**UNIVERSIDADE REGIONAL DE BLUMENAU**

**CENTRO DE CIÊNCIAS EXATAS E NATURAIS**

**CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO**

# VISEDU – AQUÁRIO VIRTUAL: SIMULADOR DE ECOSSISTEMA UTILIZANDO ANIMAÇÃO COMPORTAMENTAL

**KEVIN EDUARD PISKE**

**BLUMENAU**
**2015**

**2015/2-15**

**KEVIN EDUARD PISKE**

# VISEDU – AQUÁRIO VIRTUAL: SIMULADOR DE ECOSSISTEMA UTILIZANDO 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**
**2015**

**2015/2-15**

# VISEDU – AQUÁRIO VIRTUAL: SIMULADOR DE ECOSSISTEMA UTILIZANDO ANIMAÇÃO COMPORTAMENTAL

Por

**KEVIN EDUARD PISKE**

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. Alexander Roberto Valdameri, M.Sc. – FURB |
| Membro: | Prof. Roberto Heinzle, Dr. – FURB |

Blumenau, 04 de dezembro de 2015

Dedico este trabalho aos meus pais que sempre me incentivaram e nunca mediram esforços para me ajudar.

## AGRADECIMENTOS

Aos meus pais, Guido Piske e Denilse Matteussi Piske, pelo apoio, suporte, educação e conselhos a mim dados.

Ao meu orientador, Dalton Solano dos Reis, pelo apoio durante o desenvolvimento do trabalho.

A todos os meus professores, do ensino infantil ao superior, pois eles foram partes essenciais da minha educação e trajetória até aqui.

A ciência nunca resolve um problema sem criar outros dez.

George Bernard Shaw

# RESUMO

Este trabalho apresenta o desenvolvimento de um simulador de ecossistema de aquário marinho que visa auxiliar o estudo de conceitos biológicos. Seu desenvolvimento envolve a extensão de um motor de jogos e um módulo de raciocínio. O motor de jogos estendido foi implementado com a linguagem Javascript, com o elemento canvas do HTML5 e com a biblioteca gráfica ThreeJS; o módulo de raciocínio é uma aplicação que utiliza técnicas de inteligência artificial para realizar o raciocínio. Para a implementação do módulo de raciocínio, optou-se por utilizar o interpretador Jason para o desenvolvimento de agentes sob o modelo BDI, utilizando a linguagem AgentSpeak. No decorrer da implementação deste trabalho, o motor de jogos provou ser uma ferramenta capaz de proporcionar a base necessária para o desenvolvimento do ambiente de simulação. O interpretador Jason se mostra eficiente na criação de modelos mentais que proporcionam animações comportamentais aos seres virtuais da simulação. O ecossistema de um aquário marinho se mostrou um ótimo ambiente para a geração de animações comportamentais, permitindo uma variedade enorme de possibilidades. Implementou-se os comportamentos explorar, fugir, perseguir e comer, além de permitir a alteração de propriedades referentes ao mundo virtual, ao aquário e aos peixes. A aplicação também permite a visualização dos comportamentos atribuídos pelo módulo de raciocínio e exibe uma câmera secundária representando a visão do peixe. Na análise comparativa dos navegadores utilizados, o Chrome, o Firefox e o Opera demonstraram bons desempenhos e estabilidade na execução da simulação.

Palavras-chave: Animação comportamental. Ecossistema. Simulador.

# ABSTRACT

This paper presents the development of a marine aquarium ecosystem simulator that aims to assist the study of biological concepts. Its development involves the extension of a game engine and a reasoning module. The extended game engine has been implemented with the Javascript language, with the HTML5 canvas element and the graphics library ThreeJS; the reasoning module is an application that uses artificial intelligence techniques to perform the reasoning. To implement the reasoning module has been decided to use Jason interpreter for development of agents under the BDI model, using the AgentSpeak language. During the implementation of this work, the game engine proved to be a tool that can provide the necessary basis for the development of the simulation environment. Jason interpreter is efficient in creation of mental models that provide behavioral animations to virtual simulation beings. The ecosystem of a marine aquarium proved to be a great environment for the generation of behavioral animation, allowing a wide range of possibilities. Was implemented the behaviors explore, flee, pursue and eat, also allows the change of the properties for the virtual world, aquarium and fishes. The application also allows viewing of behaviors attributed by reasoning module and displays a secondary camera representing the fish's view. In the comparative analysis of used browsers, Chrome, Firefox and Opera showed good performance and stability in the simulation run.

Key-words: Behavioral animation. Ecosystem. Simulator.

# LISTA DE FIGURAS

# LISTA DE QUADROS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

2D – Duas Dimensões

ASL – *Agent Speak Language*

BDI – *Belief Desire Intention*

CSS – *Cascading Style Sheets*

FPS - *Frames Per Second*

FURB – Fundação Universidade Regional de Blumenau

GCG - Grupo de Pesquisa em Computação Gráfica, Processamento de Imagens e Entretenimento Digital

HTML - *HyperText Markup Language*

JSON - *JavaScript Object Notation*

MASSIVE - *Multiple Agent Simulation System In Virtual Environment*

MB – *MegaByte*

MTL – *Material Template Library*

OBJ – *Wavefront Object Files*

RF – Requisito Funcional

RNF – Requisito Não Funcional

STEVE - *Soar Training Expert for Virtual Environments*

UC – *Use Case*

VISEDU – Visualizador de Material Educacional

# SUMÁRIO

# 1 INTRODUÇÃO

Atualmente presencia-se a era da informação, da tecnologia e é evidente a insatisfação dos alunos em relação a aulas ditas "tradicionais", ou seja, aulas expositivas nas quais são utilizados apenas o quadro-negro e o giz. Uma vez que os alunos gostam tanto de aulas que utilizam a tecnologia, por que não aproveitar essa oportunidade e usá-la a seu favor? A aula pode entusiasmar os alunos de maneira parecida com que os jogos e filmes os entusiasmam (SOUZA, [200-?]). Portanto, a tendência é que softwares educacionais estejam cada vez mais presentes em centros de educação infantil, escolas e universidades, fazendo com que a aprendizagem se dê de uma forma mais divertida, interessante e cativante.

Dentre os softwares educacionais, destacam-se os simuladores, que trabalham com a noção de modelo e de processo. Neles os conteúdos são mais divertidos, seguros e criam situações de aprendizagem que permitem aos alunos experimentarem várias possibilidades (PEREIRA, 2011). Basicamente um simulador é um conjunto de hardware e software que reproduz um determinado ambiente real no meio computacional, podendo reproduzir por exemplo, um ecossistema inteiro, auxiliando no estudo e aprendizagem do mesmo. Os simuladores podem ser usados nas mais diversas áreas da educação, desde simulações de propriedades físicas até na representação de ecossistemas biológicos.

Desta forma, pode-se definir um ecossistema como um sistema biológico formado por dois elementos indissociáveis, a biocenose e o biótopo. A biocenose é o conjunto de seres vivos que convivem entre si e o biótopo é o meio que gera recursos para prover e dar manutenção à vida desses seres (DAJOZ, 2005). Logo os elementos da biocenose e do biótopo interagem uns com os outros, formando um todo coerente e ordenado (TAVARES, 2013).

Para simular um ecossistema, pode-se utilizar a Animação Comportamental em conjunto com Agentes Inteligentes e outras técnicas de Inteligência Artificial, fazendo com que a biocenose e o biótopo interajam entre si de forma autônoma. Cada ser vivo é um agente inteligente com capacidade de ter crenças, desejos e intenções, gerando animações sem precisar de um ser humano os auxiliando (REYNOLDS, 1997).

Segundo Feltrin (2014, p.15) “Para o desenvolvimento de Animação Comportamental, necessariamente a mesma precisa ocorrer em algum meio, que é um simulador.” Portanto, esse trabalho se destina ao desenvolvimento de um simulador de ecossistema de aquário marinho, utilizando como base para a implementação, o módulo de Animação

Comportamental do Visualizador de Material Educacional (VISEDU-SIMULA) e o módulo de Computação Gráfica do Visualizador de Material Educacional (VISEDU-CG).

## 1.1 OBJETIVOS

O objetivo deste trabalho é desenvolver um simulador de ecossistema de aquário marinho.

Os objetivos específicos do trabalho são:

a) estender o módulo desenvolvido por Feltrin (2014), desenvolvendo um modelo mental mais complexo;
b) ter um ambiente que permita a inserção de agentes dotados de representações gráficas;
c) adicionar funcionalidades que permitam gerar animações comportamentais para os personagens.

## 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 trabalho, descrevendo a implementação do aquário virtual, de um novo módulo de raciocínio estendido de Feltrin (2014) e de ajustes feitos no Motor de Jogos desenvolvido por Harbs (2013) e estendido por Koehler (2015). Demonstra também os diagramas de pacotes, de classes e de arquitetura utilizados, além de apresentar os resultados e discussões. No capítulo 4 são apresentadas as conclusões e as extensões sugeridas.

# 2 FUNDAMENTAÇÃO TEÓRICA

A seção 2.1 trata do uso de tecnologias na educação; a seção 2.2 define simulador e simulação; a seção 2.3 apresenta o projeto VISEDU-SIMULA 1.0; a seção 2.4 aborda animação comportamental e por fim a seção 2.5 apresenta os trabalhos correlatos.

## 2.1 USO DE TECNOLOGIAS NA EDUCAÇÃO

Este cenário tecnológico e informacional atual requer novos hábitos, uma nova forma de conceber, armazenar e transmitir o conhecimento, dando origem a novas formas de representação (PEREIRA, 2011). Segundo Pereira (2011, p. 20), "A aplicação do computador e das novas Tecnologias na Educação, requer novas formas de aprender e de ensinar." Porém, diante da quantidade de possibilidades para a construção de conhecimento, o aprendizado requer dos profissionais novas atitudes para desenvolver uma educação direcionada para nosso tempo, criando estratégias e situações de aprendizagem que possam tornar-se significativas para o aluno (PEREIRA, 2011). Apesar disso, não se trata somente dos profissionais da área da educação, mas também dos profissionais da área da computação, já que a união do conhecimento de ambos pode resultar em métodos de ensino e aprendizagem mais interessantes e divertidos para os alunos.

Souza ([200-?]) defende que "Aulas modernizadas pelo uso de recursos tecnológicos têm vida longa e podem ser adaptadas para vários tipos de alunos, para diferentes faixas etárias e diversos níveis de aprendizado. O trabalho acaba tendo um retorno muito mais eficaz." Leva-se em consideração que

> A internet invade nossos lares com todas as suas cores, seus movimentos e sua velocidade, fazendo o impossível tornar-se palpável, como navegar pelo corpo humano e visualizar a Terra do espaço sem sair do lugar. É difícil, portanto, prender a atenção do aluno em aulas feitas do conjunto lousa + professor (SOUZA, [200-?]).

## 2.2 SIMULADOR

Conforme Rosa (2008, p. 16), "Na área da computação um simulador consiste basicamente em criar um cenário virtual que propicie sua execução o mais próximo possível do mundo real e que aborde o maior número de variáveis reais possíveis." Em outras palavras, um simulador é um conjunto de hardware e software que reproduz um determinado ambiente real no meio computacional (virtual).

Já a simulação, segundo Shimizu (1975, p. 2), "É uma modalidade experimental de pesquisa que procura tirar conclusões através de exercícios com modelos que representam a realidade. Simulação é portanto um processo de imitar uma realidade através de modelos."

Historicamente a simulação, como técnica, originou-se dos estudos de Von Neumann e Ulan, onde estes estudos ficaram conhecidos como análise ou técnica de Monte Carlo. Após isso, a simulação começou a ser mais utilizada como técnica para solução de problemas, principalmente para o tratamento de problemas probabilísticos, cuja solução analítica é, geralmente, muito mais difícil, senão impossível (SCHULTER, 2007).

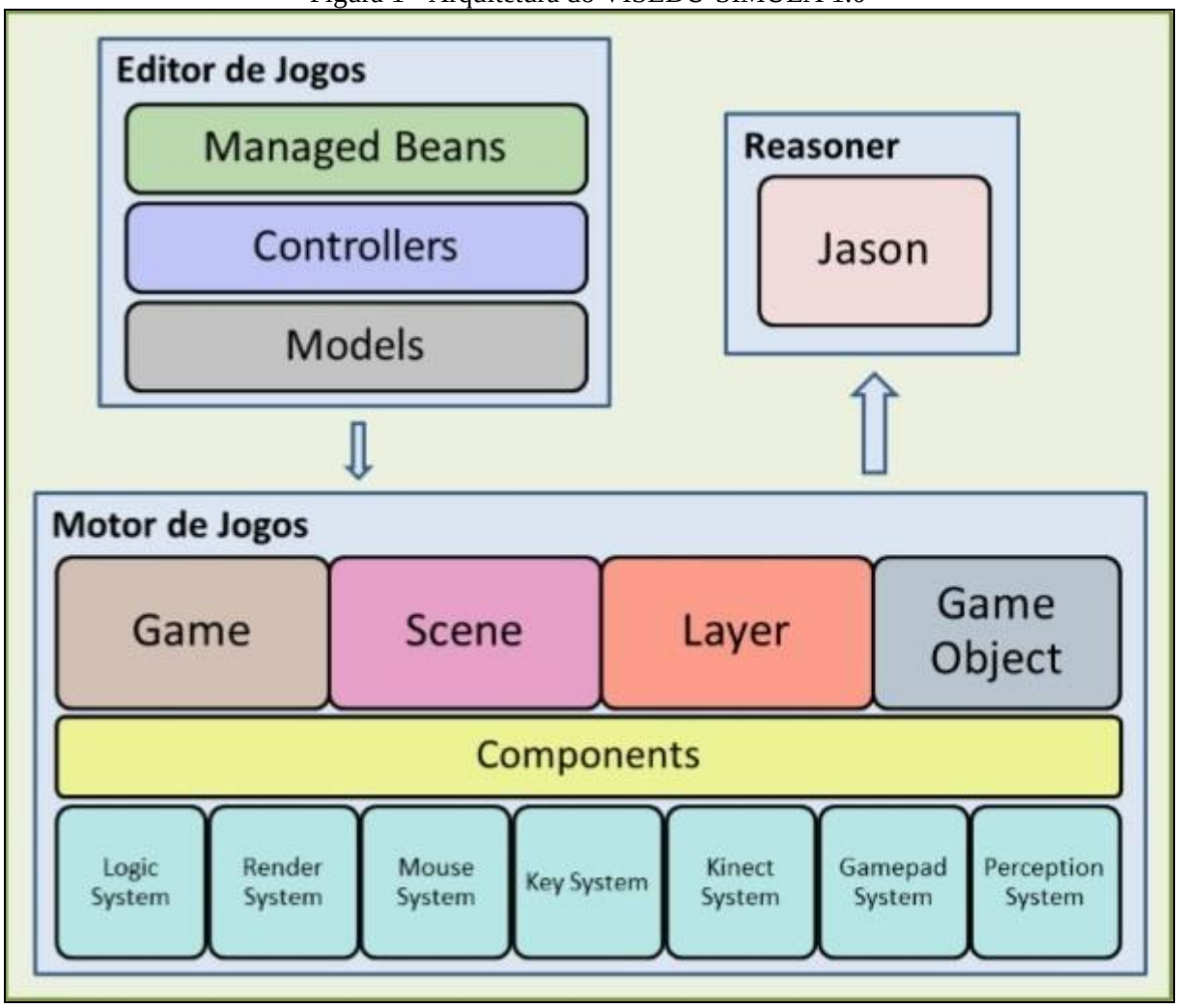
## 2.3 VISEDU-SIMULA 1.0: VISUALIZADOR DE MATERIAL EDUCACIONAL, MÓDULO DE ANIMAÇÃO COMPORTAMENTAL

O Visualizador de Material Educacional (VISEDU), é um projeto do Grupo de Pesquisa em Computação Gráfica, Processamento de Imagens e Entretenimento Digital (GCG) da Universidade Regional de Blumenau. Seu objetivo, segundo o próprio GCG (2015), é "produzir tecnologia e sistemas informatizados para facilitar a disponibilização de material educacional interdisciplinar, usando objetos de aprendizagem para facilitar a decomposição em módulos pequenos e potencialmente reutilizáveis." Um deles é o módulo de Animação Comportamental.

Segundo Feltrin (2014, p. 16), esse módulo tem por objetivo "criar um simulador 2D para geração de animações comportamentais." Mas para isso ser possível, 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 Reasoner, que por sua vez trata de preparar as informações do ambiente externo para interpretação do modelo mental Jason (FELTRIN, 2014), conforme apresenta a Figura 1.

Nessa mesma figura pode-se observar a arquitetura inteira do trabalho de Feltrin (2014), onde existem o editor de jogos e seus componentes, responsáveis por fornecer um ambiente para edição dos jogos, o motor de jogos, responsável por implementar funções que facilitam o desenvolvimento dos mesmos e o Reasoner, apresentado anteriormente.

Figura 1 - Arquitetura do VISEDU-SIMULA 1.0




Fonte: Feltrin (2014).

Para validar seu trabalho, Feltrin (2014) desenvolveu um protótipo de simulação, onde basicamente existe uma presa e um predador. Se o predador estiver dentro do campo de visão da presa e colidir com ela, o mesmo não irá consumi-la. Caso contrário, a presa será devorada (FELTRIN, 2014). Portanto, dentre as possibilidades de extensão do trabalho de Feltrin (2014), destaca-se o modelo mental simples e pouco elaborado.

Seus resultados foram satisfatórios, levando em consideração que seus objetivos foram alcançados. Tais como a finalização do simulador com um espaço bidimensional (2D) para geração de animações comportamentais, proporcionar um controle mínimo de percepção, raciocínio e atuação dos personagens e utilizar o modelo BDI para testar o simulador, mantendo compatibilidade com os principais navegadores do mercado (FELTRIN, 2014).

## 2.4 ANIMAÇÃO COMPORTAMENTAL

Animação Comportamental pode ser definida como sendo uma cena contendo personagens/objetos com características próprias, comportamentos próprios, objetivos,

restrições, que se utilizam de técnicas de Inteligência Artificial para interagir uns com os outros e também com o meio ao seu redor, de forma autônoma (MENDONÇA JR, 1999). Dessa forma, segundo Reynolds (1997, tradução nossa), “Isto dá ao personagem a capacidade de improvisar, e libera o animador da necessidade de especificar cada detalhe do movimento de cada personagem.”

Mesmo que o termo Animação Comportamental ainda não seja muito difundido, suas aplicações já geram pesquisas, especialmente na indústria de entretenimento. O jogo Creatures foi o primeiro a adicionar redes neurais, que permitem que você possa ensinar sua criatura como ela deve se comportar. O jogo foi um grande sucesso e gerou continuações. Mais recentemente, The Sims[1] focou na simulação do cotidiano de humanos virtuais, com resultados bem sucedidos e divertidos. Mesmo com comportamentos ainda limitados, esses jogos provam que as pessoas são atraídas a jogarem com agentes autônomos (MAGNENAT-THALMANN; THALMANN, 2004).

Apesar disso, agentes autônomos não estão restritos ao entretenimento. Eles podem ajudar a treinar pessoas em situações difíceis (MAGNENAT-THALMANN; THALMANN, 2004), como também nos processos de ensino e aprendizagem.

