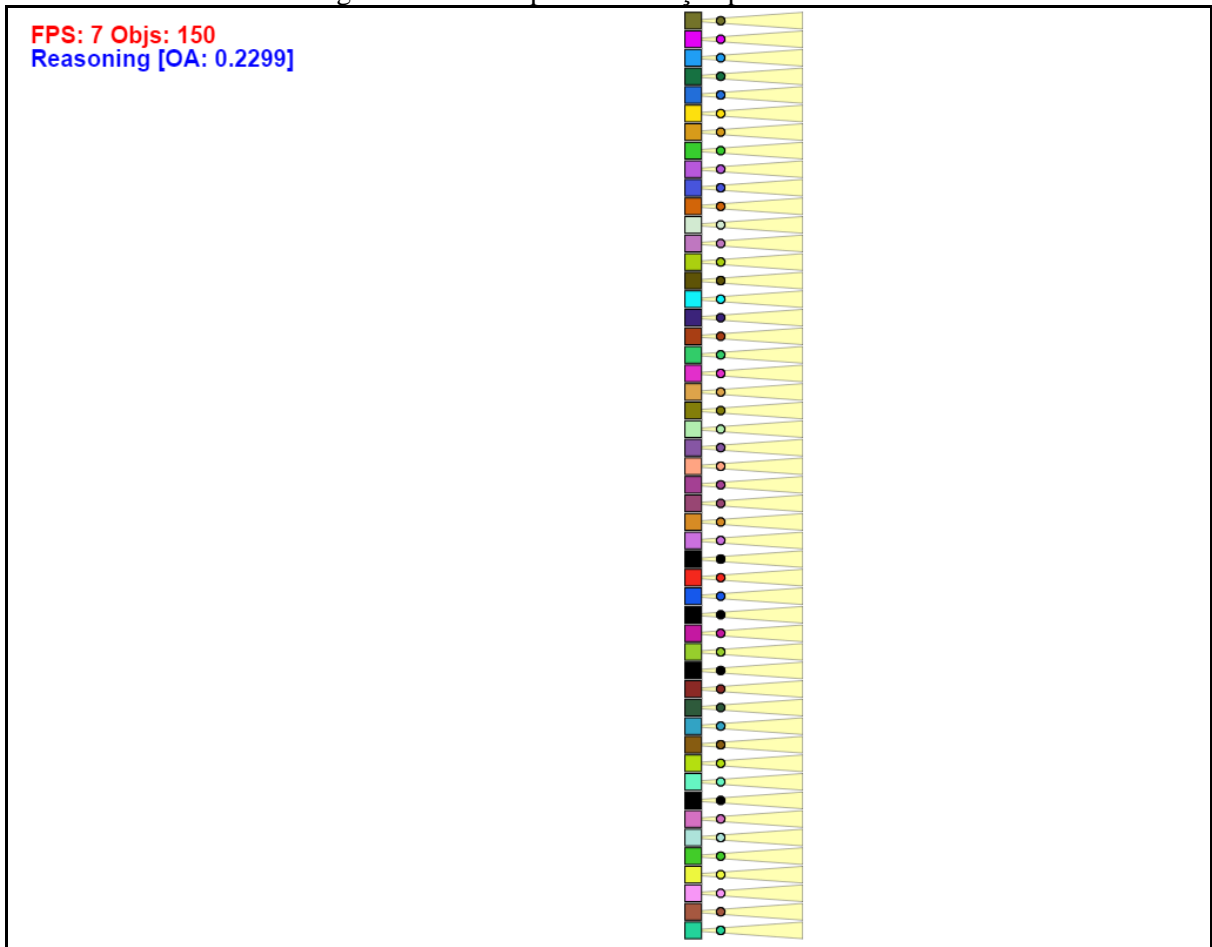


O navegador Internet Explorer, demonstrou ser o mais eficiente ao realizar o *download* da mente do agente, realizar a *HAND_SHAKE* e estabilizar a conexão com o raciocinador, dentro de suas limitações foi o navegador mais eficiente em todas as amostras que teve participação. Nas duas últimas amostras (sem a presença Internet Explorer) o Google Chrome demonstrou maior eficiência ao estabelecer a conexão. Em especial na última amostra, o navegador se demonstrou 16% mais eficiente que o Opera, indo contra a maioria das amostras anteriores, devido a ambos se intercalarem como o segundo navegador mais performático.

3.4.3.2 Cenário B

Compete ao Cenário B, demonstrar o capacidade de raciocínio do módulo desenvolvido e o impacto do mesmo quando a taxa de atualização de *frames per second* (FPS) da simulação. Assim como o Cenário A, a simulação cria um número variável de agentes (1 a 200, conforme a amostra) a um obstáculos para cada agente, que será percebido e destruído a cada exploração do ambiente. A Figura 30 demonstra a execução do ambiente.

Figura 30 - Protótipo de simulação para Cenário B



As amostras são recuperadas após a inicialização de todos os agentes, evitando que o Cenário A influencie no Cenário B. A cada ciclo de percepção durante a exploração que o agente realize uma percepção de um obstáculo, uma percepção é disparada para o raciocinador. Após interpretar essa percepção recebida com base na mente do agente (realização do raciocínio do agente), caso o raciocinador determine uma ação, a mesma é enviada para a simulação e executada pela mesma. Instantes antes do agente colidir com o objeto (destruindo-o) são registrados os valores de FPS e da média do raciocínio dos agentes da simulação. Foram criadas médias dos dados obtidos com o Cenário B e dispostos na Tabela 2. A amostra completa está disponível no Apêndice B.

Tabela 2 – FPS versus tempo médio de raciocínio

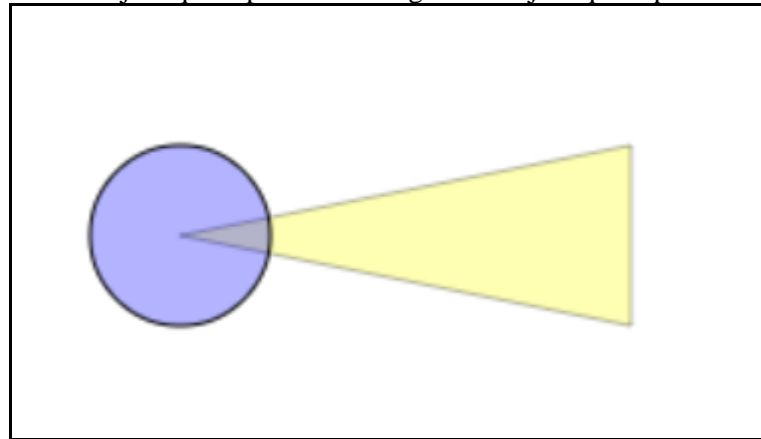
Agentes	Objetos na Cena	Google Chrome		Mozilla Firefox		Internet Explorer		Opera	
		FPS	RT	FPS	RT	FPS	RT	FPS	RT
1	3	61	0,0062	51	0,0230	60	0,0040	60	0,0080
5	15	58	0,0148	40	0,0370	60	0,0060	56	0,0150
10	30	48	0,0210	35	0,0610	59	0,0190	42	0,0210
15	45	30	0,0243	25	0,0830	49	0,0270	35	0,0270
20	60	19	0,0362	19	0,1010	41	0,0330	26	0,0350
25	75	15	0,0424	16	0,1160	29	0,0370	21	0,0390
30	90	13	0,0532	13	0,1330	24	0,0470	13	0,0490
40	120	10	0,0652	11	0,1810	17	0,0630	11	0,0620
50	150	8	0,0806	9	0,2160	13	0,0780	9	0,0720
60	180	8	0,0917	7	0,2770	11	0,0950	8	0,0830
80	240	5	0,1237	6	0,3280	9	0,1840	6	0,1260
100	300	4	0,1549	6	0,3740	8	0,2080	5	0,1950
128	384	5	0,1787	6	0,4640	7	0,4300	5	0,1730
150	450	4	0,2401	5	0,5380			5	0,2480
200	600	3	0,3255	4	0,6350			3	0,2990

A tabela apresenta a média de 5 amostras obtidas nos principais navegadores. Para cada um dos navegadores, obteve-se a média de atualização dos quadros por segundo (FPS) e a média de tempo de raciocínio de todos os agentes da simulação (RT).

Conforme destaque realizado na Tabela 2, observou-se que o módulo de Animação Comportamental desenvolvido, gerenciou até 15 agentes mantendo uma taxa acima ou próxima dos 30 FPS, com exceção do navegador Internet Explorer que destacou dos demais ao manter essa taxa de FPS enquanto o módulo desenvolvido trabalhava com até 25 agente.

Ao elevar o número de agentes na simulação, é perceptível o impacto no FPS a medida que é aumentado o número de agentes e de objetos na simulação. A adição do pipeline de percepção, contribui para impacto gerado, pois sua adição no *loop* do jogo faz com que o processo de disparo das percepções exerça concorrência com os demais pipelines da simulação. Outro fator que também influencia na queda da taxa de FPS, é a colisão permanente de cada objeto que representa um agente com seu respectivo objeto que representam o campo de visão. A Figura 31 demonstra a colisão entre o objeto que representa o agente e o objeto que representa o seu campo de visão.

Figura 31 - Colisão entre objeto que representa um agente e objeto que representa seu campo de visão



Por uma questão de relação com o mundo real, para aumentar o campo de visão inicial de um objeto, optou-se por definir a mesma origem para os objetos que representam o agente o campo de visão. Devido o objeto que representa o campo de visão se tratar de um sensor, para o motor de jogos o objeto que compõe o campo de visão do agente não propaga colisão. Porém para a motor de física utilizado pelo motor de jogos (Box2DJS), essa colisão continua ocorrendo constantemente, pois agente e campo de visão não são repulsados.

Outra consideração sobre os dados obtidos na amostra, é o tempo médio de raciocínio obtido. Com exceção as amostras de 150 e 200 agentes no navegador Mozilla Firefox, todas as demais tiveram uma média de raciocínio abaixo de 0,5 segundos. Conforme visto no Quadro 10 e no Quadro 20, foi implementado um raciocínio simples de percepção (reativo), pois até fugiria dos objetivos propostos a criação de uma mente mais elaborada. Porém pode-se observar na seção 2.1 que o modelo do ASL (mente do modelo mental do Jason) pode ser estendido para se ter maiores funcionalidades, possibilitando novos testes de performance do raciocinador.

Também foram realizados testes com a estrutura de identificação de percepção, porém isentando a simulação do estresse proporcionado pela criação da *WebSocket* e da comunicação da simulação com o raciocinador. A média dos dados da amostra estão dispostos na Tabela 3. A amostra completa está disponível no Apêndice C.