A Animação Comportamental possui conceitos semelhantes aos de agentes inteligentes: percepção, raciocínio e ação. O primeiro é um processo de reconhecimento e coleta de informações por sensores externos e enviadas para si próprio (WOOLDRIDGE; JENNINGS, 1995, p. 235). O segundo é a capacidade de tirar conclusões sobre essas informações, formando crenças e fatos (MULLER et al., 1997, p. 118). O terceiro é a reação imediata de um estímulo externo em forma de comportamento (WOOLDRIDGE; JENNINGS, 1995, p. 303).

Entre os possíveis modelos mentais utilizados para realização do raciocínio de agentes, pode-se citar o modelo BDI, caracterizado pela implementação de crenças, desejos e intenções de agentes, sendo uma das abordagens mais conhecidas para o desenvolvimento de agentes cognitivos. Baseada na arquitetura BDI, uma das linguagens orientadas a agentes mais influentes é a linguagem AgentSpeak (JASON, 2014a), popularizando-se ainda mais após o desenvolvimento de uma versão estendida da mesma pela plataforma Jason, que é um interpretador para essa versão, onde ele implementa a semântica operacional da linguagem e fornece uma plataforma para o desenvolvimento de sistemas multiagente, com muitas características customizáveis pelo usuário (JASON, 2014b).

## 2.5 TRABALHOS CORRELATOS

Foram selecionados três trabalhos correlatos, o simulador de multidões MASSIVE, desenvolvido pela Massive Software (MASSIVE, 2014a); o simulador de comportamento de peixes de Schmickl (2002) e o agente pedagógico STEVE, desenvolvido por Rickel e Johnson (1998).

### 2.5.1 MASSIVE

O Multiple Agent Simulation System In Virtual Environment (MASSIVE) é um simulador multiagente que utiliza Animação Comportamental, e segundo a própria MASSIVE (2014a, tradução nossa), "capaz de produzir uma simulação com a quantidade de agentes que for preciso". Com agentes muito simples, milhões podem ser executados em uma passagem. Já com agentes mais complexos, como humanoides, podem ser feitas várias passagens de até 100.000 agentes, permitindo que cada grupo possa ver e reagir com outros previamente simulados (MASSIVE, 2014a).

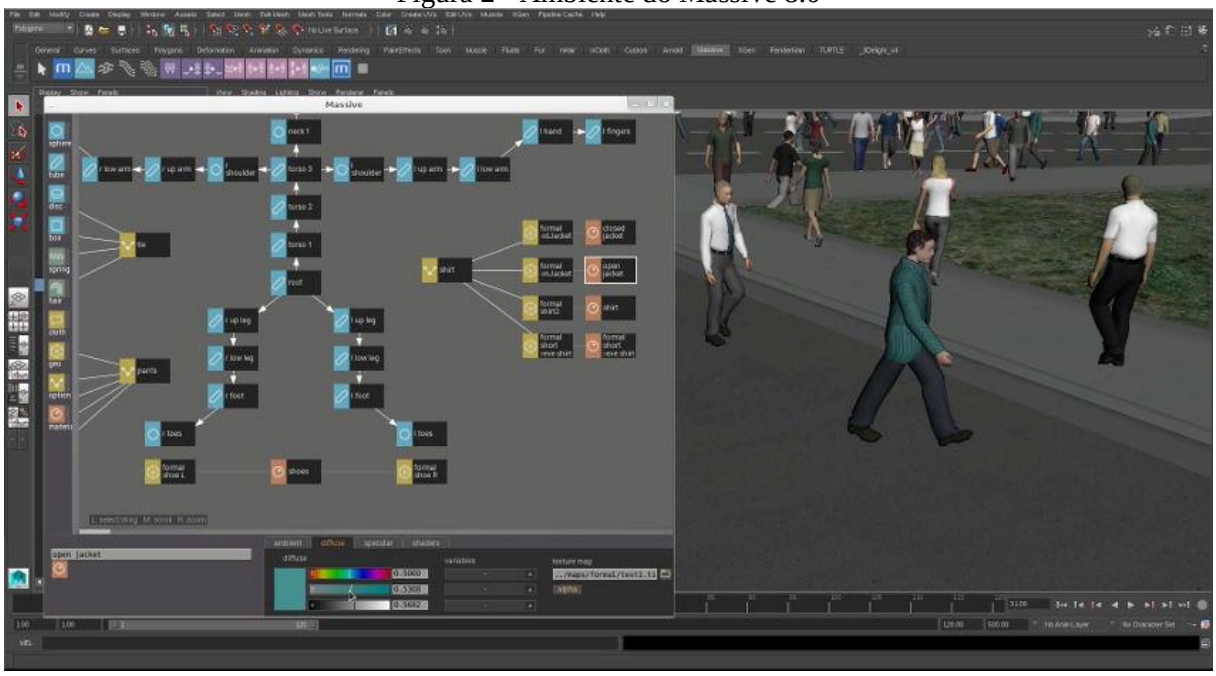
Foi originalmente desenvolvido para ser utilizado na trilogia O Senhor dos Anéis, mas posteriormente a empresa Massive Software foi criada para trazer essa tecnologia para produções de cinema e televisão. Desde então ele se tornou o software líder para geração de multidões relacionadas com efeitos visuais e animações autônomas (MASSIVE, 2014b).

Dentre alguns produtos da linha MASSIVE, destaca-se o Massive Prime, que é seu principal produto, possuindo uma interface intuitiva, baseada em nós que permitem que o usuário crie agentes inteligentes de forma interativa, com respostas personalizadas para comportamentos específicos, sem qualquer programação. Pode-se ver isso na Figura 2, onde a janela à direita mostra a "renderização" da cena e uma espécie de grafo na parte esquerda representa a programação visual dos agentes inteligentes (MASSIVE, 2014c).

---

[1] Jogo eletrônico de simulação de vida real.

Figura 2 - Ambiente do Massive 8.0



Fonte: MASSIVE (2015).

### 2.5.2 Fish School and Obstacles

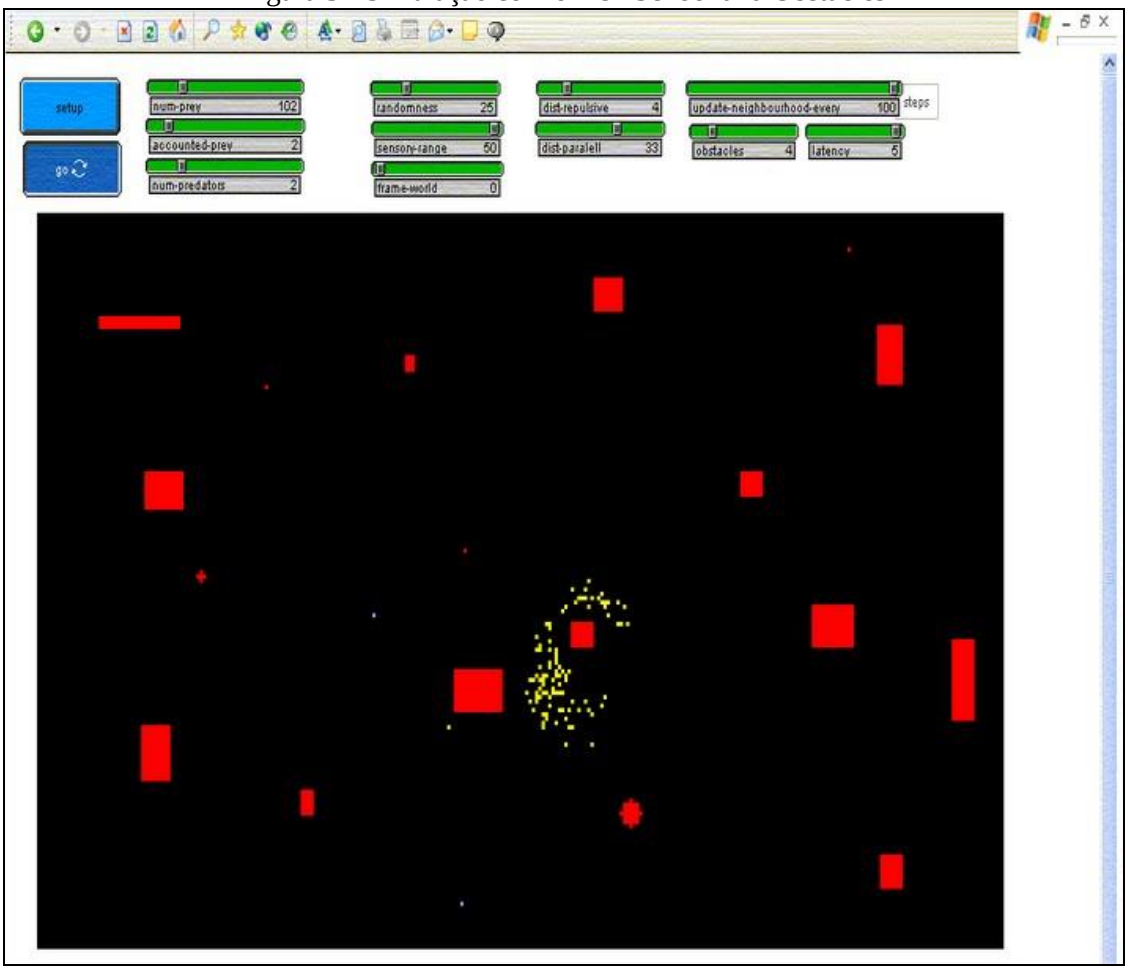
Este trabalho é um simulador desenvolvido em Java, voltado para a web, e tem como objetivo simular o comportamento de peixes diante de predadores e obstáculos. Suas principais características são a formação de cardumes, evitar obstáculos e predadores. Basicamente os predadores e as presas ficam se movendo aleatoriamente, porém os peixes possuem comportamentos individuais que podem influenciar o comportamento do cardume (SCHMICKL, 2002).

O comportamento do peixe (presa) se dá da seguinte forma. A cada determinado número de passos, cada presa atualiza sua lista de vizinhos visíveis, que são todos os peixes dentro do alcance configurado pelo simulador. Após isso, alguns peixes são retirados aleatoriamente da visão dos vizinhos, onde cada um desses peixes tem uma influência sobre os outros. Se o peixe escolhido estiver muito próximo dos outros, os outros se afastarão um pouco, caso estiver muito longe, os outros se aproximarão, caso estiverem com uma distância aceitável, eles se alinharão. Porém quando um predador ou um obstáculo entra no campo de visão das presas, as mesmas se afastam brutalmente. Já o comportamento do predador é simples, ele se movimenta aleatoriamente, e quando as presas entram no seu campo de visão ele vai em direção às mesmas (SCHMICKL, 2002).

O simulador traz características reais do comportamento dos peixes, porém sua representação gráfica é pobre, trazendo gráficos em duas dimensões com quadrados e retângulos representando peixes e obstáculos, além de usar somente três cores, sem considerar

as cores de fundo e dos componentes. A Figura 3 apresenta o ambiente visual desse simulador, onde os obstáculos estão em vermelho, as presas em amarelo, os predadores em azul e na parte superior da tela as barras de configuração do simulador, permitindo que o usuário defina a quantidade de presas, predadores, obstáculos, a distância do alcance de visão das presas, entre outras funcionalidades.

Figura 3 - Simulação com o Fish School and Obstacles




Fonte: Schmickl (2002).

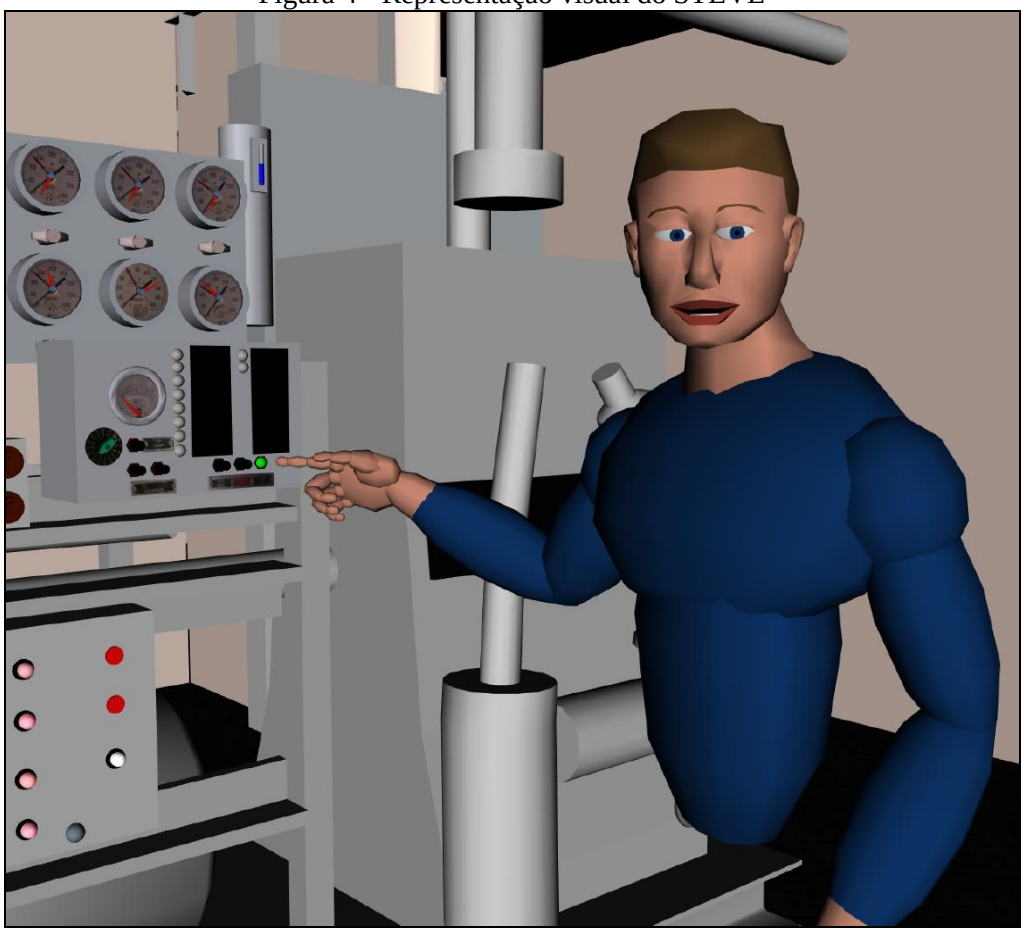
### 2.5.3 STEVE

*O Soar Training Expert for Virtual Environments* (STEVE) é um tutor virtual em forma de humano que pode interagir com estudantes, os ajudando a aprender e podendo demonstrar ações, usar o olhar e gestos para direcionar a atenção do aluno, orientando o mesmo dentro do mundo virtual, conforme apresenta a Figura 4. Além de poder substituir alunos reais em tarefas de equipe (RICKEL; JOHNSON, 1998).

Ele consiste em dois componentes principais, o primeiro que lida com o processamento cognitivo de alto nível e o segundo que trata o processamento sensório-motor. O componente cognitivo interpreta o estado do mundo virtual, executa planos para atingir metas e toma

decisões sobre as ações a tomar. Já o componente sensório-motor funciona como uma interface com o mundo virtual, permitindo que o componente cognitivo perceba o estado do mundo ao seu redor e que cause mudanças no mesmo (RICKEL; JOHNSON, 1998).

Figura 4 - Representação visual do STEVE



Fonte: Information Sciences Institute (2000).

# 3 DESENVOLVIMENTO

São abordadas nesse capítulo as etapas do desenvolvimento do Aquário Virtual e a extensão dos módulos de Computação Gráfica e Animação Comportamental do VISEDU, desenvolvidos por Koehler (2015) e Feltrin (2014) respectivamente. São abordados os principais requisitos, a especificação, a implementação, os resultados e as discussões.

## 3.1 REQUISITOS

O Aquário Virtual deve:

a) trabalhar em espaço de três dimensões (3D) (Requisito Não Funcional - RNF);
b) utilizar o Reasoner para gerenciar a animação comportamental (RNF);
c) ser compatível com os mesmos navegadores que os módulos desenvolvidos por Feltrin (2014) e Koehler (2015) (RNF);
d) implementar controle orbital de câmera (RF);
e) ser implementado com as tecnologias HTML5, Javascript e CSS (RNF);
f) utilizar a biblioteca gráfica ThreeJS (RNF);
g) possuir ao menos um predador e uma presa (RF);
h) possuir uma câmera auxiliar para exibir a visão dos peixes (RF);
i) permitir que somente a visão do peixe selecionado apareça na câmera auxiliar (RF);
j) remover a câmera auxiliar quando nenhum peixe estiver selecionado (RF);
k) possuir uma área de texto para informar os comportamentos enviados pelo Reasoner (RF);
l) permitir que somente os comportamentos do peixe selecionado apareçam na área de texto (RF);
m) limpar a área de texto caso nenhum peixe estiver selecionado (RF);
n) garantir que nenhum peixe saia dos limites do aquário (RF);
o) permitir que o usuário altere propriedades dos peixes, do aquário e do mundo (RF);
p) permitir que a alteração das propriedades reflitam em tempo real no aquário (RNF);
q) permitir a inclusão e remoção de peixes (RF);
r) excluir as presas devoradas do aquário e da árvore de peças (RF);
s) permitir a procriação de presas e predadores (RF);
t) excluir predadores que não se alimentem até determinado tempo (RF);

u) aumentar a população de plâncton caso a população de sardinhas seja baixa e diminuir caso seja alta. Também deve refletir na cor da água do aquário, caso haja muitos plânctons a água deve ficar esverdeada, caso contrário azulada (RF);
v) bloquear a procriação de presas caso a população de plâncton for muito baixa (RF).

O Reasoner deve:

a) utilizar o modelo BDI (RNF);
b) utilizar o interpretador Jason para gerenciar o modelo BDI (RNF);
c) ser desenvolvido com a linguagem de programação Java (RNF);
d) permitir a comunicação de vários agentes e de mais de um tipo com o interpretador Jason (RF).

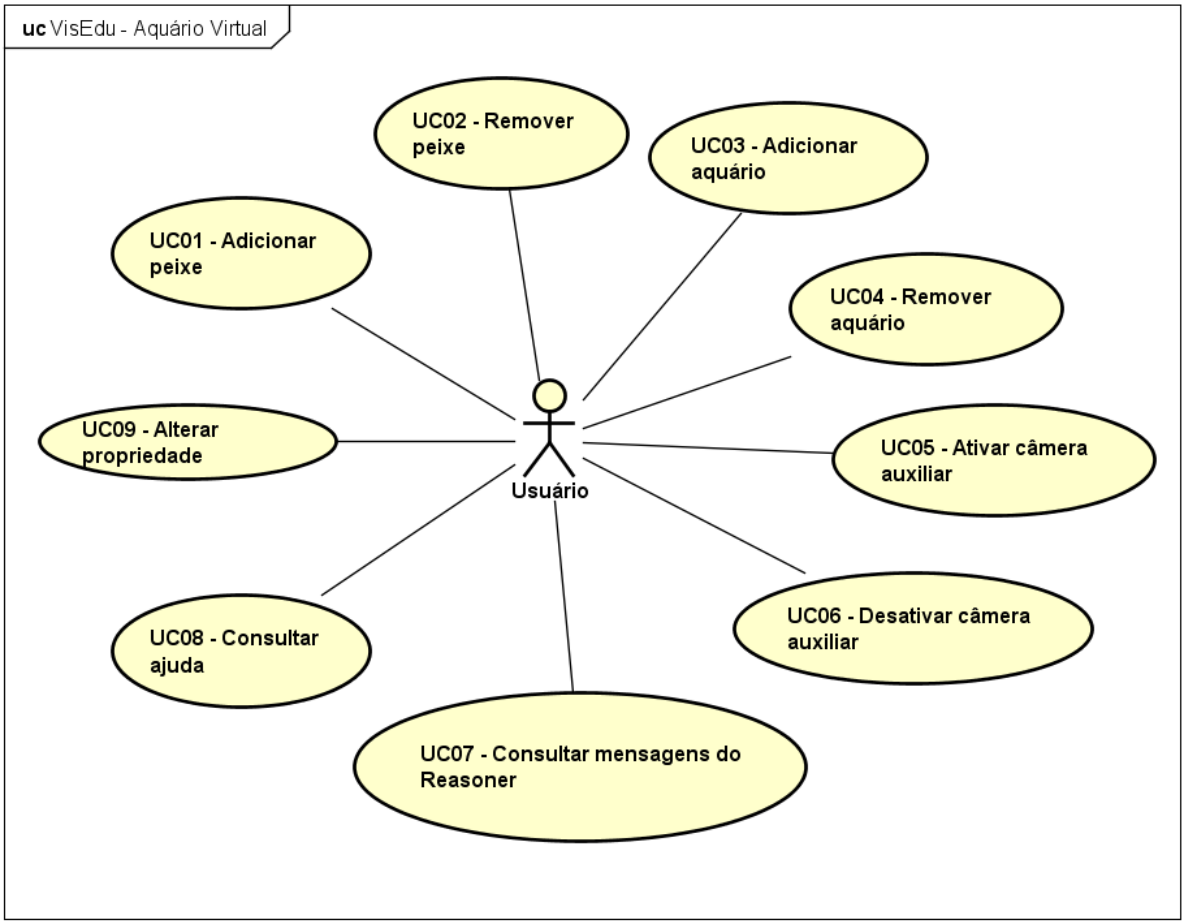
## 3.2 ESPECIFICAÇÃO

Para realizar a especificação do Aquário Virtual, do Motor de Jogos e do Reasoner, utilizou-se a *Unified Modeling Language* (UML), sendo criados diagramas de caso de uso, de pacotes e de classes com a ferramenta Astah versão *Community*, além de diagramas de arquitetura com a ferramenta draw.io. Nas próximas seções são apresentados os respectivos diagramas.

### 3.2.1 Diagrama de caso de uso do Aquário Virtual

Conforme pode ser visto na Figura 5, foram criados nove casos de uso para o ator `Usuário`. Os casos de uso foram criados com base nos requisitos funcionais do Aquário Virtual.

Figura 5 - Diagrama de caso de uso do Aquário Virtual




No Quadro 1 é apresentado o detalhamento do primeiro caso de uso operado pelo ator `Usuário`.

Quadro 1 - Caso de uso `UC01: Adicionar peixe`

| UC01 – Adicionar peixe | |
|---|---|
| Descrição | Permite que o `Usuário` adicione um peixe ao Aquário Virtual. |
| Cenário Principal | 1. `Usuário` abre a aba `Fábrica de Peças`;<br>2. `Usuário` arrasta a peça `Tubarão` ou `Sardinha` para a árvore de peças. |
| Pré-Condição | A aplicação deve possuir um aquário adicionado. |

No Quadro 2 é apresentado o detalhamento do segundo caso de uso operado pelo ator `Usuário`.

Quadro 2 - Caso de uso `UC02: Remover peixe`

| UC02 – Remover peixe | |
|---|---|
| Descrição | Permite que o `Usuário` remova um peixe ao Aquário Virtual. |
| Cenário Principal | 1. `Usuário` arrasta a peça correspondente a um peixe contida na árvore de peças para a lixeira. |
| Pré-Condição | A aplicação deve possuir um peixe adicionado. |

No Quadro 3 é apresentado o detalhamento do terceiro caso de uso operado pelo ator `Usuário`.

Quadro 3 - Caso de uso UC03: Adicionar aquário

| UC03 – Adicionar aquário | |
|---|---|
| Descrição | Permite que o Usuário adicione um aquário ao Aquário Virtual. |
| Cenário Principal | 1. Usuário abre a aba Fábrica de Peças;<br>2. Usuário arrasta a peça Aquário para a árvore de peças. |
| Pós-Condição | O Usuário possui um aquário capaz de receber peixes. |

No Quadro 4 é apresentado o detalhamento do quarto caso de uso operado pelo ator Usuário.

Quadro 4 - Caso de uso UC04: Remover aquário

| UC04 – Remover aquário | |
|---|---|
| Descrição | Permite que o Usuário remova um aquário do Aquário Virtual. |
| Cenário Principal | 1. Usuário arrasta a peça correspondente a um aquário contida na árvore de peças para a lixeira. |
| Pré-Condição | A aplicação deve possuir um aquário adicionado. |
| Pós-Condição | Peixes contidos no aquário também são removidos. |

No Quadro 5 é apresentado o detalhamento do quinto caso de uso operado pelo ator Usuário.

Quadro 5 - Caso de uso UC05: Ativar câmera auxiliar

| UC05 – Ativar câmera auxiliar | |
|---|---|
| Descrição | Permite que o Usuário ative a câmera auxiliar. |
| Cenário Principal | 1. Usuário clica na peça de determinado peixe na árvore de peças que não esteja selecionado. |
| Pré-Condição | A aplicação deve possuir um peixe adicionado. |
| Pós-Condição | O Usuário possui a exibição da visão do peixe. |

No Quadro 6 é apresentado o detalhamento do sexto caso de uso operado pelo ator Usuário.

Quadro 6 - Caso de uso UC06: Desativar câmera auxiliar

| UC06 – Desativar câmera auxiliar | |
|---|---|
| Descrição | Permite que o Usuário desative a câmera auxiliar. |
| Cenário Principal | 1. Usuário clica na peça selecionada na árvore de peças. |
| Cenário Alternativo | 1. Usuário clica na peça correspondente ao mundo ou ao aquário na árvore de peças. |
| Pré-Condição | A aplicação deve estar com a câmera auxiliar habilitada. |

No Quadro 7 é apresentado o detalhamento do sétimo caso de uso operado pelo ator Usuário.

Quadro 7 - Caso de uso UC07: Consultar mensagens do Reasoner

| UC07 – Consultar mensagens do Reasoner | |
|---|---|
| Descrição | Permite que o Usuário visualize mensagens enviadas pelo Reasoner. |
| Cenário Principal | 1. Usuário clica na peça de determinado peixe na árvore de peças que não esteja selecionado. |
| Pré-Condição | A aplicação deve possuir um peixe adicionado. |
| Pós-Condição | O Usuário pode visualizar as mensagens enviadas pelo Reasoner na área de texto abaixo da árvore de peças. |

No Quadro 8 é apresentado o detalhamento do oitavo caso de uso operado pelo ator Usuário.

Quadro 8 - Caso de uso UC08: Consultar ajuda

| UC08 – Consultar ajuda | |
|---|---|
| Descrição | Permite que o Usuário visualize informações sobre a aplicação. |
| Cenário Principal | 1. Usuário clica na aba Ajuda. |
| Pós-Condição | O Usuário pode visualizar tanto informações sobre a aplicação quanto conceitos sobre o conteúdo biológico abordado. |

No Quadro 9 é apresentado o detalhamento do último caso de uso operado pelo ator Usuário.

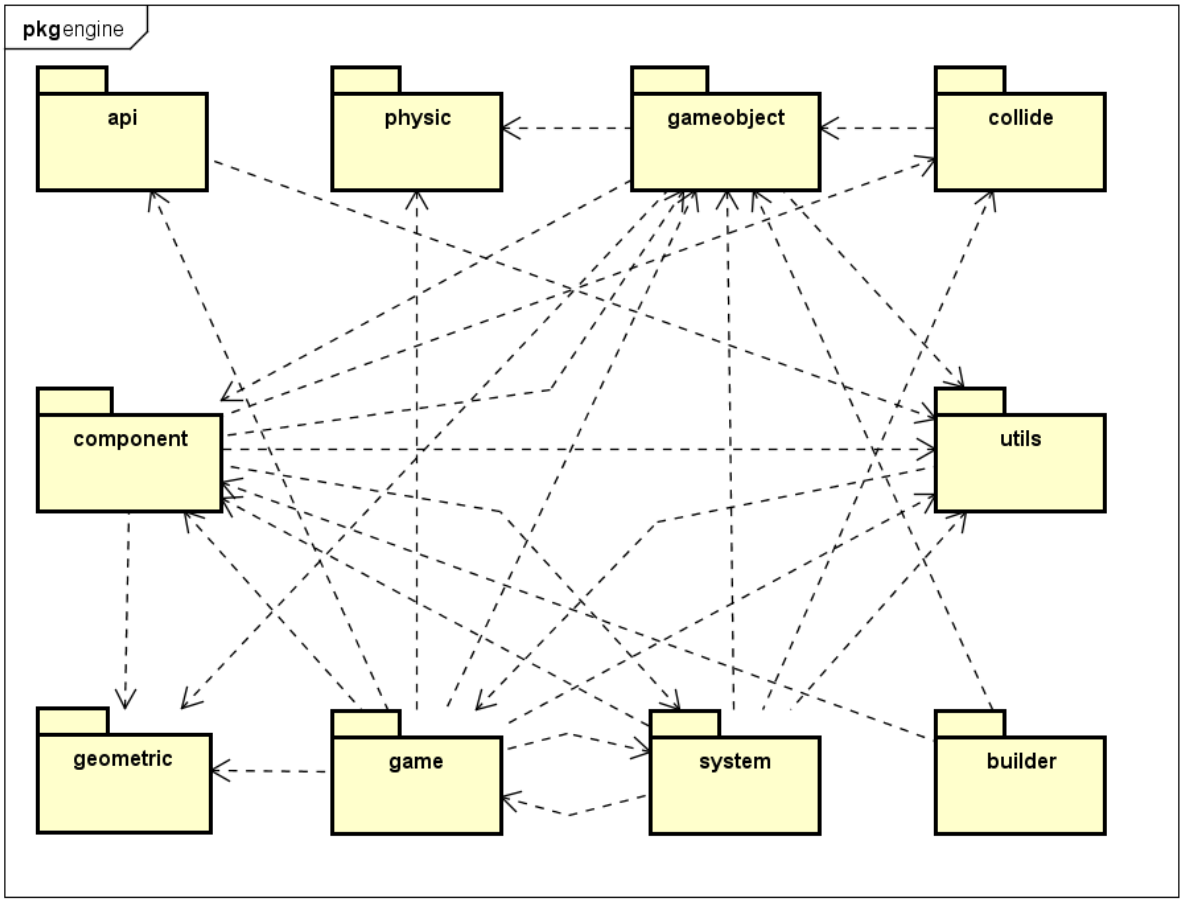
Quadro 9 - Caso de uso UC09: Alterar propriedade

| UC09 – Alterar propriedade | |
|---|---|
| Descrição | Permite que o Usuário altere alguma propriedade de determinada peça. |
| Cenário Principal | 1. Usuário clica em determinada peça presente na árvore de peças;<br>2. Usuário localiza a propriedade desejada na aba Propriedades da Peça;<br>3. Usuário altera o valor da propriedade e aperta a tecla *enter*. |
| Pré-Condição | A aplicação deve possuir um peixe adicionado. |
| Pós-Condição | O Usuário pode visualizar o resultado da alteração em tempo real na aplicação. |

### 3.2.2 Diagrama de pacotes do Motor de Jogos

Nessa seção são descritos os pacotes e as classes do Motor de Jogos que foram alteradas. Na Figura 6 pode-se observar o diagrama de pacotes do Motor de Jogos utilizado.

Figura 6 - Diagrama de pacotes do Motor de Jogos

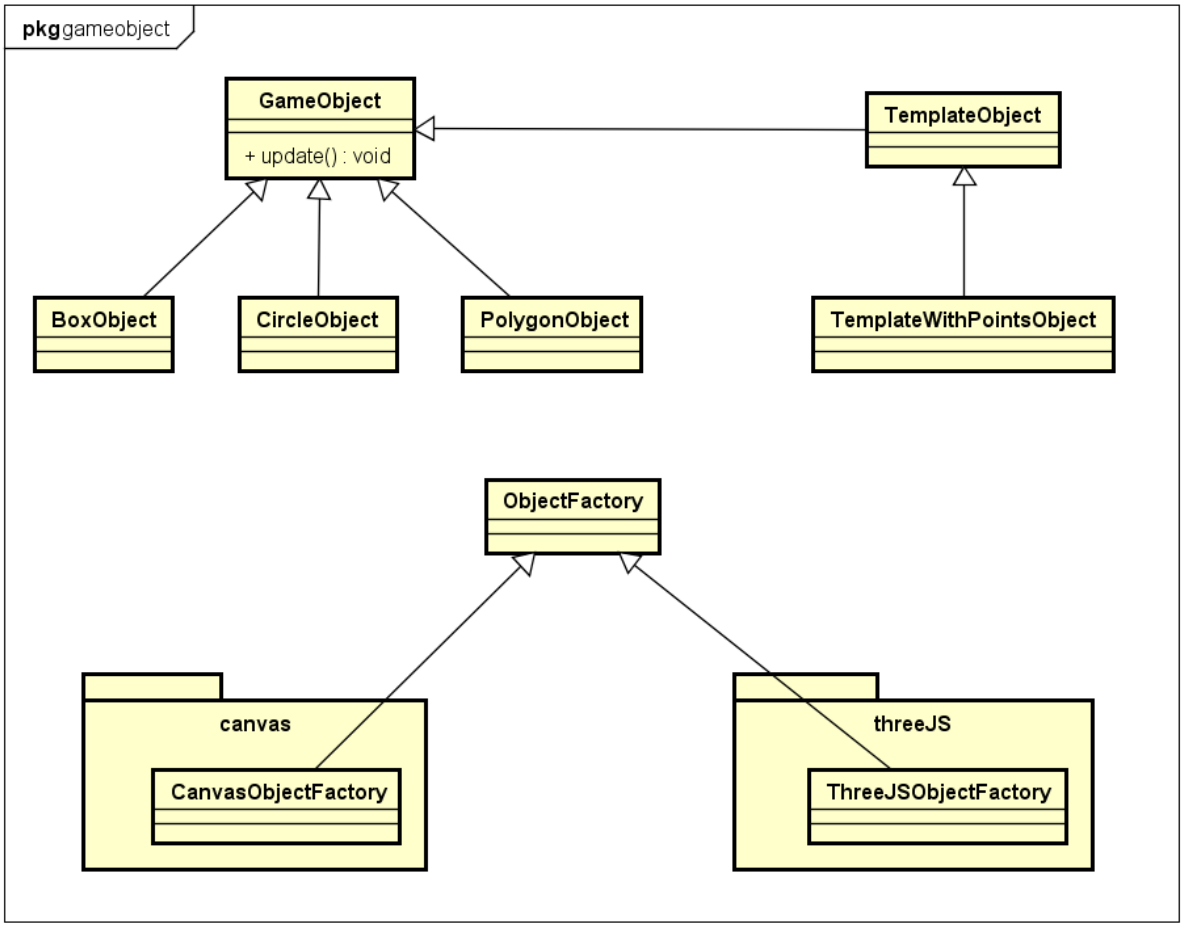



O Motor de Jogos originalmente foi desenvolvido por Harbs (2013) e estendido por Koehler (2015), dentre esses pacotes, foram alterados somente os pacotes `gameobject` e `utils`. As seções 3.2.2.1 e 3.2.2.2 descrevem as classes que compõe os pacotes manipulados para essa aplicação.

### 3.2.2.1 Pacote `gameobject`

Conforme Harbs (2013, p. 28), "o pacote denominado `gameobject` possui a classe necessária para definir um objeto do jogo dentro do motor assim como alguns objetos disponibilizados previamente." Esse pacote também possui uma classe abstrata capaz de gerar objetos especializados tanto para o `canvas` do HTML5 quanto para o ThreeJS, garantindo compatibilidade entre ambos (KOEHLER, 2015, p. 35). O diagrama de classes desse pacote pode ser visto na Figura 7.

Figura 7 - Diagrama de classes do pacote `gameobject`

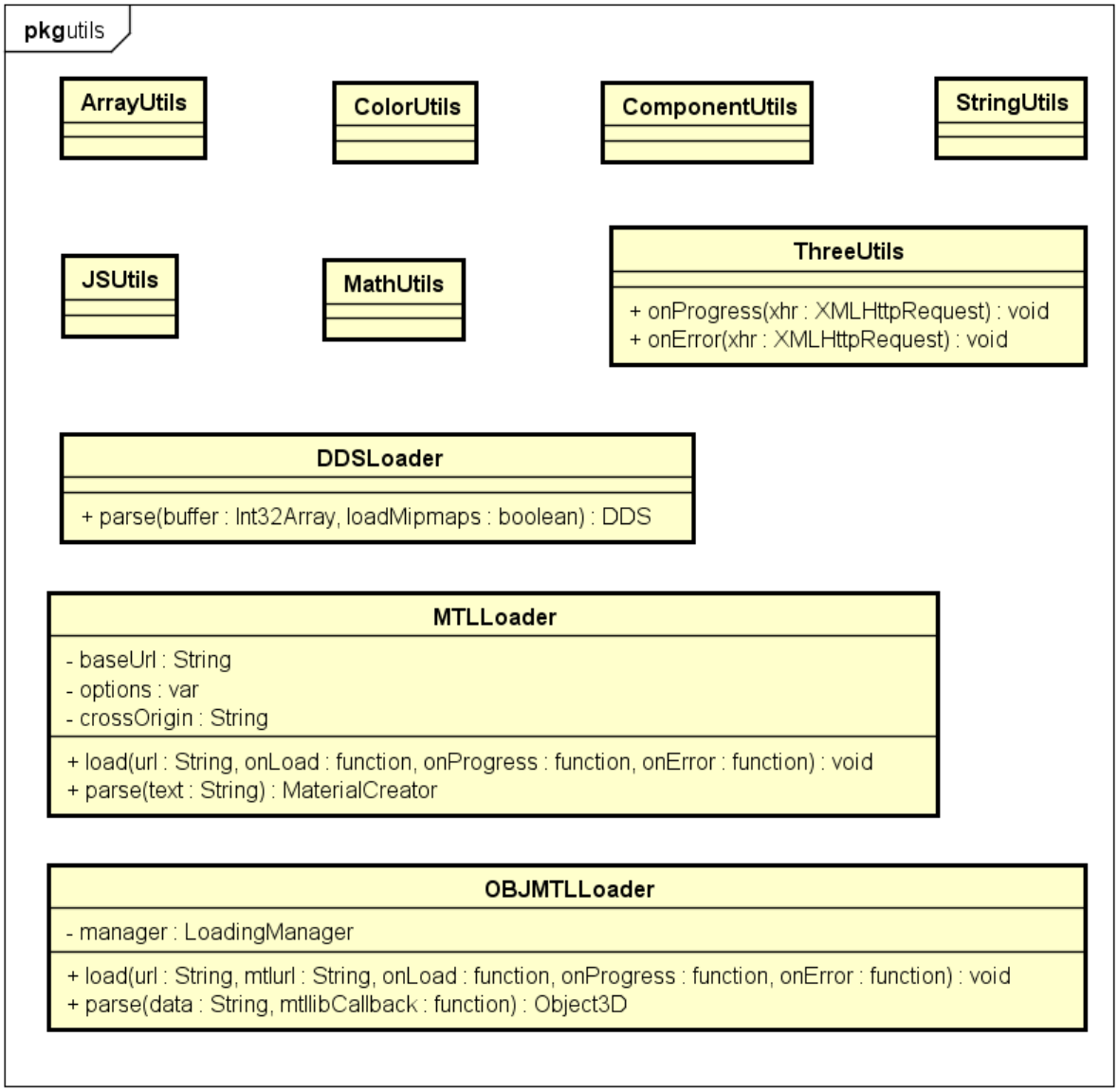



A classe `GameObject` ganhou a função `update`, responsável por permitir e facilitar a animação de objetos em tempo real, dessa maneira cada objeto pode controlar sua animação.