Tabela 3 - Quantidade de quadros por segundo por objetos na cena (com detecção da percepção)

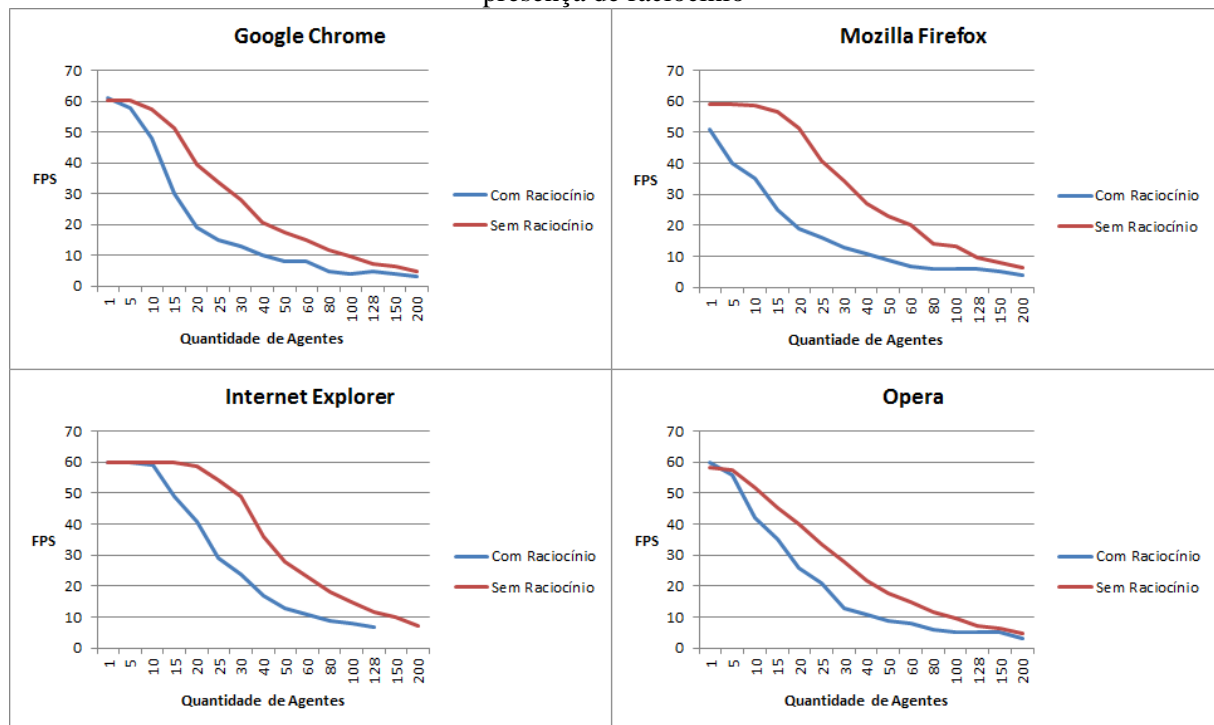
Quantidade de Agentes	Objetos na Cena	Google Chrome	Mozilla Firefox	Internet Explorer	Opera
1	3	60,4	59	60	58,4
5	15	60,2	59,2	60	57,6
10	30	57,4	58,8	60	51,6
15	45	51,2	56,6	60	45,4
20	60	39,4	51,4	58,8	40
25	75	33,8	41	54,4	33,4
30	90	28	34,4	49	28
40	120	20,8	27	36	21,8
50	150	17,4	23	28	17,6
60	180	15	20,2	23,2	15
80	240	11,6	14,2	18	11,6
100	300	9,6	13,2	14,8	9,6
128	384	7,2	9,8	11,6	7,4
150	450	6,6	8,2	10	6,4
200	600	4,8	6,6	7,2	4,8

Para cada agente adicionado a simulação, são adicionados outros dois objetos na cena, um representando o campo de visão do agente e outro como obstáculo a ser identificado durante a exploração do ambiente. Devido a natureza do teste de não utilizar a estrutura de comunicação com o raciocinador utilizando *WebSocket*, foi possível com o navegador Internet Explorer recolher amostras utilizando mais de 128 agentes na cena.

Conforme esclarecido nesse capítulo, devido a colisão constante do objeto que representam o agente com o objeto que representa o campo de visão, não é possível realizar uma comparação direta com o estado anterior a implementação do módulo de Animação Comportamental, porém uma comparação entre a simulação utilizando o raciocinador e a mesma simulação sem utilizar o raciocinador pode ser vista na Figura 32.

Pode-se identificar nos gráficos dispostos que a utilização de várias mentes exercem impacto direto na taxa de FPS da simulação. Fica evidente que o navegador que mais sofreu influencia com o módulo de raciocínio foi o Mozilla Firefox. Na amostra com um agente, o mesmo apresentou uma taxa média de 51 FPS, enquanto todos os demais estavam apresentaram uma média de 60 FPS. O navegador da Mozilla também apresentou uma taxa média abaixo de 30 FPS na simulação com 15 agentes, enquanto o demais apenas na amostra com 20 agentes. Conforme destacado por Harbs (2013), os navegadores Google Chrome e Opera, por compartilharem o mesmo motor de renderização e do motor de JavaScript apresentaram dados de performance parecido. O mesmo ocorreu com a implementação do módulo, pois foi bastante semelhante o impacto sobre ambos.

Figura 32 - Comparação da quantidade de quadros por segundo de uma mesma simulação com e sem a presença de raciocínio



3.4.4 Comparativo entre o módulo desenvolvido e seus correlatos

Nessa seção será apresentada uma comparação entre as principais características do módulo de Animação Comportamental desenvolvido e dos trabalhos definidos como correlatos do mesmo.