#### 3.2.2.2 Pacote `utils`

Originalmente o pacote `utils` foi criado para organizar classes utilitárias para operações Javascript (HARBS, 2013, p. 31). Para o desenvolvimento dessa aplicação foi necessário incluir utilitários para o carregamento de arquivos que contém informações de objetos em três dimensões, tais como as classes `DDSLoader`, `MTLLoader` e `OBJMTLLoader`, disponibilizadas pelos desenvolvedores do ThreeJS. Também foram incluídas as funções `onError` e `onProgress` na classe `ThreeUtils`. O diagrama de classes desse pacote pode ser visto na Figura 8.

Figura 8 - Diagrama de classes do pacote `utils`

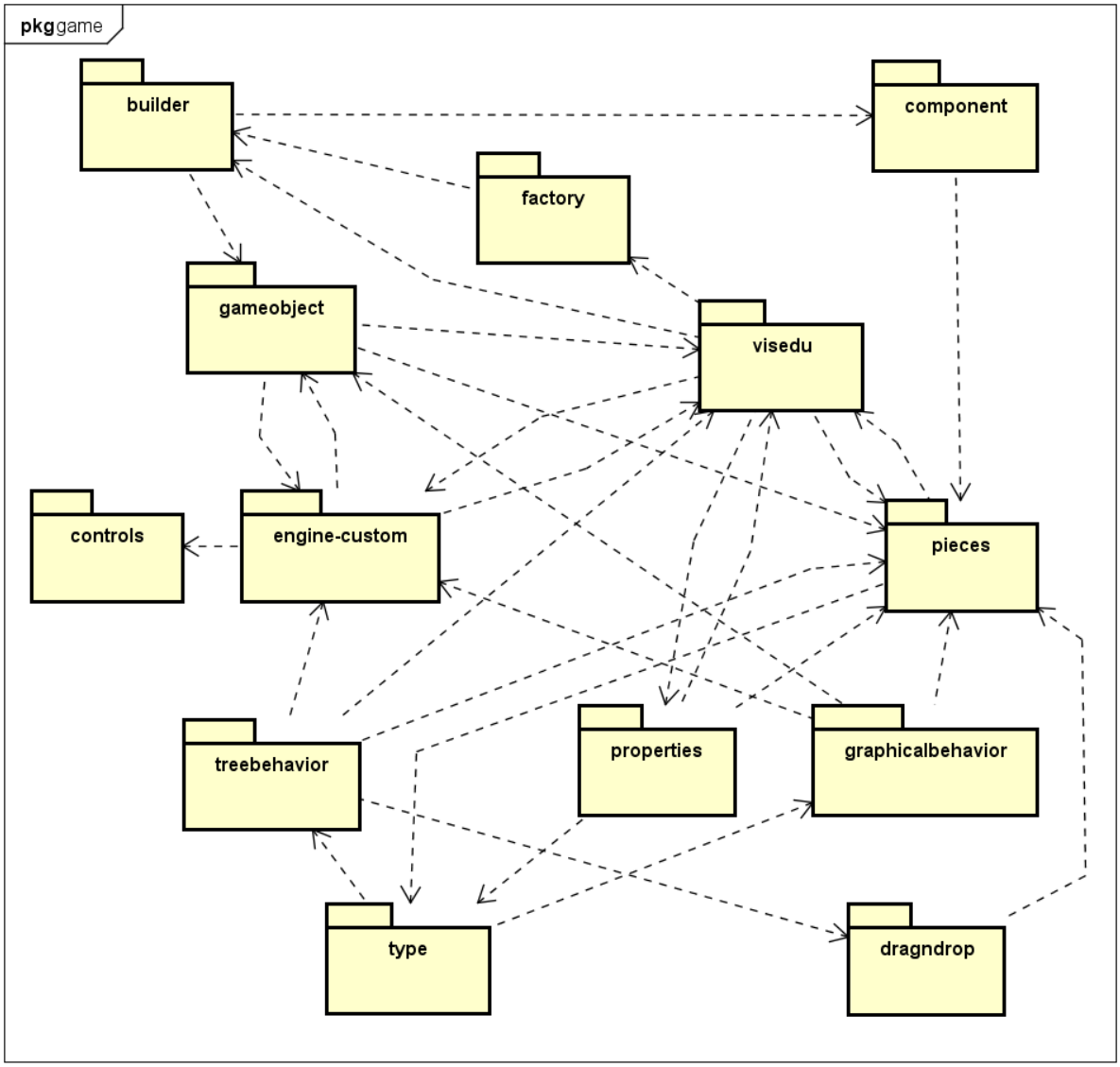



A classe `DDSLoader` é responsável por gerenciar o carregamento de texturas, enquanto que a classe `MTLLoader` se responsabiliza pelo carregamento de materiais. A classe `OBJMTLLoader` faz o carregamento de objetos em três dimensões baseado em um arquivo com extensão OBJ, que armazena informações sobre a malha tridimensional e outro arquivo de extensão MTL, que armazena informações do material utilizado pelo objeto. Para seu funcionamento ela utiliza internamente as duas classes mencionadas anteriormente. A função `onProgress` é utilizada para informar o progresso de carregamento de arquivos OBJ, enquanto que a função `onError` é responsável por informar erros durante o carregamento.

### 3.2.3 Diagrama de pacotes do Aquário Virtual

Nessa seção são descritos os pacotes e as classes do Aquário Virtual. Na Figura 9 pode-se observar o diagrama de pacotes do Aquário Virtual.

Figura 9 - Diagrama de pacotes do Aquário Virtual

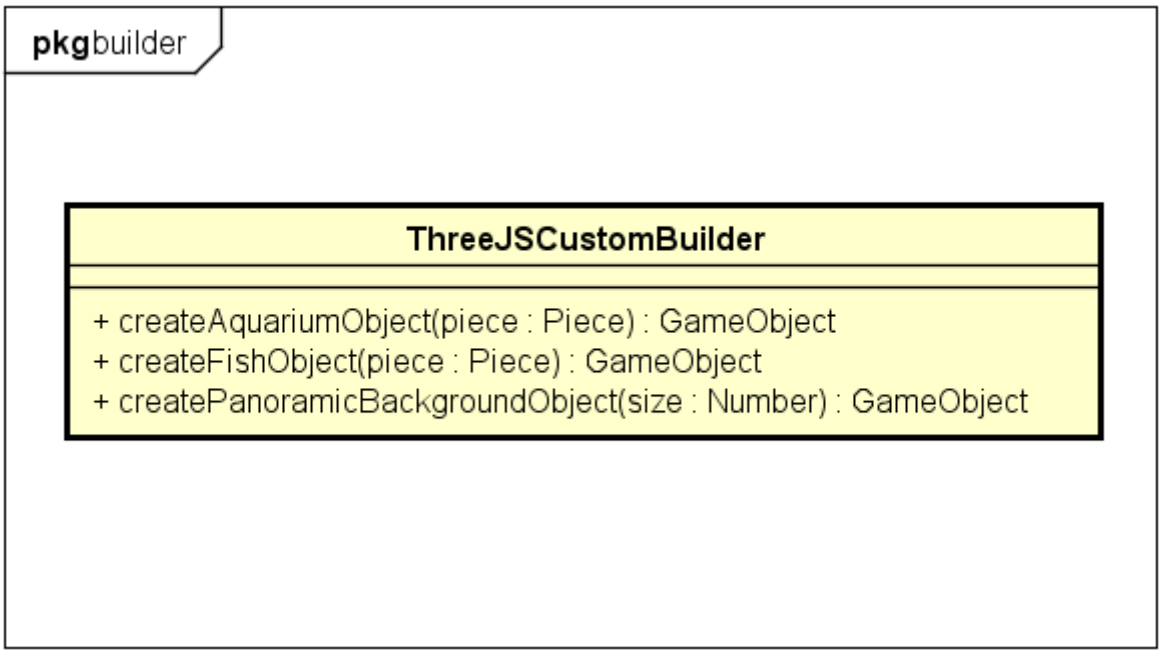



Como base para essa aplicação foi utilizado o módulo de Computação Gráfica do VISEDU, com sua última versão desenvolvida por Koehler (2015). Os pacotes `builder`, `component` e `gameobject` foram criados enquanto que os únicos que não sofreram alterações foram os pacotes `controls`, `dragndrop e properties`. As seções a seguir descrevem as classes que compõe os pacotes manipulados para essa aplicação.

### 3.2.3.1 Pacote `builder`

O pacote `builder` comporta uma classe criada para auxiliar o processo de criação de objetos específicos do Aquário Virtual. O diagrama de classes do pacote pode ser observado na Figura 10.

Figura 10 - Diagrama de classes do pacote `builder`

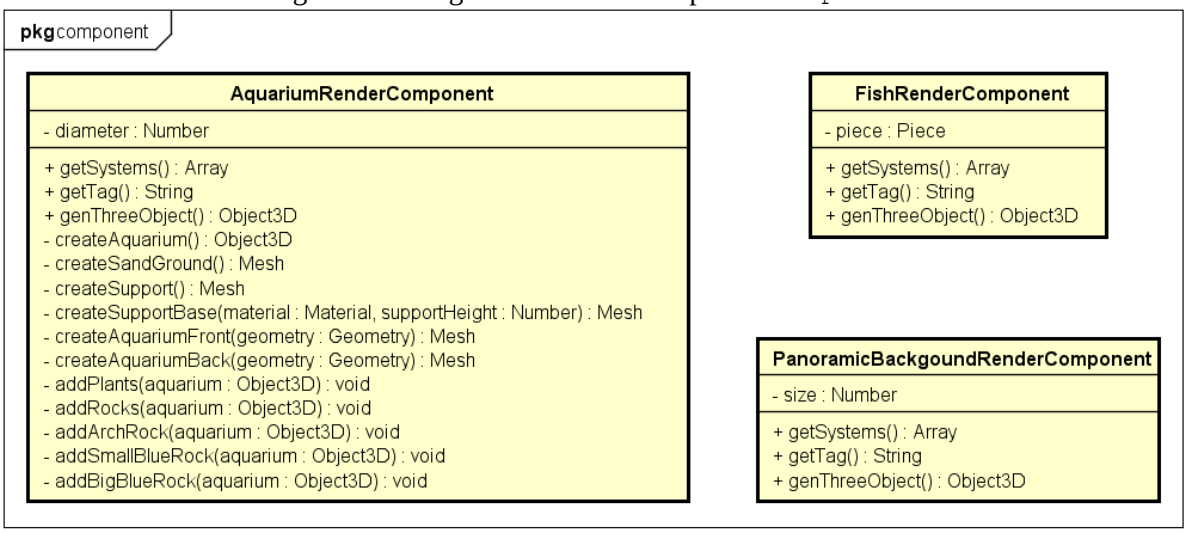



A classe `ThreeJSCustomBuilder` estende a classe `ThreeJSBuilder` do Motor de Jogos e é responsável por centralizar a criação de objetos específicos para o uso no Aquário Virtual, como é o caso dos objetos aquário, peixe e fundo panorâmico, representados pelas funções `createAquariumObject`, `createFishObject` e `createPanoramicBackgroundObject`, respectivamente. A primeira função recebe o parâmetro `piece`, responsável pela peça que representa o aquário. A segunda também recebe um parâmetro `piece`, responsável pela peça que representa o peixe a ser criado dentro do aquário, contendo suas informações e atributos. A terceira e última recebe o parâmetro `size`, que representa o tamanho do fundo panorâmico.

#### 3.2.3.2 Pacote `component`

O pacote `component` é responsável por armazenar componentes utilizados para adicionar comportamentos aos objetos, essa forma de funcionamento foi herdada do Motor de Jogos. Por exemplo, pode-se adicionar um componente de rotação para permitir que o objeto seja rotacionado no ambiente virtual. Na Figura 11 pode-se observar o diagrama de classes desse pacote.

Figura 11 - Diagrama de classes do pacote `component`




Ambas as classes foram herdadas da classe `RenderableComponent` do Motor de Jogos, ou seja, elas são responsáveis por desenhar algo na tela. A classe `AquariumRenderComponent` é responsável por desenhar o aquário e possui o atributo `diameter` que irá definir o diâmetro do aquário. Enquanto que a classe `FishRenderComponent` se encarrega de desenhar um peixe e possui o atributo `piece` que armazena informações e atributos do peixe a ser desenhado. Por fim a classe `PanoramicBackgroundRenderComponent` assume o desenho do fundo panorâmico e possui o atributo `size` que define o tamanho do mesmo.

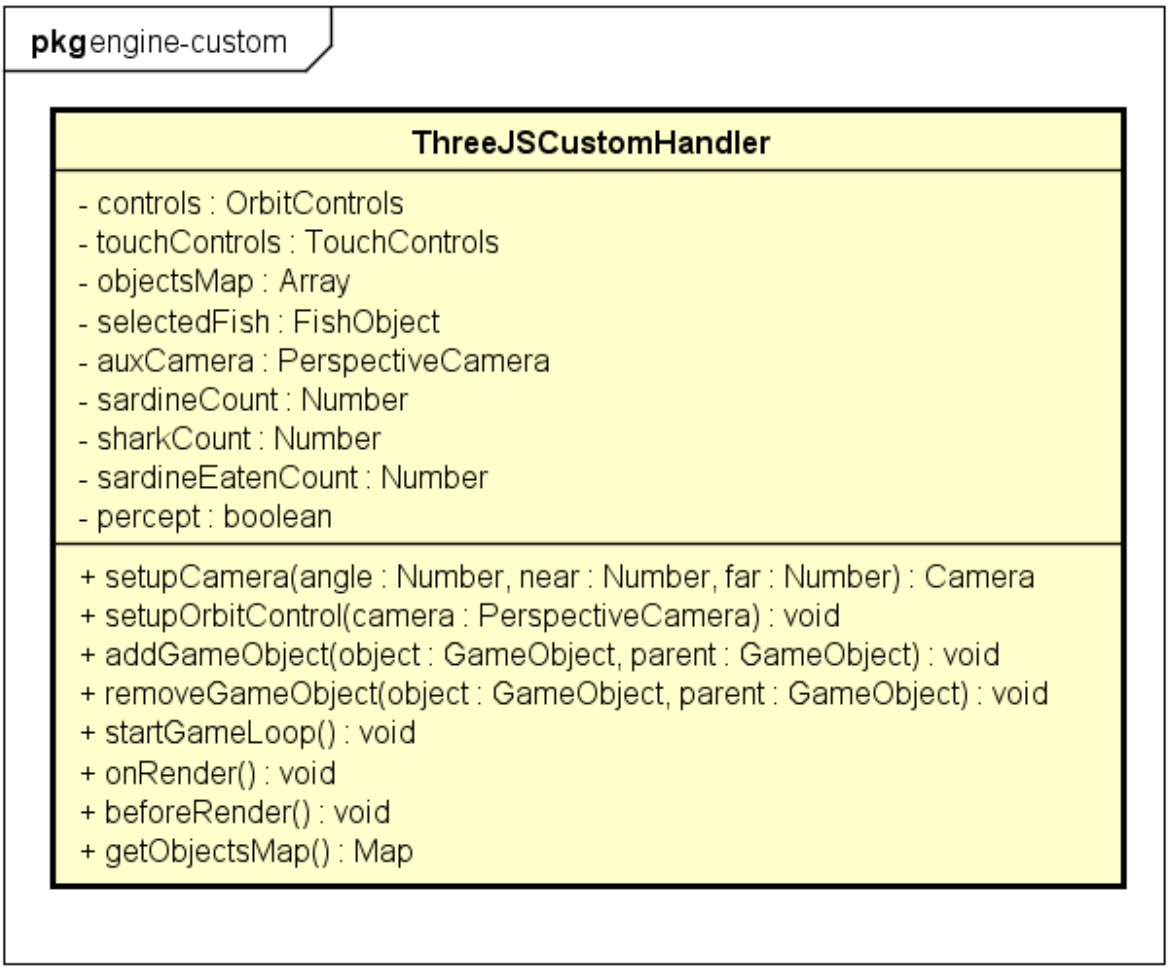
As três contam com a função `getSystems` que retornará qual o sistema do motor de jogos que deve gerenciar esse componente, nesses três casos é o `RenderSystem`. A função `getTag` também pertencente as três classes e tem como objetivo informar uma palavra-chave responsável por identificar o componente. Por último, a função `genThreeObject` que pertence as três classes, é responsável por gerar um objeto em três dimensões específico para ser utilizado na biblioteca gráfica ThreeJS.

A classe `AquariumRenderComponent` possui as funções privadas `createAquarium`, `createSandGround`, `createSupport`, `createSupportBase`, `createAquariumFront`, `createAquariumBack`, `addPlants`, `addRocks`, `addArchRock`, `addSmallBlueRock` e `addBigBlueRock`. A primeira tem a função de criar o aquário como um todo, chamando as funções necessárias para construir as partes envolvidas. A segunda cria o chão de areia, a terceira o suporte do aquário, a quarta a base do suporte. A quinta a parte de fora do vidro do aquário, a sexta a parte de dentro do vidro do aquário, a sétima centraliza a inclusão de plantas no aquário. A oitava centraliza a inclusão das rochas, a nona adiciona a rocha arcada, a décima adiciona a rocha azul pequena e a última adiciona a rocha azul grande.

#### 3.2.3.3 Pacote `engine-custom`

O pacote `engine-custom` armazena a classe responsável por fazer manipulações customizadas do Aquário Virtual na biblioteca gráfica ThreeJS. O diagrama de classes do pacote pode ser visto na Figura 12.

Figura 12 - Diagrama de classes do pacote `engine-custom`



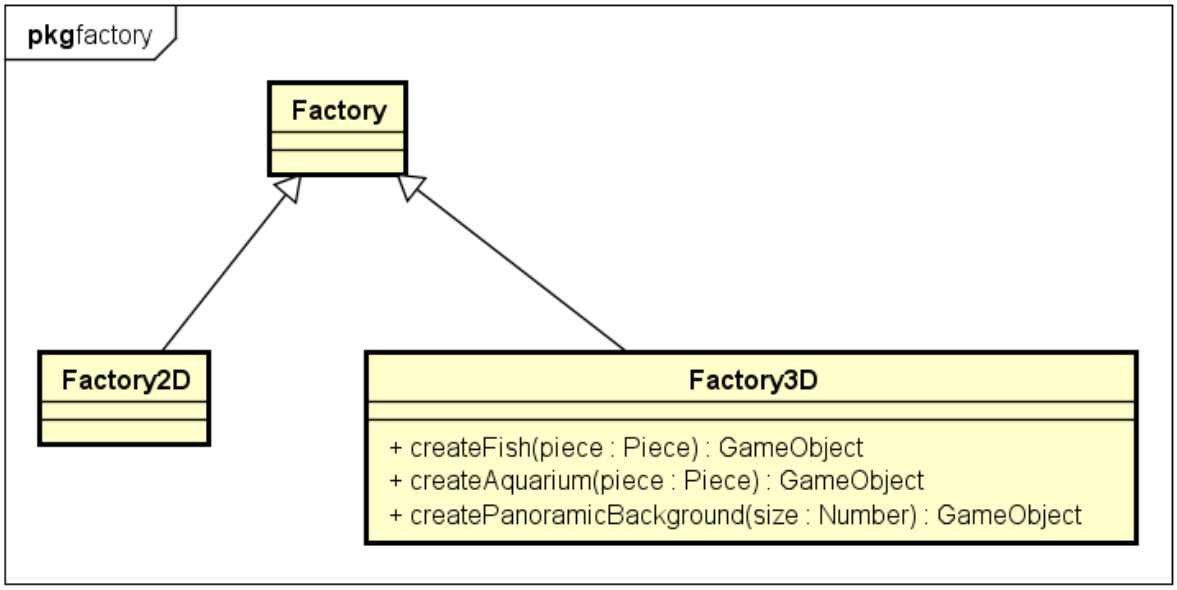

A classe `ThreeJSCustomHandler` estende a classe `ThreeJSHandler` do Motor de Jogos e é responsável por gerenciar tudo que deve ser encaminhado para a biblioteca gráfica ThreeJS, como objetos a serem desenhados e a atualização dos controles da câmera. Ela possui os atributos `controls`, `touchControls`, `objectsMap`, `selectedFish`, `auxCamera`, `sardineCount`, `sharkCount`, `sardineEatenCount` e `percept`. Os dois primeiros representam o controle orbital da câmera e o controle de toque respectivamente. O terceiro é responsável por mapear os objetos do Aquário Virtual pelo nome, garantindo acesso rápido. O quarto armazena a referência do peixe selecionado, o quinto representa a câmera auxiliar, responsável por exibir a visão do peixe. O sexto armazena a quantidade de sardinhas no aquário, o sétimo a quantidade de tubarões, o oitavo a quantidade de sardinhas devoradas e o último indica se deve ativar a percepção dos seres virtuais.

A classe possui a função `setupCamera`, que tem por finalidade a criação e configuração da câmera principal do simulador, também possui a função `setupOrbitControl`, responsável por criar e configurar os controles da câmera principal. Outra função é `addGameObject`, que adiciona um objeto ao mapa de objetos mencionado anteriormente e ao grafo de cena. Também a função `removeGameObject`, responsável por remover o objeto do grafo de cena e do mapa de objetos. A função `getObjectsMap` retorna o mapa de objetos e as outras três funções, `startGameLoop`, `onRender` e `beforeRender` são sobrescritas da classe `ThreeJSHandler`. Das quais a primeira inicializa o ciclo de desenho, a segunda é responsável por desenhar e atualizar os objetos e a terceira prepara tudo que for necessário antes da função `onRender` ser chamada.

#### 3.2.3.4 Pacote `factory`

O pacote `factory` contém as classes responsáveis por fabricar objetos. Conforme pode-se observar na Figura 13, existe a classe `Factory` que possui atributos e funções em comum tanto para o contexto de duas dimensões quanto para o de três dimensões. A classe `Factory2D` que cria os objetos para o contexto de duas dimensões e a classe `Factory3D` responsável por criar objetos para o contexto de três dimensões.

Figura 13 - Diagrama de classes do pacote `factory`

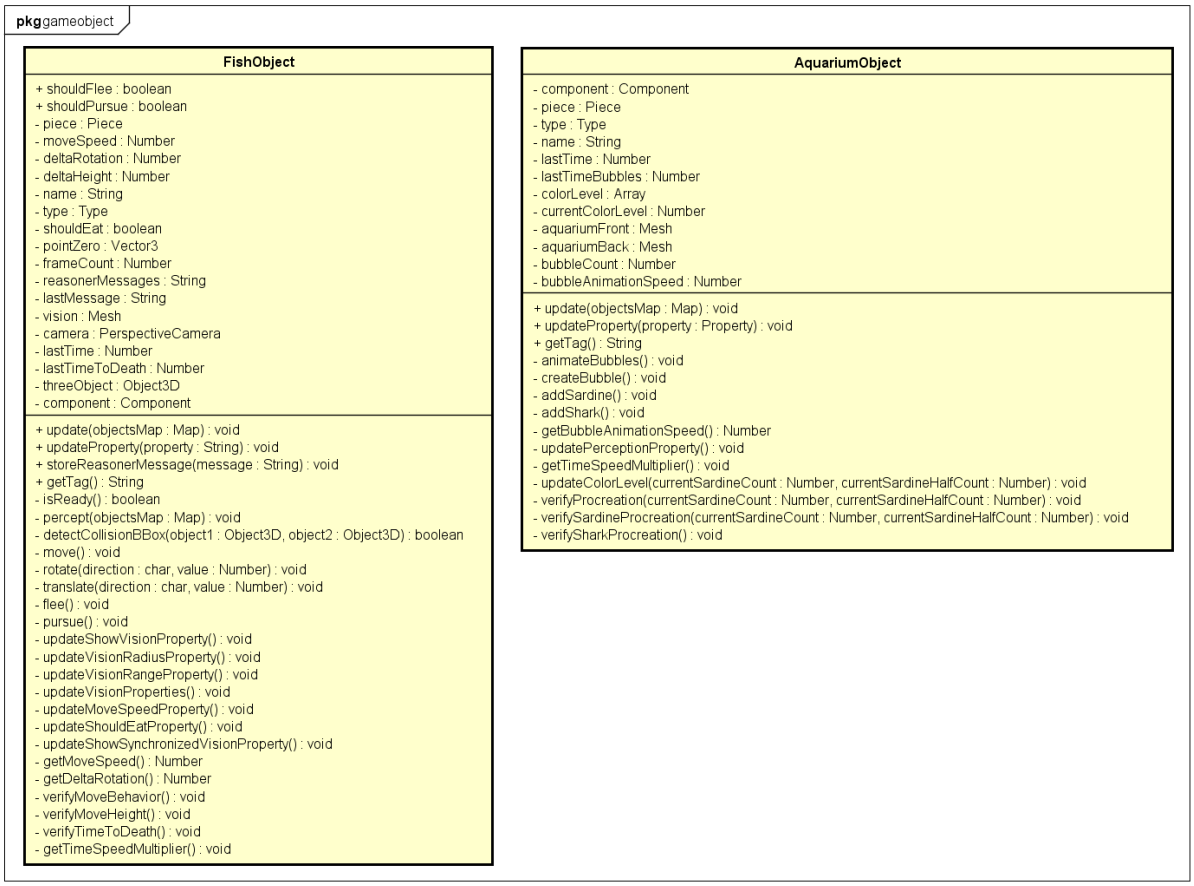



A única alteração nesse pacote ocorreu na classe `Factory3D`, na qual foram implementadas as funções `createFish`, `createAquarium` e `createPanoramicBackground`, responsáveis por criar um objeto de peixe, de aquário e de fundo panorâmico respectivamente.

### 3.2.3.5 Pacote `gameobject`

Esse pacote foi criado para a aplicação e conta com as classes `FishObject` e `AquariumObject`, ambas são uma especialização da classe `GameObject` do Motor de Jogos. Das quais a primeira representa um peixe no Aquário Virtual e a segunda o próprio aquário. Na Figura 14 pode-se observar o diagrama de classes desse pacote.

Figura 14 - Diagrama de classes do pacote `gameobject`




A classe `FishObject` possui os atributos públicos `shouldFlee` e `shouldPursue`, dos quais o primeiro indica se o peixe deve fugir e o segundo se o peixe deve perseguir. Além dos mencionados anteriormente, ela possui os atributos privados `piece`, `moveSpeed`, `deltaRotation`, `deltaHeight`, `name`, `type`, `shouldEat`, `pointZero`, `frameCount`, `reasonerMessages`, `lastMessage`, `vision`, `camera`, `lastTime`, `lastTimeToDeath`, `threeObject` e `component`.

O primeiro atributo privado armazena a peça que representa o peixe na árvore de peças e armazena algumas propriedades do mesmo. O segundo armazena a velocidade de movimento do peixe, o terceiro representa a variação de rotação em radianos que define se o peixe vai para a esquerda ou direita. O quarto indica a variação de rotação em radianos que define se o peixe vai para cima ou para baixo, o quinto armazena seu nome, o sexto seu tipo

de peixe. O sétimo indica se o peixe deve comer suas presas o oitavo armazena o ponto central do peixe, o nono guarda a quantidade de frames. O décimo armazena as mensagens recebidas do Reasoner que devem ser exibidas na área de texto quando o peixe for selecionado. O décimo primeiro guarda a última mensagem enviada pelo Reasoner, o décimo segundo representa a malha de pontos da visão do peixe, o décimo terceiro representa a câmera que exibe a visão do peixe. O décimo quarto guarda o último tempo utilizado para ter uma noção de quando executar determinadas funções, enquanto que o décimo quinto guarda o último tempo utilizado para verificar quando o peixe deve morrer de fome. O décimo sexto é responsável por armazenar o objeto tridimensional no formato da biblioteca ThreeJS e o último armazena o componente referente ao objeto.

A classe `FishObject` possui as funções públicas `update`, `updateProperty`, `storeReasonerMessage` e `getTag`. Das quais a primeira é responsável por atualizar o objeto a cada quadro, a segunda atualiza determinada propriedade do peixe, a terceira armazena determinada mensagem enviada pelo Reasoner e a última retorna a palavra-chave que identifica o tipo do objeto.

Ela também possui as funções privadas `isReady`, `percept`, `detectCollisionBBox`, `move`, `rotate`, `translate`, `flee`, `pursue`, `updateShowVisionProperty`, `updateVisionRadiusProperty`, `updateVisionRangeProperty`, `updateVisionProperties`, `updateMoveSpeedProperty`, `updateShouldEatProperty`, `updateShowSynchronizedVisionProperty`, `getMoveSpeed`, `getDeltaRotation`, `verifyMoveBehavior`, `verifyMoveHeight`, `verifyTimeToDeath` e `getTimeSpeedMultiplier`.

A primeira função privada indica se o objeto tridimensional e seus materiais e texturas já foram completamente carregados, a segunda é responsável por identificar alguma percepção. A terceira indica se houve colisão por *bounding box* entre determinados objetos, a quarta trata a movimentação do peixe, a quinta é responsável por rotacionar o peixe, a sexta por deslocar. A sétima trata o comportamento de fuga, a oitava o comportamento de perseguição. A nona atualiza a propriedade habilitar visão, a décima atualiza a propriedade raio da visão, a décima primeira atualiza a propriedade alcance da visão, a décima segunda atualiza as propriedades da visão. A décima terceira atualiza a propriedade velocidade de movimento, a décima quarta atualiza a propriedade que habilita a função comer, a décima quinta atualiza a propriedade mostrar visão sincronizada. A décima sexta calcula a velocidade de movimentação a ser utilizada, a décima sétima calcula a variação de rotação em radianos

que influencia a direção para a esquerda ou direita. A décima oitava faz a verificação do comportamento de movimentação, a décima nona faz a verificação da altura de movimentação, a vigésima verifica se o peixe ficou tempo demais sem comer. A última obtém o multiplicador de tempo em relação a velocidade do mundo, ou seja, quanto maior for o multiplicador de velocidade do mundo, menor será o tempo para ocorrer eventos como a procriação e a atualização do nível de plâncton.

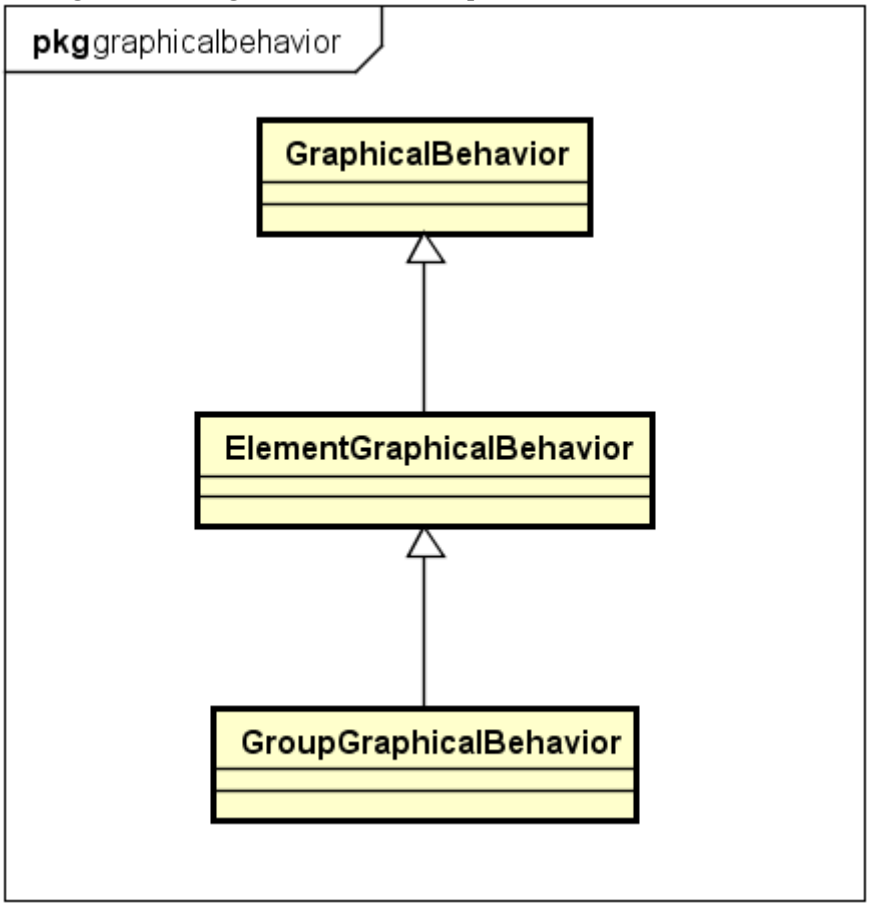
A classe `AquariumObject` possui os atributos `component`, `piece`, `type`, `name`, `lastime`, `lastTimeBubbles`, `colorLevel`, `currentColorLevel`, `aquariumFront`, `aquariumBack`, `bubbleCount` e `bubbleAnimationSpeed`. Dos quais o primeiro armazena o componente referente ao objeto, o segundo armazena a peça com as propriedades do aquário, o terceiro representa o tipo do objeto. O quarto representa o nome do objeto, o quinto guarda o último tempo utilizado como base para executar determinadas funções, o sexto guarda o último tempo utilizado como base para a criação de novas bolhas. O sétimo armazena as cores para cada nível de plâncton na água, o oitavo representa o nível atual de cor da água, o nono armazena o objeto que representa a parte de fora do aquário e o décimo armazena o objeto que representa a parte de dentro. O décimo primeiro armazena a quantidade de bolhas presente no aquário e o último representa a velocidade de movimento das bolhas.

A classe possui as funções públicas `update`, `updateProperty` e `getTag`. Das quais a primeira é responsável por atualizar o aquário a cada quadro, a segunda por atualizar determinada propriedade e a última retorna o identificador do objeto aquário. Ela também possui as funções privadas `animateBubbles`, `createBubble`, `addSardine`, `addShark`, `getBubbleAnimationSpeed`, `updatePerceptionProperty` e `getTimeSpeedMultiplier`, `updateColorLevel`, `verifyProcreation`, `verifySardineProcreation` e `verifySharkProcreation`. Dos quais o primeiro é responsável por gerar as animações das bolhas, o segundo por criar uma bolha, o terceiro por adicionar uma sardinha tanto na árvore quanto no aquário e a quarta por adicionar um tubarão. A quinta por retornar a velocidade de animação da bolha, a sexta por atualizar a propriedade `Percepção`, a sétima por retornar o multiplicador de tempo em relação a velocidade do mundo. A oitava por atualizar o nível de cor da água, a nona por verificar a procriação dos peixes, a décima por verificar se as sardinhas devem procriar e a última por verificar se os tubarões devem procriar.

#### 3.2.3.6 Pacote `graphicalbehavior`

Conforme Koehler (2015, p. 42), “as classes do pacote `graphicalbehaviour` são encarregadas de tratar os eventos gráficos de forma a adicionar, recarregar e remover os componentes da tela.” O diagrama de classes desse pacote pode ser observado na Figura 15.

Figura 15 - Diagrama de classes do pacote `graphicalbehavior`

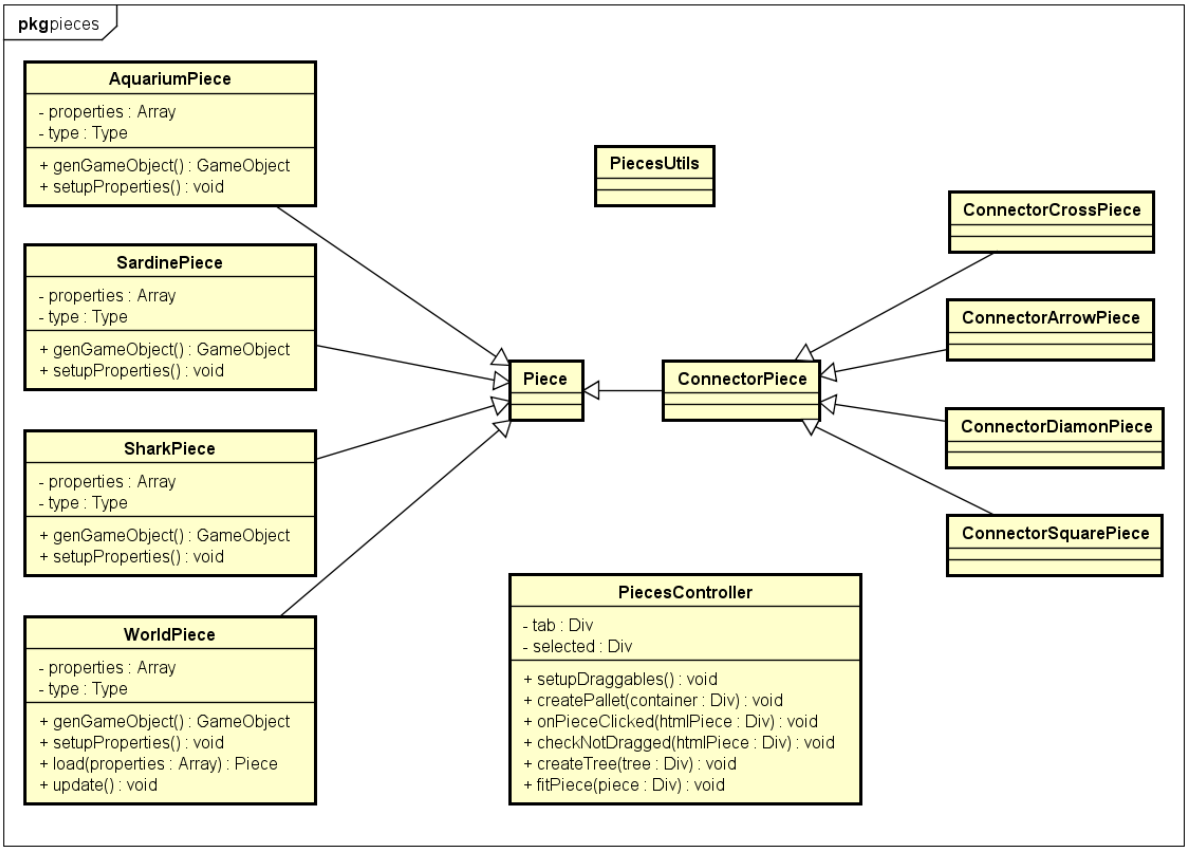



O pacote original se chamava `graphicalbehaviour` mas foi renomeado para `graphicalbehavior` e continha as classes `CameraGraphicalBehaviour`, `LightGraphicalBehaviour`, `TransformationGraphicalBehaviour`, `GraphicalBehaviour`, `ElementGraphicalBehaviour` e `GroupGraphicalBehaviour`. As três primeiras foram removidas por não terem utilidade enquanto que as outras três somente tiveram seu nome alterado. Seus nomes foram alterados para padronizar o idioma utilizado no código, pois “*behaviour*” é geralmente utilizado no inglês britânico enquanto que “*behavior*” no inglês americano.