Característica	Unity 3D	Mattedi	Massive	Visedu-Simula
Proporciona animação comportamental	?	?	X	X
Desenvolvido Fortemente desacoplado	?	?	?	X
Controle mínimo da percepção, raciocínio e ação			X	X
Utiliza Modelo Clássico de IA	?	?	X	X
Desenvolvido com HTML5 e JavaScript	X			X

[@@ Dalton: precisa de mais características?]

[@@ annot: deve existir uma discussão...]

[@@ Dalton: o professor Capobianco recomendou comparar com os principais requisitos, mas isso não tende a destacar meu trabalho?]

4 CONCLUSÕES

[As conclusões devem refletir os principais resultados alcançados, realizando uma avaliação em relação aos objetivos previamente formulados. Deve-se deixar claro se os objetivos foram atendidos, se as ferramentas utilizadas foram adequadas e quais as principais contribuições do trabalho para o seu grupo de usuários ou para o desenvolvimento científico/tecnológico.]

[Deve-se também incluir aqui as principais vantagens do seu trabalho e limitações.]

Apesar da similaridade da linguagem JavaScript com a linguagem Java (linguagem que até então o autor detém maior conhecimento) e da alta curva de aprendizado proporcionada pela mesma junto da linguagem de marcação HTML (versão 5), a compreensão do motor de jogos e sua arquitetura orientada a componentes se demonstrou um fator dificultoso até a definição do modelo de solução que seria desenvolvido.

Outro fator problemático durante o desenvolvimento do raciocinador foi a realização do *download* da mente, quando executado pelo editor de jogos. Devido o mesmo executar o jogo em uma sessão, não foi possível do raciocinador realizar o download de uma sessão do navegador.

[@@ Dalton: como especificar que eu não consegui, o que não quer dizer que não seja possível fazer? Adicionando como extensão?]

Vale ressaltar que para a utilização de outro modelo mental, basta no raciocinador disponibilizar uma nova *WebSocket* que realiza o raciocínio para a simulação.

[@@ annot: o que mais pode ser mencionado?]

4.1 EXTENSÕES

[@@ suportar mais de 50 agentes em um mesmo ambiente, Dalton ficou de ajudar na explicação ... não me recordo do por que 50]

[Sugestões para trabalhos futuros.]

[propor criar modelos mais complexos para mente usando o conceito do ASL]

[extensão: estender o modelo do asl]]

[resolver problema no IE sem a necessidade de registro do windows]

Evitar que a *box2djs* calcule a colisão quando objeto dor um sensor, apenas realize o registro.

[usar *webworkers*] (*threads*) para realização do disparo das mensagens

REFERÊNCIAS

- APACHE TOMCAT. **Apache Tomcat**. [S.l.], 2014. Disponível em: <<http://tomcat.apache.org/index.html>>. Acesso em: 30 out. 2014.
- ARNAB, Sylvester et al. The development approach of a pedagogically-driven serious game to support Relationship and Sex Education (RSE) within a classroom setting. **Computers & Education**, [S.l.], v. 69, p. 15-30, nov. 2013.
- CANIUSE. **Web Sockets**. [S.l.], 2014. Disponível em: <<http://caniuse.com/#search=websocket/>>. Acesso em: 09 nov. 2014.
- HARBS, Marcos. **Motor para jogos 2D utilizando HTML5**. 2013. 77 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.
- HICKSON, Ian. **Re: [hybi] web socket protocol in "last call"?**. [S.l.], 2009. Disponível em: <<http://www.ietf.org/mail-archive/web/hybi/current/msg00784.html>>. Acesso em: 07 abr. 2014.
- LEVENT, Ibrahim. **Object-oriented JavaScript: advanced techniques for serious web applications**. [S.l.], 2013. Disponível em: <http://cdn.dzone.com/sites/all/files/refcardz/rc174-010d-Object-oriented-js_4.pdf>. Acesso em: 06 abr. 2014.
- LUBBERS, Peter. **HTML5 WebSocket: full-duplex, real-time web Communication**. [S.l.], 2011. Disponível em: <<http://refcardz.dzone.com/refcardz/html5-websocket>>. Acesso em: 07 nov. 2014.
- JASON. **About Jason**. [S.l.], 2014a. Disponível em: <<http://jason.sourceforge.net/wp/>>. Acesso em: 30 out. 2014.
- _____. **Description**. [S.l.], 2014b. Disponível em: <<http://jason.sourceforge.net/wp/description/>>. Acesso em: 07 nov. 2014.
- _____. **Examples**. [S.l.], 2014c. Disponível em: <<http://jason.sourceforge.net/wp/examples/>>. Acesso em: 07 nov. 2014.
- _____. **Documents**. [S.l.], 2014d. Disponível em: <<http://jason.sourceforge.net/wp/documents/>>. Acesso em: 11 nov. 2014.
- JBoss. **JBoss Tools**. [S.l.], 2014. Disponível em: <<http://tools.jboss.org/>>. Acesso em: 11 nov. 2014.
- _____. **All JBoss Application Server Downloads**. [S.l.], 2014b. Disponível em: <<http://jbossas.jboss.org/downloads.html>>. Acesso em: 11 nov. 2014.
- MAGNENAT-THALMANN, Nadia; THALMANN, Daniel. **Computer animation '90**. Tokyo: Springer, 1990.
- _____. **Handbook of virtual humans**. Chichester: John Wiley & Sons, 2004.
- MATTEDI, Filipe A. **Aperfeiçoamento de reações comportamentais de non-player character (NPC) no jogo Doom**. 2007. 99 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.
- MASSIVE. **About MASSIVE**. [S.l.], 2014a. Disponível em: <<http://www.massivesoftware.com/about.html>>. Acesso em: 12 abr. 2014.

_____. **Frequently asked questions**. [S.l.], 2014b. Disponível em: <<http://www.massivesoftware.com/faq.html>>. Acesso em: 12 abr. 2014.

_____. **Simulation life**. [S.l.], 2014c. Disponível em: <<http://www.massivesoftware.com>>. Acesso em: 12 abr. 2014.

MENDONÇA JR, Glaudiney M. **Animação comportamental**. [S.l.], [1999?]. Disponível em: <http://www.propgpq.uece.br/semana_universitaria/anais/anais1999/SemanaIV/VIII_IC/exatas/4iniexa11.htm>. Acesso em: 23 mar. 2014.

MICROSOFT. **Internet feature controls**. [S.l.], 2014. Disponível em: <http://msdn.microsoft.com/en-us/library/ee330736%28v=vs.85%29.aspx#websocket_maxconn>. Acesso em: 20 oct. 2014.

MULLER, Jorg P. et al. **Intelligent agents III**: proceedings. Berlin: Springer, 1997.

REYNOLDS, Craig. **Behavioral animation**. [S.l.], abr. 1997. Disponível em: <<http://www.red3d.com/cwr/behave.html>>. Acesso em: 23 mar. 2014.

RONCARELLI, Robi. **The computer animation dictionary**: including related terms used in computer graphics, film and video, production, and desktop publishing. New York: Springer, 1989.

SCHULTER, Fábio. **Simulador de uma partida de futebol com robôs virtuais**. 2007. 90 f. Trabalho de Conclusão de Curso - (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SILVA, Osmar J. **JavaScript avançado**: animação, interatividade e desenvolvimento de aplicativos. São Paulo: Érica, 2003.

SUBLIME. **Sublime Text**. [S.l.], 2014. Disponível em: <<http://www.sublimetext.com/>>. Acesso em: 30 out. 2014.

SUGRUE, James. **HTML5**: the evolution of web standards. [S.l.], 2010. Disponível em: <http://cdn.dzone.com/sites/all/files/refcardz/rc123-010d-html5_1.pdf>. Acesso em: 06 abr. 2014.

UNITY. **Crie jogos que você aprecia com o Unity**. [S.l.], 2014. Disponível em: <<http://unity3d.com/pt/unity>>. Acesso em: 05 abr. 2014a.

_____. **What is Unity?**. [S.l.], 2014. Disponível em: <<http://unity3d.com/pages/what-is-unity>>. Acesso em: 23 mar. 2014b.

_____. **Unity Pro**. [S.l.], 2014. Disponível em: <<https://store.unity3d.com/>>. Acesso em: 09 abr. 2014c.

W3C. **Application foundations for the open web platform**. [S.l.], 2014a. Disponível em: <<http://www.w3.org/blog/2014/10/application-foundations-for-the-open-web-platform/>>. Acesso em: 07 nov. 2014.

_____. **Open web platform milestone achieved with HTML5 recommendation**. [S.l.], 2014b. Disponível em: <<http://www.w3.org/blog/2014/10/application-foundations-for-the-open-web-platform/>>. Acesso em: 07 nov. 2014.

W3SCHOOLS. **HTML5 introduction**. [S.l.], 2014a. Disponível em: <http://www.w3schools.com/html/html5_intro.asp>. Acesso em: 08 abr. 2014.

_____. **JavaScript introduction**. [S.l.], 2014b. Disponível em: <http://www.w3schools.com/html/html5_intro.asp>. Acesso em: 08 abr. 2014.

WIKIPEDIA. **AgentSpeak**. [S.l.], 2014. Disponível em:
<<http://en.wikipedia.org/wiki/AgentSpeak>>. Acesso em: 07 nov. 2014.

WOOLDRIDGE, Michael J; JENNINGS, Nick. **Intelligent agents**: proceedings. New York: Springer, 1995.

Apêndice A – Tabelas com amostras do Cenário A.

Tabela 4 - Média de tempo para estabelecer conexão com o raciocinador - Google Chrome

Google Chrome								
Agentes	Objetos	Amostras de tempo (segundos)					Soma	Media
1	2	0,121	0,060	0,059	0,053	0,055	0,348	0,070
5	10	0,099	0,086	0,088	0,089	0,090	0,452	0,090
10	20	0,119	0,124	0,142	0,168	0,170	0,723	0,145
15	30	0,219	0,166	0,128	0,147	0,201	0,861	0,172
20	40	0,243	0,192	0,191	0,196	0,245	1,067	0,213
25	50	0,196	0,215	0,293	0,199	0,230	1,133	0,227
30	60	0,308	0,245	0,293	0,292	0,250	1,388	0,278
40	80	0,285	0,315	0,375	0,376	0,293	1,644	0,329
50	100	0,367	0,377	0,339	0,332	0,358	1,773	0,355
60	120	0,465	0,474	0,402	0,472	0,427	2,240	0,448
80	160	0,615	0,525	0,501	0,514	0,606	2,761	0,552
100	200	0,595	0,688	0,708	0,762	0,711	3,464	0,693
128	256	0,841	0,678	0,682	0,722	0,836	3,759	0,752
150	300	1,097	1,115	0,994	0,901	1,030	5,137	1,027
200	400	1,121	1,259	1,195	1,319	1,415	6,309	1,262