#### 3.2.3.7 Pacote `pieces`

O pacote `pieces` é responsável por armazenar as classes que representam as peças do Aquário Virtual e também seus controladores e utilitários. Pode-se observar o diagrama de classes desse pacote na Figura 16.

Figura 16 - Diagrama de classes do pacote `pieces`

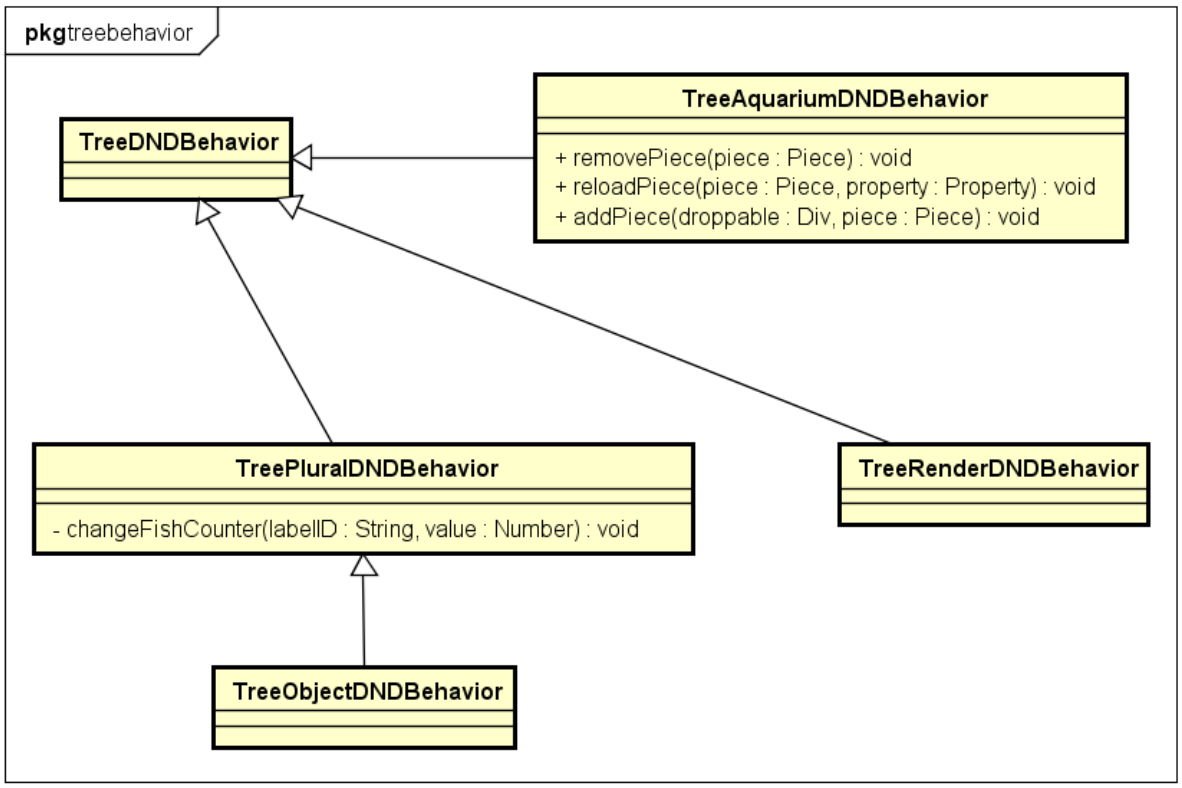



As classes `Piece`, `PiecesUtils`, `ConnectorPiece`, `ConnectorCrossPiece`, `ConnectorArrowPiece`, `ConnectorDiamondPiece` e `ConnectorSquarePiece` não foram alteradas, enquanto que as classes `CameraPiece`, `CubePiece`, `GraphicObjectPiece`, `LightPiece`, `PolygonPiece`, `RendererPiece`, `RotatePiece`, `ScalePiece`, `SplinePiece`, `TransformPiece` e `TranslatePiece` foram removidas e as classes `AquariumPiece`, `SardinePiece`, `SharkPiece` e `WorldPiece` foram criadas. Todas as classes criadas possuem o atributo `properties` que armazena as propriedades de cada peça e o atributo `type` que indica e guarda informações do tipo da peça. Todas elas também possuem as funções `genGameObject` e `setupProperties`, das quais a primeira é responsável por gerar o objeto que representa a peça e a segunda por popular as propriedades da peça. A classe `WorldPiece` possui duas funções a mais, a `load` que serve para carregar as propriedades do mundo e a `update` para atualizar essas propriedades.

### 3.2.3.8 Pacote `treebehavior`

O pacote `treebehavior` armazena classes responsáveis por controlar o comportamento de adicionar, realocar e remover determinada peça, disparando os eventos necessários a serem refletidos no contexto gráfico (KOEHLER, 2015, p. 41). O diagrama de classes desse pacote pode ser observado na Figura 17.

Figura 17 - Diagrama de classes do pacote `treebehavior`



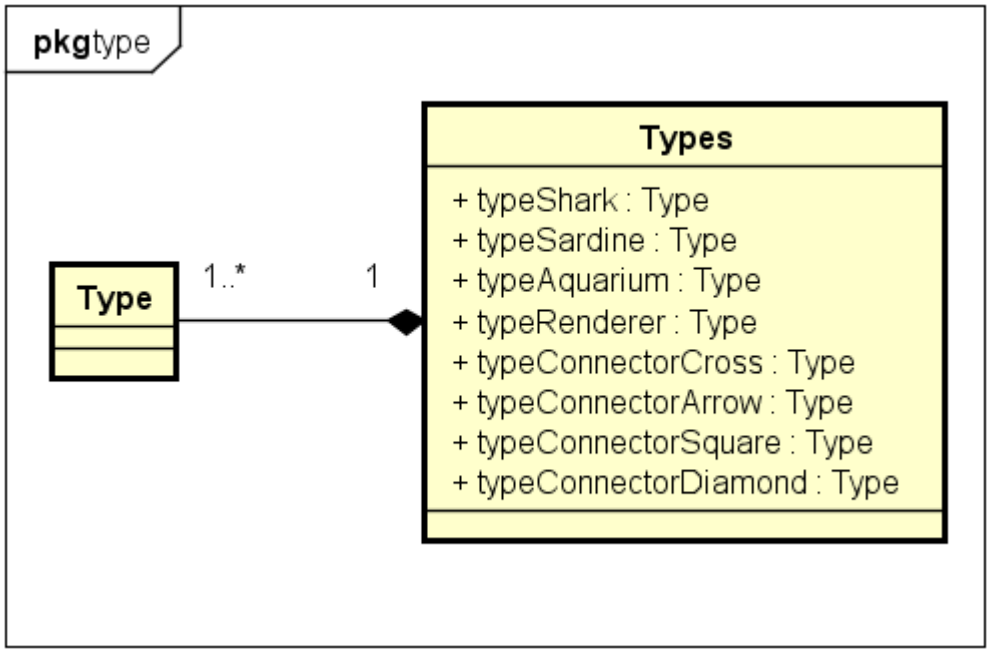

O pacote original `treebehaviour` foi renomeado para `treebehavior`, além das classes `TreeDNDBehaviour`, `TreePluralDNDBehaviour`, `TreeObjectDNDBehaviour` e `TreeRenderDNDBehaviour` que foram renomeadas para `TreeDNDBehavior`, `TreePluralDNDBehavior`, `TreeObjectDNDBehavior` e `TreeRenderDNDBehavior` respectivamente. A classe `TreePluralDNDBehavior` recebeu a função privada `changeFishCounter`, responsável por alterar o contador de peixes mantido na área de propriedades da peça `Aquário`.

A classe `TreeAquariumDNDBehavior` foi criada para implementar os comportamentos da peça `Aquário`, ela implementa as funções `removePiece`, `reloadPiece` e `addPiece`, das quais a primeira é responsável pelo comportamento de remover a peça, a segunda por recarregar e a terceira por adicionar. As outras peças do Aquário Virtual reutilizaram as classes existentes desse pacote.

### 3.2.3.9 Pacote `type`

O pacote `type` armazena classes responsáveis por definir os tipos de peças. O diagrama de classes desse pacote pode ser visto na Figura 18.

Figura 18 - Diagrama de classes do pacote `type`

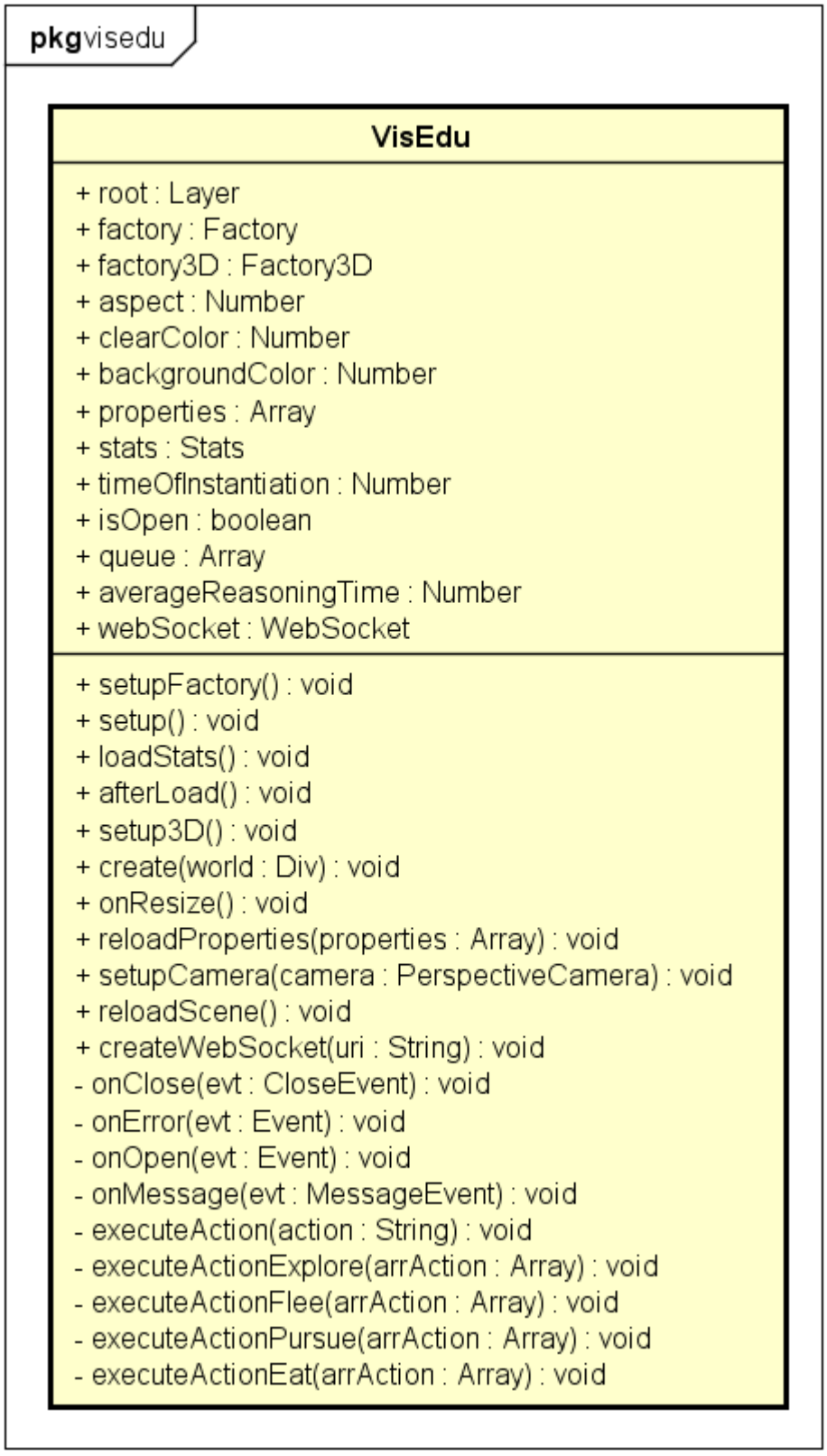



A estrutura original das classes desse pacote não foi alterada, porém foram retirados os tipos que não dizem respeito ao Aquário Virtual e inseridos novos tipos como `typeShark`, `typeSardine` e `typeAquarium`, representando o tipo da peça `Tubarão`, `Sardinha` e `Aquário` respectivamente.

### 3.2.3.10 Pacote `visedu`

O pacote `visedu` contém a classe principal da aplicação, que gerencia todo o resto. O diagrama de classes desse pacote pode ser observado na Figura 19.

Figura 19 - Diagrama de classes do pacote `visedu`




A classe `VisEdu` possui os atributos `root`, `factory`, `factory3D`, `aspect`, `clearColor`, `backgroundColor`, `properties`, `stats`, `timeOfInstantiation`, `isOpen`, `queue`, `averageReasoningTime` e `webSocket`. Dos quais o primeiro armazena a camada principal da

aplicação, o segundo a fábrica de objetos selecionada, o terceiro a instância da fábrica de objetos tridimensionais. O quarto representa a proporção da câmera principal, o quinto a cor de fundo da câmera quando não possuir nada instanciado na cena atual. O sexto a cor de fundo da aplicação, o sétimo representa as propriedades do mundo do Aquário Virtual, o oitavo representa o componente da biblioteca ThreeJS, que traz estatísticas sobre a performance da aplicação. O nono guarda o tempo de instanciação do Reasoner, o décimo indica se o `websocket` está aberto, o décimo primeiro armazena as datas das mensagens enviadas para o Reasoner. O décimo segundo armazena o tempo médio de resposta do Reasoner e o último guarda a instância do `websocket`.

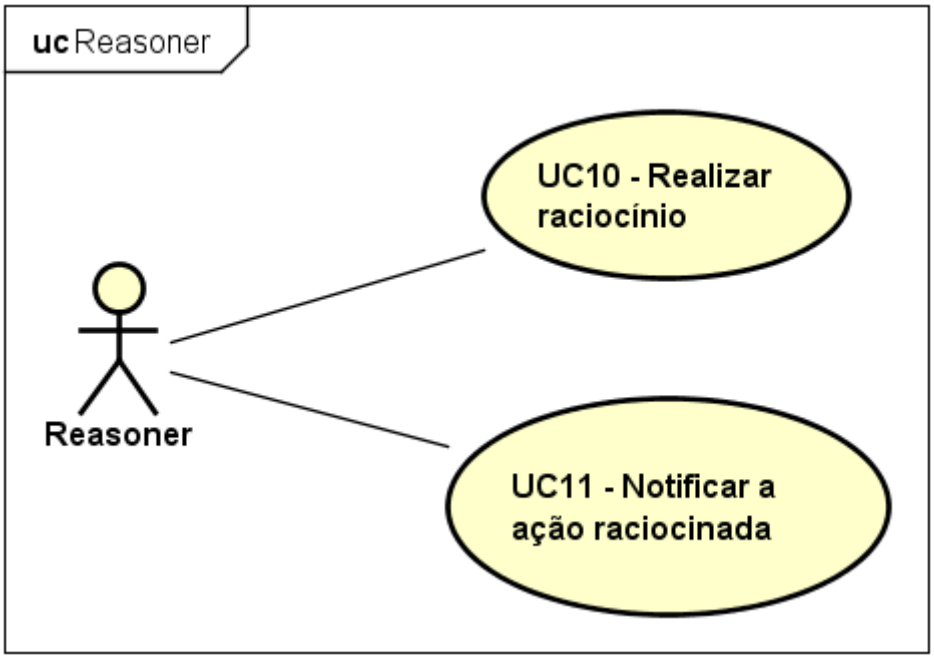
Essa classe possui as funções públicas `setupFactory`, `setup`, `loadStats`, `afterLoad`, `setup3D`, `create`, `onResize`, `reloadProperties`, `setupCamera`, `reloadScene` e `createWebSocket`. A primeira é responsável por inicializar a fábrica de objetos, a segunda por inicializar objetos básicos para o funcionamento da aplicação, a terceira por carregar o componente de estatísticas de performance da biblioteca ThreeJS. A quarta por carregar objetos gráficos que só podem ser criados após a instanciação dos componentes básicos para o mesmo. A quinta por inicializar as funções tridimensionais de controle e percepção da câmera principal, a sexta por iniciar a criação do ambiente virtual utilizado para esta aplicação. A sétima por ajustar a aplicação ao ter a tela redimensionada, a oitava por recarregar as propriedades do mundo, a nona por atualizar a câmera principal, a décima por recarregar a aplicação e a última por criar o `websocket` utilizado para se comunicar com o Reasoner.

As funções privadas dessa classe estão relacionadas com o `websocket` e o Reasoner, são elas, `onClose`, `onError`, `onOpen`, `onMessage`, `executeAction`, `executeActionExplore`, `executeActionFlee`, `executeActionPursue` e `executeActionEat`. Das quais a primeira é chamada quando o `websocket` é fechado e é responsável por atualizar a *flag* `isOpen`. A segunda é chamada quando o `websocket` informa um erro e é responsável por exibir esse erro. A terceira é chamada quando o `websocket` é aberto e é responsável por definir o tempo de instanciação do Reasoner. A quarta é chamada quando o Reasoner envia uma mensagem, se responsabilizando por ajustar o tempo médio de resposta e encaminhar a mensagem para sua respectiva ação ser executada dentro do Aquário Virtual. A quinta tem a finalidade de gerenciar a execução das ações, a sexta de executar a ação explorar, a sétima de executar a ação fugir, a oitava de executar a ação perseguir e a última de executar a ação comer.

### 3.2.4 Diagrama de caso de uso do Reasoner

Conforme pode ser visto na Figura 20, foram criados dois casos de uso para o ator `Reasoner`. Os casos de uso foram criados com base nos requisitos funcionais do Reasoner.

Figura 20 - Diagrama de caso de uso do Reasoner




Em comparação com a primeira versão do Reasoner, desenvolvido por Feltrin (2014), foi retirado o caso de uso "Realizar download da mente". Originalmente ele foi desenvolvido para ser utilizado em conjunto com um editor de jogos, gerando a necessidade de manipular as mentes. Nessa aplicação as mentes não são controladas pelo usuário e ficam armazenadas no servidor de aplicação. No Quadro 10 é apresentado o detalhamento do primeiro caso de uso operado pelo ator `Reasoner`.

Quadro 10 - Caso de uso `UC10: Realizar raciocínio`

| UC10 – Realizar raciocínio | |
|---|---|
| Descrição | Permite que o Reasoner realize o processamento do raciocínio. |
| Cenário Principal | 1. A simulação envia a mensagem com as percepções para o servidor de aplicação.<br>2. O servidor de aplicação transmite a mensagem para o Reasoner.<br>3. O Reasoner verifica a mente correspondente e a interpreta com base nas percepções recebidas.<br>4. O Reasoner invoca o UC11 para retornar a ação determinada. |
| Pré-Condição | Receber percepções da simulação. |
| Pós-Condição | A simulação recebe a ação raciocinada. |

No Quadro 11 é apresentado o detalhamento do segundo caso de uso operado pelo ator `Reasoner`.

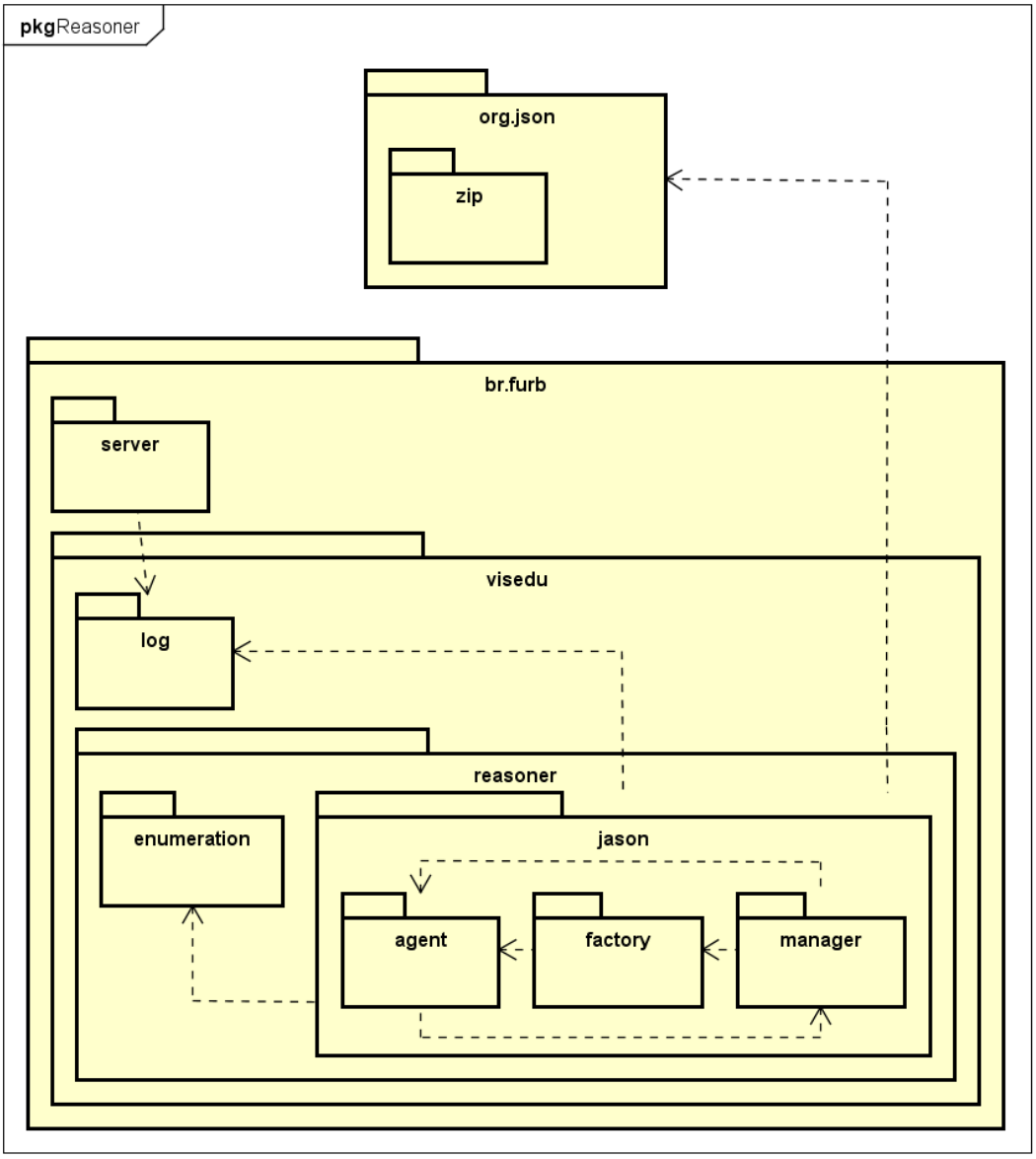
Quadro 11 - Caso de uso `UC11: Notificar a ação raciocinada`

| UC11 – Notificar a ação raciocinada | |
|---|---|
| Descrição | Reasoner notifica a simulação com a ação determinada. |
| Cenário Principal | 1. O Reasoner invoca o UC10.<br>2. O Reasoner transmite a ação determinada para o servidor de aplicação.<br>3. O servidor de aplicação transmite a ação raciocinada para a simulação. |
| Pré-Condição | Receber percepções da simulação. |
| Pós-Condição | A simulação é notificada com a ação raciocinada. |

### 3.2.5 Diagrama de pacotes do Reasoner

Nessa seção são descritos os pacotes e as classes criadas no desenvolvimento do Reasoner. Na Figura 21 pode-se observar o diagrama de pacotes do Reasoner.

Figura 21 - Diagrama de pacotes do Reasoner

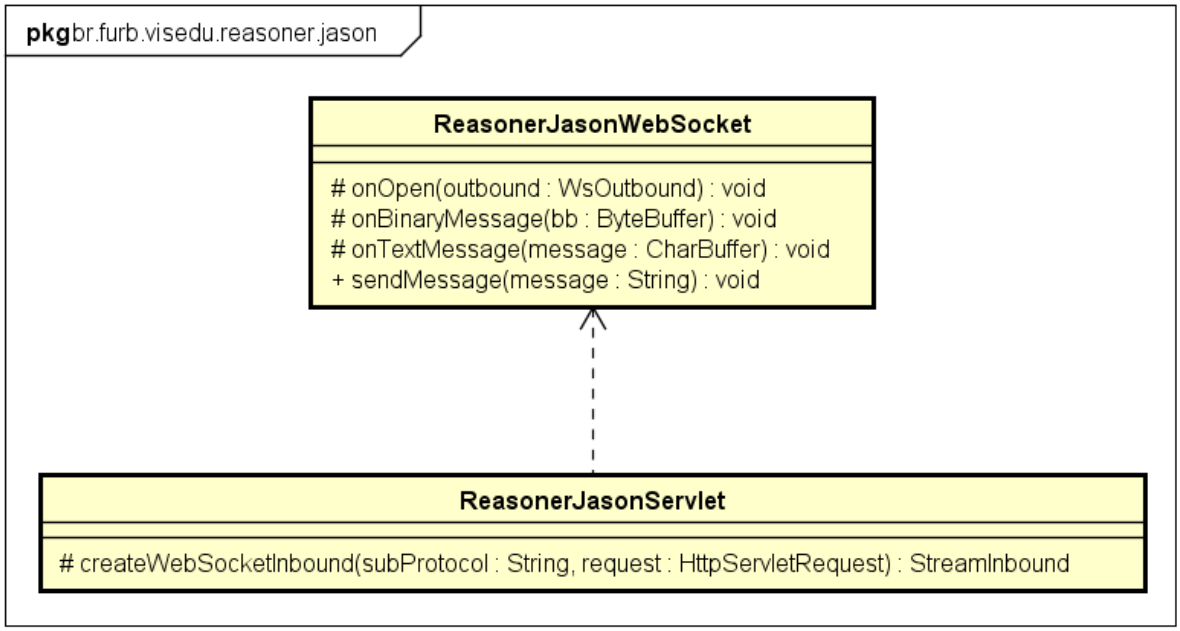


O Reasoner é composto por dois pacotes principais, `br.furb` e `org.json`, dos quais o primeiro comporta as classes e sub-pacotes da aplicação em si e o segundo trata da implementação de um tratador de mensagens JSON em Java. Nas seções a seguir serão apresentados o detalhamento dos sub-pacotes pertencentes ao pacote `br.furb` que foram alterados.

#### 3.2.5.1 Pacote `jason`

O pacote `jason` possui as classes responsáveis por gerenciar o interpretador Jason (FELTRIN, 2014, p. 38), conforme pode ser visto na Figura 22.

Figura 22 - Diagrama de classes do pacote `jason`

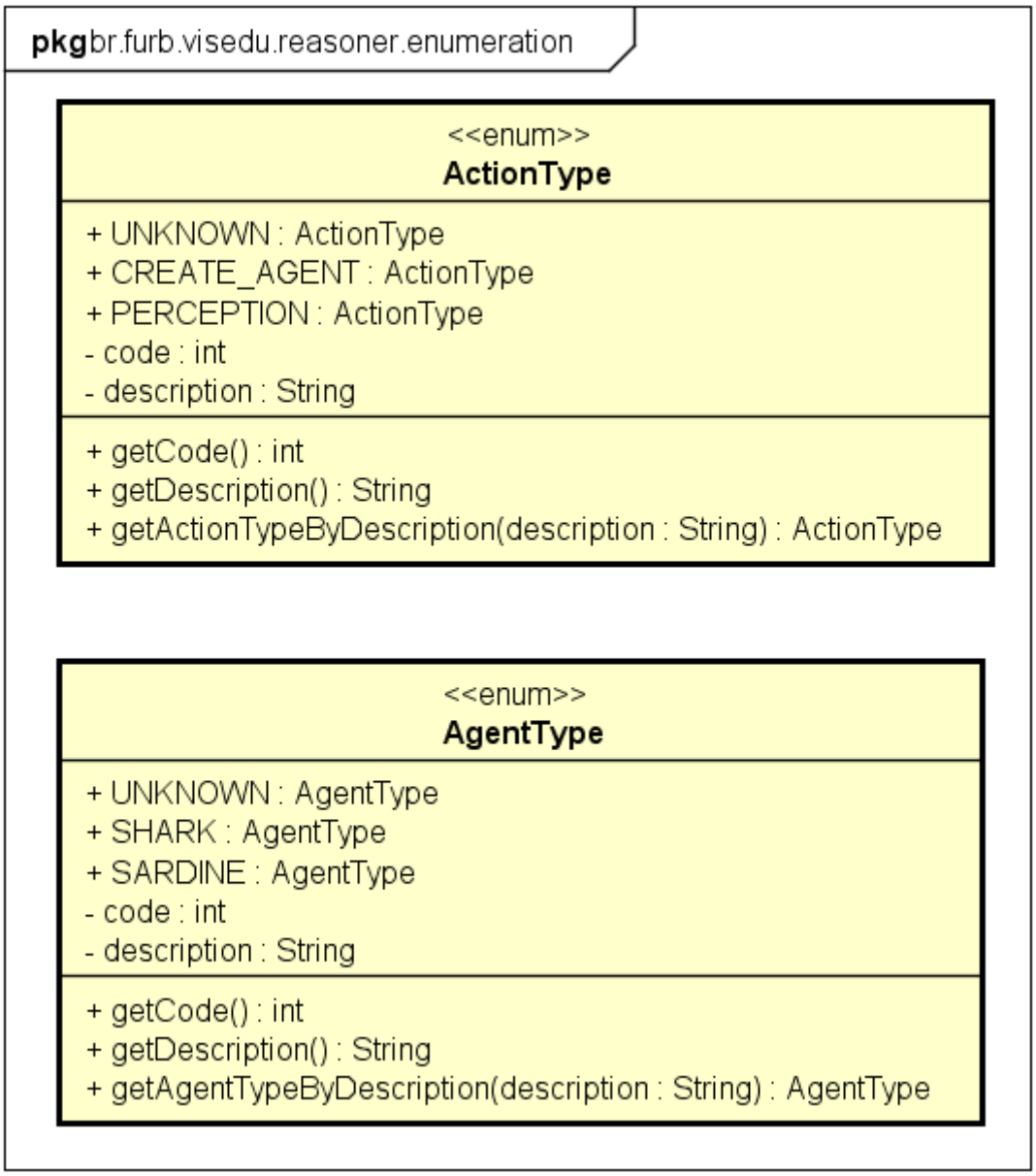



A classe `ReasonerJasonServlet` estende a classe `WebSocketServlet` e é responsável por estabelecer conexão com o Aquário Virtual via *WebSocket*, enquanto que a classe `ReasonerJasonWebSocket` estende a classe `MessageInbound` e é responsável por gerenciar o envio e recebimento de mensagens.

#### 3.2.5.2 Pacote `enumeration`

O pacote `enumeration` é responsável por armazenar as enumerações do Reasoner. O diagrama de classes desse pacote pode ser observado na Figura 23.

Figura 23 - Diagrama de classes do pacote `enumeration`

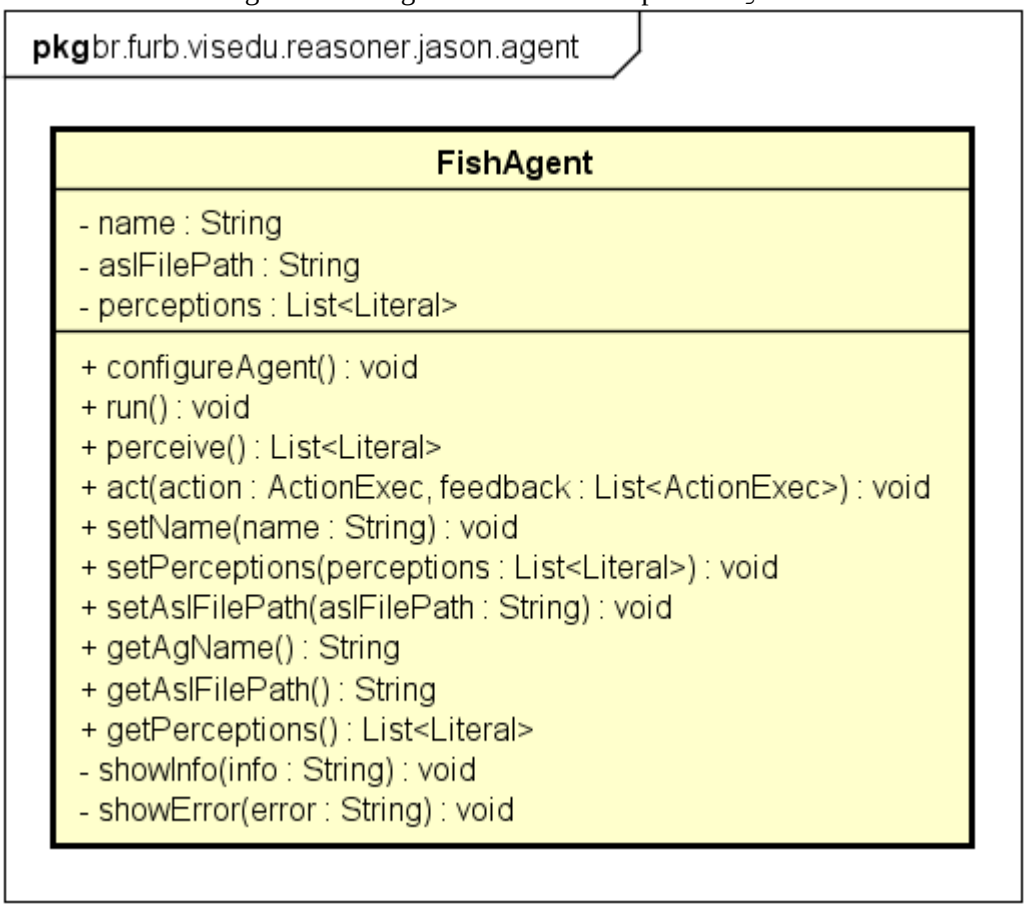



O pacote `enumeration` foi criado e conta com as classes `ActionType` e `AgentType`, das quais a primeira é responsável por enumerar os tipos de ação que o Reasoner reconhece. São eles, `CREATE_AGENT` para indicar a ação de criar um objeto de agente, `PERCEPTION` para indicar a ação de percepção e `UNKNOWN` para indicar uma ação desconhecida. A segunda é responsável por enumerar os tipos de agente suportados pelo Reasoner. São eles, `SHARK` para indicar um agente do tipo tubarão, `SARDINE` do tipo sardinha e `UNKNOWN` para indicar um tipo de agente desconhecido.

#### 3.2.5.3 Pacote `agent`

O pacote `agent` foi criado para armazenar as classes que representam a implementação de arquiteturas de agentes, conforme pode ser visto na Figura 24.

Figura 24 - Diagrama de classes do pacote `agent`




Esse pacote possui somente a classe `FishAgent`, que estende da classe `AgArch` do interpretador Jason. Ela funciona como intermediário entre o Aquário Virtual e o interpretador Jason, tratando as informações de forma genérica para qualquer tipo de peixe, já que o que vai diferenciar o comportamento entre os diferentes tipos é a mente carregada na instância dessa classe. Essa classe possui os atributos `name`, `aslFilePath` e `perceptions`, dos quais o primeiro armazena o nome do agente, o segundo o caminho do arquivo correspondente à mente do agente e o último guarda sua lista de percepções.

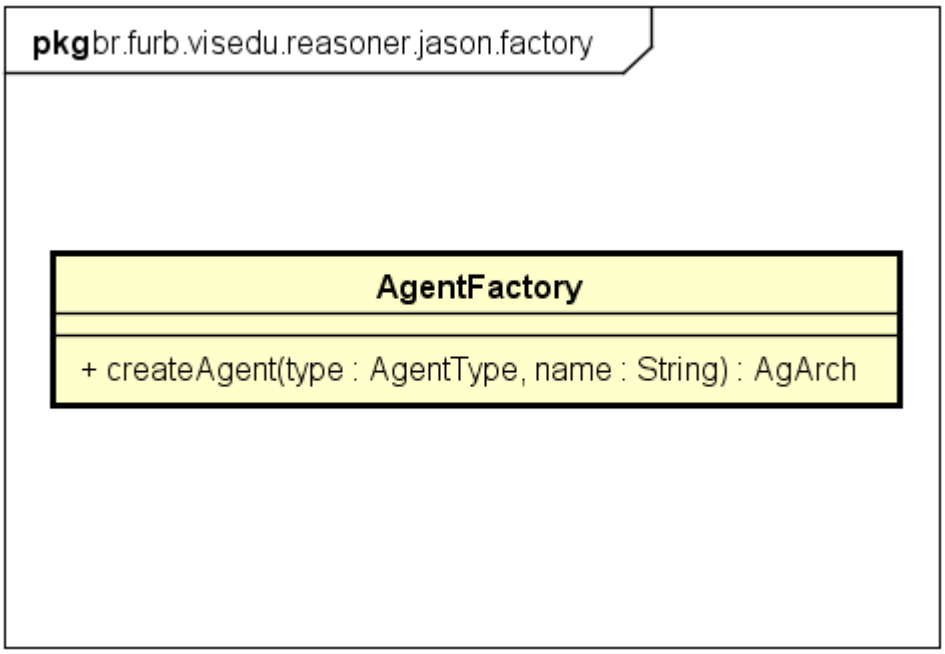
A classe possui os métodos públicos `configureAgent`, `run`, `perceive`, `act`, `setName`, `setPerceptions`, `setAslFilePath`, `getAgName`, `getAslFilePath` e `getPerceptions`. O primeiro é responsável por configurar o agente, o segundo por executar um ciclo de raciocínio e o terceiro por carregar as percepções na mente. O quarto por carregar a resposta do raciocínio, o quinto por definir o nome do agente, o sexto por definir as percepções atuais, o sétimo por definir o caminho do arquivo correspondente à mente do agente. O oitavo por obter seu nome, o nono por obter o caminho do arquivo correspondente à sua mente e o último por obter as percepções atuais. Ela também possui os métodos privados `showInfo` e

`showError`, dos quais o primeiro é responsável por exibir informações e o segundo por exibir erros.