Tabela 5 - Média de tempo para estabelecer conexão com o raciocinador - Internet Explorer

Internet Explorer								
Agentes	Objetos	Amostras de tempo (segundos)					Soma	Media
1	2	0,013	0,013	0,013	0,013	0,014	0,066	0,013
5	10	0,036	0,035	0,039	0,034	0,034	0,178	0,036
10	20	0,067	0,055	0,067	0,074	0,055	0,318	0,064
15	30	0,088	0,085	0,085	0,091	0,114	0,463	0,093
20	40	0,140	0,124	0,112	0,110	0,108	0,594	0,119
25	50	0,171	0,124	0,168	0,140	0,170	0,773	0,155
30	60	0,174	0,159	0,169	0,175	0,170	0,847	0,169
40	80	0,205	0,205	0,207	0,205	0,206	1,028	0,206
50	100	0,344	0,258	0,273	0,270	0,250	1,395	0,279
60	120	0,292	0,398	0,319	0,311	0,277	1,597	0,319
80	160	0,378	0,359	0,370	0,381	0,347	1,835	0,367
100	200	0,415	0,461	0,538	0,472	0,552	2,438	0,488
128	256	0,533	0,568	0,660	0,606	0,630	2,997	0,599

Tabela 6 - Média de tempo para estabelecer conexão com o raciocinador - Opera

Opera								
Agentes	Objetos	Amostras de tempo (segundos)					Soma	Media
1	2	0,078	0,076	0,070	0,096	0,094	0,414	0,083
5	10	0,096	0,170	0,151	0,127	0,154	0,698	0,140
10	20	0,129	0,128	0,164	0,170	0,159	0,750	0,150
15	30	0,190	0,205	0,206	0,174	0,159	0,934	0,187
20	40	0,196	0,199	0,197	0,190	0,207	0,989	0,198
25	50	0,237	0,219	0,232	0,236	0,223	1,147	0,229
30	60	0,253	0,239	0,337	0,250	0,271	1,350	0,270
40	80	0,330	0,312	0,338	0,422	0,308	1,710	0,342
50	100	0,348	0,379	0,371	0,397	0,376	1,871	0,374
60	120	0,422	0,457	0,439	0,439	0,501	2,258	0,452
80	160	0,619	0,560	0,661	0,586	0,557	2,983	0,597
100	200	0,633	0,899	0,666	0,828	0,630	3,656	0,731
128	256	0,746	0,949	0,966	0,880	0,806	4,347	0,869
150	300	0,976	0,991	1,191	1,096	1,094	5,348	1,070
200	400	1,393	1,723	1,507	1,508	1,367	7,498	1,500

Tabela 7 - Média de tempo para estabelecer conexão com o raciocinador – Mozilla Firefox

Firefox								
Agentes	Objetos	Amostras de tempo (segundos)					Soma	Media
1	2	0,268	0,330	0,301	0,590	0,330	1,819	0,364
5	10	0,636	0,679	0,609	0,547	0,703	3,174	0,635
10	20	1,156	1,147	1,289	1,161	1,276	6,029	1,206
15	30	2,090	1,689	1,814	1,996	1,770	9,359	1,872
20	40	2,559	2,640	2,320	2,472	2,424	12,415	2,483
25	50	3,082	3,069	3,021	3,025	3,069	15,266	3,053
30	60	3,552	4,075	4,190	4,023	3,864	19,704	3,941
40	80	5,364	5,729	5,517	5,683	6,012	28,305	5,661
50	100	7,338	8,511	7,659	7,682	7,330	38,520	7,704
60	120	10,373	10,189	10,436	10,046	9,945	50,989	10,198
80	160	17,633	17,310	17,600	18,284	17,525	88,352	17,670
100	200	25,036	25,432	25,481	25,365	25,553	126,867	25,373
128	256	37,501	37,205	37,565	37,004	37,796	187,071	37,414
150	300	45,938	47,027	47,099	46,435	46,314	232,813	46,563
200	400	77,771	74,913	76,708	78,339	76,444	384,175	76,835

Apêndice B – Tabelas com amostras do Cenário B.

Tabela 8 - FPS versus tempo médio de raciocínio - Google Chrome

Google Chrome											
Agentes	Objetos	FPS	RT	FPS	RT	FPS	RT	FPS	RT	FPS	RT
1	3	60	0,0000	61	0,0090	61	0,0090	61	0,0068	60	0,0062
5	15	57	0,0156	57	0,0123	60	0,0148	61	0,0148	56	0,0164
10	30	33	0,0201	44	0,0243	54	0,0214	53	0,0197	57	0,0193
15	45	44	0,0000	23	0,0281	30	0,0308	25	0,0319	29	0,0306
20	60	17	0,0391	21	0,0360	19	0,0352	20	0,0355	18	0,0354
25	75	14	0,0419	15	0,0425	15	0,0428	15	0,0411	15	0,0435
30	90	12	0,0593	13	0,0544	13	0,0479	13	0,0494	13	0,0551
40	120	10	0,0585	10	0,0718	10	0,0679	10	0,0648	10	0,0630
50	150	8	0,0855	8	0,0782	8	0,0803	8	0,0796	8	0,0796
60	180	7	0,1009	7	0,1067	7	0,0909	10	0,0599	7	0,1001
80	240	5	0,1066	5	0,1257	5	0,1290	5	0,1391	7	0,1181
100	300	4	0,1739	5	0,1407	5	0,1516	4	0,1451	4	0,1633
128	384	5	0,1594	5	0,2469	6	0,1352	5	0,1526	4	0,1993
150	450	3	0,2880	5	0,2547	4	0,1919	5	0,2656	4	0,2002
200	600	3	0,3298	4	0,3842	3	0,4249	4	0,2393	3	0,2492

Tabela 9 - FPS versus tempo médio de raciocínio - Mozilla Firefox

Mozilla Firefox											
Agentes	Objetos	FPS	RT	FPS	RT	FPS	RT	FPS	RT	FPS	RT
1	3	51	0,0331	52	0,0169	52	0,0169	50	0,0209	52	0,0282
5	15	36	0,0300	32	0,0402	45	0,0483	39	0,0347	47	0,0297
10	30	34	0,0596	39	0,0534	29	0,0834	36	0,0616	37	0,0490
15	45	24	0,0766	20	0,1303	32	0,0682	23	0,0723	24	0,0661
20	60	19	0,0826	16	0,1043	22	0,0982	18	0,1231	20	0,0949
25	75	14	0,1065	15	0,1245	19	0,1260	16	0,1163	15	0,1072
30	90	13	0,1396	13	0,1369	13	0,1210	14	0,1393	13	0,1304
40	120	11	0,1802	11	0,1812	12	0,1689	11	0,1770	10	0,1975
50	150	9	0,2041	9	0,2143	9	0,2285	9	0,2105	9	0,2211
60	180	7	0,2572	7	0,2645	8	0,2439	8	0,2374	7	0,3813
80	240	6	0,3511	5	0,4032	6	0,3312	7	0,3060	8	0,2497
100	300	6	0,3574	7	0,3614	5	0,5596	6	0,2613	5	0,3290
128	384	6	0,5067	5	0,5034	6	0,4737	5	0,4532	6	0,3828
150	450	5	0,6224	5	0,4034	5	0,5918	6	0,5587	5	0,5119
200	600	4	0,6963	4	0,5709	4	0,6134	5	0,6002	4	0,6948

Tabela 10 - FPS versus tempo médio de raciocínio - Internet Explorer

Internet Explorer											
Agentes	Objetos	FPS	RT	FPS	RT	FPS	RT	FPS	RT	FPS	RT
1	3	60	0,0043	60	0,0039	60	0,0040	60	0,0040	60	0,0035
5	15	60	0,0064	60	0,0069	60	0,0060	60	0,0067	60	0,0060
10	30	57	0,0212	59	0,0187	59	0,0178	57	0,0182	61	0,0169
15	45	40	0,0303	55	0,0232	47	0,0262	57	0,0280	48	0,0255
20	60	29	0,0374	34	0,0359	56	0,0275	47	0,0287	38	0,0337
25	75	26	0,0366	29	0,0311	31	0,0403	31	0,0371	28	0,0390
30	90	22	0,0449	20	0,0445	21	0,0432	24	0,0573	33	0,0452
40	120	16	0,0539	15	0,0770	15	0,0681	18	0,0602	20	0,0542
50	150	16	0,0763	13	0,0746	12	0,0893	13	0,0707	13	0,0798
60	180	11	0,0865	11	0,0996	11	0,0856	11	0,0977	10	0,1054
80	240	10	0,1028	8	0,3860	8	0,1245	9	0,1892	8	0,1188
100	300	9	0,2725	9	0,1454	8	0,1243	8	0,2192	8	0,2779
128	384	7	0,4293	8	0,4603	7	0,4482	7	0,4385	8	0,3751

Tabela 11 - FPS versus tempo médio de raciocínio - Opera

Opera											
Agentes	Objetos	FPS	RT	FPS	RT	FPS	RT	FPS	RT	FPS	RT
1	3	60	0,0103	60	0,0076	60	0,0068	60	0,0078	60	0,0078
5	15	54	0,0145	59	0,0137	54	0,0156	59	0,0155	55	0,0158
10	30	33	0,0241	39	0,0200	53	0,0212	43	0,0215	40	0,0196
15	45	36	0,0282	29	0,0277	27	0,0269	52	0,0252	31	0,0270
20	60	21	0,0369	36	0,0370	22	0,0352	32	0,0336	19	0,0344
25	75	16	0,0461	16	0,0448	15	0,0436	26	0,0292	33	0,0323
30	90	13	0,0482	13	0,0476	13	0,0527	13	0,0501	13	0,0479
40	120	14	0,0601	10	0,0636	12	0,0556	10	0,0582	11	0,0700
50	150	9	0,0740	9	0,0742	9	0,0698	9	0,0694	9	0,0728
60	180	7	0,0894	8	0,0782	7	0,0860	7	0,0840	9	0,0776
80	240	6	0,1287	6	0,1301	6	0,1273	6	0,1335	6	0,1122
100	300	4	0,3354	4	0,1738	5	0,1503	6	0,1373	5	0,1797
128	384	5	0,1619	4	0,2042	5	0,1583	6	0,1214	6	0,2193
150	450	5	0,1708	4	0,1904	4	0,4461	5	0,1645	5	0,2677
200	600	3	0,2320	3	0,2190	3	0,2955	3	0,5506	4	0,1973

Apêndice C – Tabelas com amostras do Cenário B sem o processamento do raciocinador.

Tabela 12 - Quantidade de quadros por segundo por objetos na cena (com detecção da percepção) - Google Chrome

Agentes	Objetos	Google Chrome					FPS Médio
1	3	61	60	61	60	60	60,4
5	15	59	61	61	59	61	60,2
10	30	54	58	55	60	60	57,4
15	45	51	49	55	52	49	51,2
20	60	32	33	39	47	46	39,4
25	75	30	40	33	33	33	33,8
30	90	26	30	27	28	29	28
40	120	20	21	21	21	21	20,8
50	150	17	17	18	17	18	17,4
60	180	15	15	15	15	15	15
80	240	12	11	12	11	12	11,6
100	300	10	9	10	9	10	9,6
128	384	7	7	7	7	8	7,2
150	450	7	6	7	6	7	6,6
200	600	5	5	5	4	5	4,8

Tabela 13 - Quantidade de quadros por segundo por objetos na cena (com detecção da percepção) - Mozilla Firefox

Agentes	Objetos	Mozilla Firefox					FPS Médio
1	3	60	59	56	60	60	59
5	15	59	60	60	60	57	59,2
10	30	59	60	60	60	55	58,8
15	45	55	58	56	59	55	56,6
20	60	53	50	55	51	48	51,4
25	75	37	49	45	41	33	41
30	90	35	37	33	26	41	34,4
40	120	22	28	26	31	28	27
50	150	21	22	23	24	25	23
60	180	20	20	20	20	21	20,2
80	240	16	10	16	13	16	14,2
100	300	13	13	13	14	13	13,2
128	384	10	9	10	10	10	9,8
150	450	7	9	9	7	9	8,2
200	600	6	7	6	7	7	6,6

Tabela 14 - Quantidade de quadros por segundo por objetos na cena (com detecção da percepção) - Internet Explorer

Agentes	Objetos	Internet Explorer					FPS Médio
1	3	60	60	60	60	60	60
5	15	60	60	60	60	60	60
10	30	60	60	60	60	60	60
15	45	60	60	60	60	60	60
20	60	57	59	59	59	60	58,8
25	75	54	56	53	56	53	54,4
30	90	52	46	51	52	44	49
40	120	35	35	36	36	38	36
50	150	27	29	28	29	27	28
60	180	23	23	24	22	24	23,2
80	240	19	17	19	17	18	18
100	300	15	14	16	14	15	14,8
128	384	12	11	12	11	12	11,6
150	450	11	9	11	9	10	10
200	600	8	7	7	7	7	7,2

Tabela 15 - Quantidade de quadros por segundo por objetos na cena (com detecção da percepção) - Opera

Agentes	Objetos	Opera					FPS Médio
1	3	58	60	59	60	55	58,4
5	15	56	59	56	58	59	57,6
10	30	50	47	55	52	54	51,6
15	45	38	48	47	44	50	45,4
20	60	32	38	38	48	44	40
25	75	31	39	32	32	33	33,4
30	90	24	31	28	30	27	28
40	120	21	22	22	22	22	21,8
50	150	17	17	18	18	18	17,6
60	180	15	15	15	15	15	15
80	240	12	11	12	11	12	11,6
100	300	10	9	10	9	10	9,6
128	384	8	7	8	7	7	7,4
150	450	7	6	7	6	6	6,4
200	600	4	5	5	5	5	4,8

ANEXO A – Exemplo de uma mente criada em AgentSpeak interpretada pelo Jason.

Quadro 23- Implementação da mente do exemplo *Domestic Robot* (JASON, 2014c)

```

1  /* Initial beliefs and rules */
2  // initially, I believe that there is some beer in the fridge
3  available(beer,fridge).
4  // my owner should not consume more than 10 beers a day :-)
5  limit(beer,10).
6  too_much(B) :-
7      .date(YY,MM,DD) &
8      .count(consumed(YY,MM,DD,_,_,_,B),QtdB) &
9      limit(B,Limit) &
10     QtdB > Limit.
11 /* Plans */
12 +!has(owner,beer)
13   : available(beer,fridge) & not too_much(beer)
14   <- !at(robot,fridge);
15       open(fridge);
16       get(beer);
17       close(fridge);
18       !at(robot,owner);
19       hand_in(beer);
20       ?has(owner,beer);
21       // remember that another beer has been consumed
22       .date(YY,MM,DD); .time(HH,NN,SS);
23       +consumed(YY,MM,DD,HH,NN,SS,beer).
24 +!has(owner,beer)
25   : not available(beer,fridge)
26   <- .send(supermarket, achieve, order(beer,5));
27       !at(robot,fridge). // go to fridge and wait there.
28 +!has(owner,beer)
29   : too_much(beer) & limit(beer,L)
30   <- .concat("The Department of Health does not allow me to give you
31 more than ", L, " beers a day! I am very sorry about that!",M);
32       .send(owner,tell,msg(M)).
33 -!has(_,_)
34   : true
35   <- .current_intention(I);
36       .print("Failed to achieve goal '!has(_,_)'. Current intention is:
37 ",I).
38 +!at(robot,P) : at(robot,P) <- true.
39 +!at(robot,P) : not at(robot,P)
40   <- move_towards(P);
41       !at(robot,P).
42 // when the supermarket makes a delivery, try the 'has' goal again
43 +delivered(beer,_Qtd,_OrderId)[source(supermarket)]
44   : true
45   <- +available(beer,fridge);
46       !has(owner,beer).
47 // when the fridge is opened, the beer stock is perceived
48 // and thus the available belief is updated
49 +stock(beer,0)
50   : available(beer,fridge)
51   <- -available(beer,fridge).
52 +stock(beer,N)
53   : N > 0 & not available(beer,fridge)
54   <- -+available(beer,fridge).
55 +?time(T) : true
56   <- time.check(T).
57

```