#### 3.2.5.4 Pacote `factory`

O pacote `factory` foi criado para armazenar classes responsáveis por fabricar objetos, sendo que no momento o pacote possui apenas a classe `AgentFactory`, conforme pode ser visto na Figura 25.

Figura 25 - Diagrama de classes do pacote `factory`

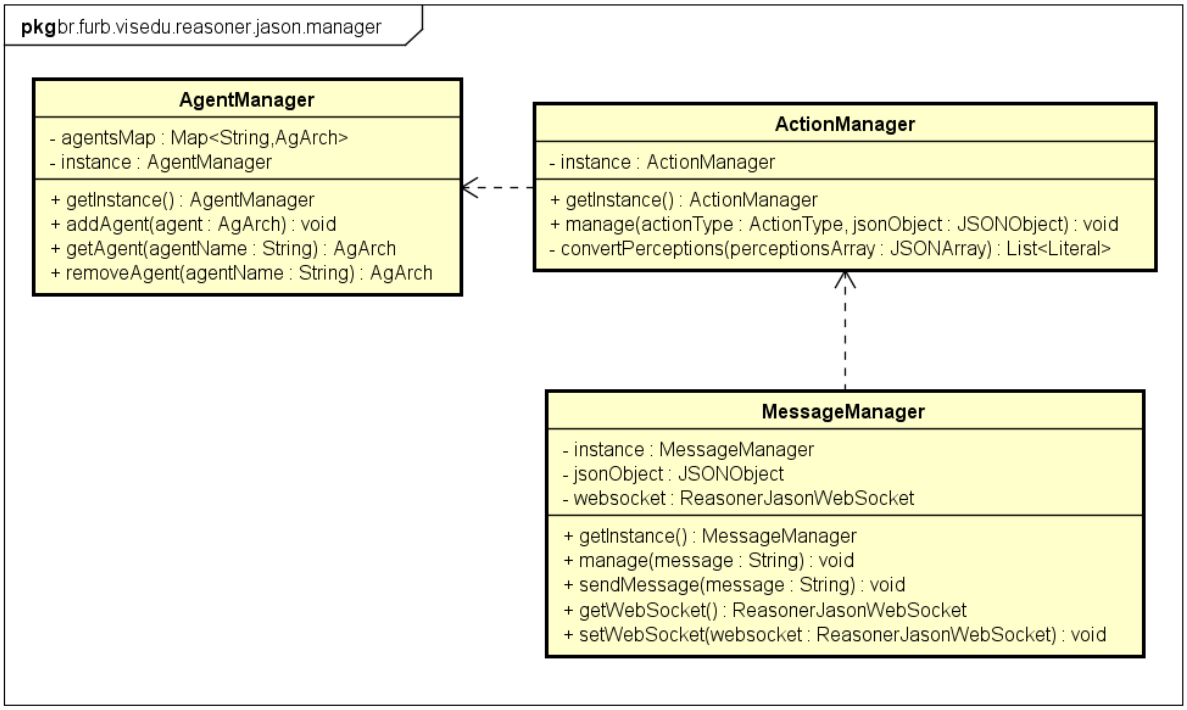



A classe `AgentFactory` é implementada com base no padrão Factory e é responsável por gerenciar a criação de agentes através do método `createAgent` que recebe o tipo e nome do agente a ser criado.

#### 3.2.5.5 Pacote `manager`

O pacote `manager` foi criado para armazenar classes gerenciadoras, conforme pode ser visto na Figura 26.

Figura 26 - Diagrama de classes do pacote `manager`




O pacote `manager` possui três classes, `AgentManager` que é responsável por gerenciar os agentes do Reasoner, `ActionManager` que se responsabiliza por gerenciar as ações que o Aquário Virtual envia e a classe `MessageManager` que gerencia o envio e recebimento de mensagens, ambas implementam o padrão Singleton. A primeira classe possui o atributo `agentsMap`, um mapa de agentes que traz facilidade e rapidez para busca-los através de seus nomes. Também possui o atributo `instance` que armazena a instância única dessa classe. Seus métodos são, `getInstance`, `addAgent`, `getAgent` e `removeAgent`. O primeiro obtém a instância única dessa classe, o segundo adiciona um agente ao mapa através de seu nome, o terceiro obtém um agente do mapa também através de seu nome e o último remove determinado agente do mapa.

A segunda classe possui apenas o atributo `instance`, que armazena a instância única da classe e os métodos `getInstance`, `manage` e `convertPerceptions`. O primeiro obtém a instância única da classe, o segundo é responsável por gerenciar as ações do Reasoner, informando o tipo da ação e a mensagem enviada em formato JSON. Desse jeito o método saberá o que fazer com essas informações. O último recebe uma matriz de percepções e converte para uma lista de literais compatíveis com o interpretador Jason.

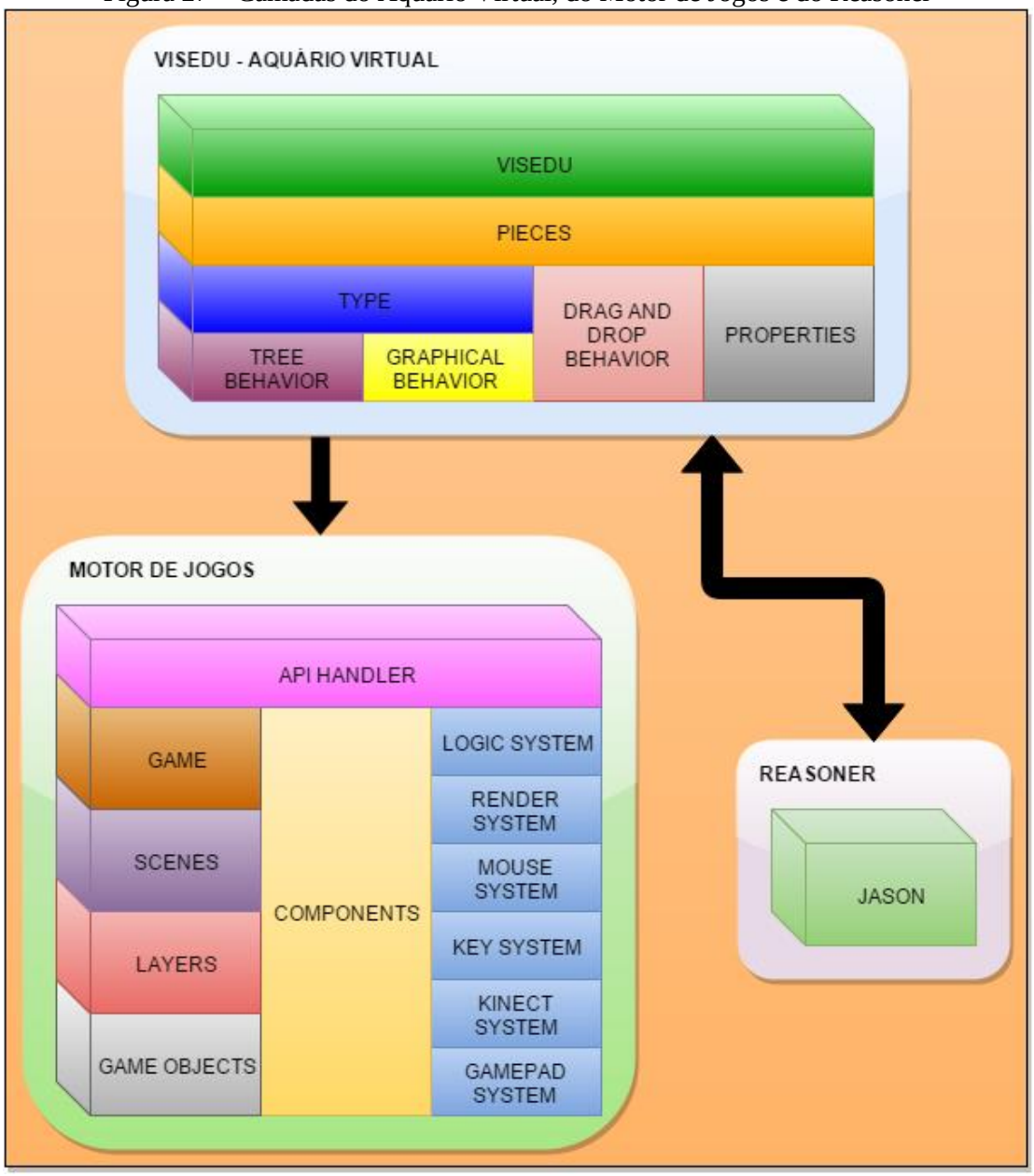
A terceira classe possui os atributos `instance`, `jsonObject` e `websocket`, dos quais o primeiro armazena a instância única da classe, o segundo guarda a referência para um objeto JSON que receberá novas instâncias a cada troca de mensagens e o terceiro guarda a referência do *WebsSocket*, necessário para enviar mensagens de volta para o Aquário Virtual.

A classe também possui os métodos `getInstance`, `manage`, `sendMessage`, `getWebSocket` e `setWebSocket`. O primeiro é responsável por obter a instância única da classe, o segundo por gerenciar as mensagens recebidas, o terceiro por enviar determinada mensagem para o Aquário Virtual através do *WebSocket*, o quarto por obter a instância do *WebSocket* e o último por definir o *WebSocket* utilizado.

### 3.2.6 Diagrama de arquitetura

A Figura 27 apresenta o diagrama das camadas da arquitetura do Aquário Virtual, do Motor de Jogos e do Reasoner.

Figura 27 – Camadas do Aquário Virtual, do Motor de Jogos e do Reasoner



A especificação da arquitetura do Motor de Jogos em relação ao trabalho de Koehler (2015) foi mantida, porém em relação ao trabalho de Feltrin (2014) retirou-se o componente `Perception System` e levou-se a interface de comunicação com o Reasoner para o Aquário Virtual. A arquitetura do Visualizador de Material Educacional foi mantida, além do desacoplamento proveniente das implementações anteriores baseadas em uma arquitetura orientada a componentes. Foi adicionado à arquitetura o componente Reasoner, responsável por adaptar as informações do ambiente externo para interpretação do modelo mental Jason (FELTRIN, 2014, p. 42).

## 3.3 IMPLEMENTAÇÃO

Nesta seção são apresentadas as técnicas e ferramentas utilizadas na implementação do Reasoner e do Aquário Virtual, além de abordar as alterações feitas no Motor de Jogos, apresentando as principais classes e funções desenvolvidas.

### 3.3.1 Técnicas e ferramentas utilizadas

Para o desenvolvimento do Reasoner foi utilizada a linguagem de programação Java na versão 6 e como ambiente de desenvolvimento foi utilizado o Eclipse IDE for Java EE Developers na versão 4.5.0 (Mars) utilizando o *plug-in* Jasonide. Para trabalhar com o modelo BDI foi utilizado o interpretador Jason na versão 1.4.2 e como servidor de aplicação foi utilizado o Apache Tomcat versão 7. Para a codificação das mentes utilizadas pelo interpretador Jason, foi utilizada uma versão estendida da linguagem orientada a agentes AgentSpeak, implementada pelo próprio Jason.

Para as alterações no Motor de Jogos foi necessário a utilização da linguagem Javascript e do ambiente de desenvolvimento Eclipse IDE for Java EE Developers na versão 4.5.0. Para a implementação do Aquário Virtual, foram utilizadas as linguagens HTML5, Javascript e CSS, além das bibliotecas JQuery para abstrair a manipulação de elementos HTML e ThreeJS na versão 72 para abstrair o uso da *Web Graphics Library* (WebGL). Também teve como ambiente de desenvolvimento o Eclipse IDE for Java EE Developers na versão 4.5.0.

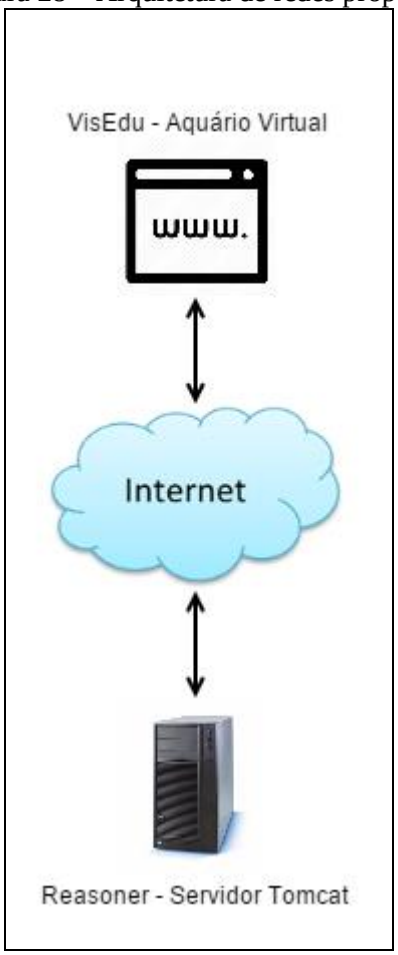
Para a realização da implementação do trabalho foram utilizados os navegadores Chrome versão 46.0.2490.80, Edge versão 20.10240.16384.0, Firefox versão 41.0.2, Opera versão 33.0.1990.43 e Internet Explorer versão 11.0.10240.16431 em um notebook ASUS S400C com sistema operacional Windows 10 64 bits, processador Intel Core i5 3317U 1.70 *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 para o envio dos mesmos foi utilizado o próprio ambiente de desenvolvimento.

### 3.3.2 Arquitetura de redes proposta

Nesta seção será descrita a arquitetura de redes proposta para implementar o trabalho. O Reasoner foi criado sob a arquitetura cliente/servidor, conforme pode ser observado na Figura 28.

Figura 28 – Arquitetura de redes proposta




O servidor Tomcat é responsável por disponibilizar o Reasoner, possibilitando a realização de raciocínio para o Aquário Virtual, que funciona como cliente e executa direto no navegador. Essa arquitetura se mostra interessante, pois o Reasoner pode ser reutilizado em outras aplicações na internet, usando apenas um servidor. O principal ponto negativo é que todo o raciocínio está centralizado nesse servidor, caso ele apresente algum problema, todas as aplicações ficam sem seus serviços.

### 3.3.3 Motor de Jogos

Nesta seção serão descritas as alterações feitas no Motor de Jogos, demonstrando as principais rotinas e classes alteradas.

Além da inclusão das classes `DDSLoader`, `MTLLoader` e `OBJMTLLoader`, disponibilizadas pelos próprios desenvolvedores da biblioteca ThreeJS, que auxiliam no processo de carregamento de arquivos contendo informações de objetos tridimensionais, seus materiais e texturas. Foi implementada a função `update` na classe `GameObject`, conforme pode ser visto no Quadro 12.

Quadro 12 - Implementação da função `update` na classe `GameObject`

```
1  GameObject.prototype.update = function(){return null;}
```

A função `update` foi implementada de forma abstrata, deixando seu comportamento específico em cada subclasse. Seu objetivo é permitir que cada objeto do Motor de Jogos seja responsável pela atualização de seus atributos a cada quadro, permitindo animações em tempo real.

### 3.3.4 Reasoner

Nesta seção será descrita a implementação do Reasoner, demonstrando seus principais métodos e a codificação dos agentes. O objetivo do Reasoner é ser um motor de raciocínio, centralizando o "cérebro" de diversos personagens que terão sua inteligência carregada através de modelos mentais. Para a criação do Reasoner optou-se por utilizar o interpretador Jason, tanto por ser uma ferramenta *open source* quanto por proporcionar a integração dos agentes de seu Sistema Multiagente (SMA) a ambientes externos através da linguagem de programação Java (FELTRIN, 2014, p. 44).

O Quadro 13, apresenta a implementação da classe `ReasonerJasonServlet`, que serve como um mini servidor para o *WebSocket* na aplicação servidora.

Quadro 13 - 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              return new ReasonerJasonWebSocket();
9        }
10
11 }
```

Na linha 1 é definida a anotação `WebServlet` contendo a chave para localização do *WebSocket*, já na linha 8 pode-se ver que o método `createWebSocketInbound` retorna um

*WebSocket* customizado. A implementação do *WebSocket* customizado para o Reasoner pode ser observado no Quadro 14.

Quadro 14 - Implementação da classe `ReasonerJasonWebSocket`

```
1  public class ReasonerJasonWebSocket extends MessageInbound {
2
3        @Override
4        protected void onOpen(WsOutbound outbound) {
5              Log.info("onOpen: websocket is open");
6              MessageManager.getInstance().setWebSocket(this);
7        }
8
9        @Override
10       protected void onBinaryMessage(ByteBuffer bb) throws IOException
11 {
12             throw new IOException("Method is not implemented!");
13       }
14
15       @Override
16       protected void onTextMessage(CharBuffer message) throws
17 IOException {
18             Log.info("onTextMessage: " + message);
19             MessageManager.getInstance().manage(message.toString());
20       }
21
22       public void sendMessage(String message) {
23             Log.info("sendMessage: " + message);
24
25             try {
26
27       getWsOutbound().writeTextMessage(CharBuffer.wrap(message));
28             } catch (IOException e) {
29                   e.printStackTrace();
30             }
31       }
32
33 }
```

O método `onOpen` é chamado pelo *WebSocket* quando ele é aberto, quando isso ocorre é armazenado sua instância na classe `MessageManager`, conforme pode ser visto na linha 6. O método `onBinaryMessage` é chamado quando o *WebSocket* recebe uma mensagem binária, porém não há necessidade de usá-lo. O método `onTextMessage` é chamado quando uma mensagem de texto é recebida, a mensagem é obtida através do parâmetro `message` e é encaminhada para o gerenciador de mensagens tratar, conforme pode ser visto na linha 16. O último método da classe é o `sendMessage`, responsável por enviar determinada mensagem em forma de texto para a outra ponta do *WebSocket*, conforme pode ser observado na linha 27.

Para representar um agente de peixe no interpretador Jason, criou-se a classe `FishAgent` que estende a classe `AgArch` do interpretador e que representa a arquitetura que um agente implementado em Java deve possuir. No Quadro 15 observa-se o método da classe `FishAgent` responsável por configurar o agente.

Quadro 15 - Implementação do método `configureAgent` da classe `FishAgent`

```
1       public void configureAgent() {
2             try {
3                   Agent jasonAgent = new Agent();
4                   new TransitionSystem(jasonAgent, null, null, this);
5                   jasonAgent.initAg(getAslFilePath());
6                   showInfo(String.format("Agent \"%s\" using static mind
7 @ %s", getAgName(), getAslFilePath()));
8             } catch (JasonException e) {
9                   e.printStackTrace();
10                  showError("Error initializing agent");
11            }
12      }
```

Na linha 3 pode ser visto a criação de um objeto `Agent` que representa um agente no interpretador. Na linha 5 observa-se o objeto `Agent` criado chamando o método `initAg` responsável por carregar o arquivo ASL que contém a mente codificada em AgentSpeak. No Quadro 16 é demonstrado o método sobrescrito da classe `AgArch`, chamado quando uma ação é definida pelo interpretador.

Quadro 16 - Implementação do método `act` da classe `FishAgent`

```
1       @Override
2       public void act(ActionExec action, List<ActionExec> feedback) {
3             showInfo("Agent " + getAgName() + ": doing: " +
4 action.getActionTerm());
5             MessageManager.getInstance().sendMessage(
6 action.getActionTerm().toString() );
7             action.setResult(true);
8             feedback.add(action);
9       }
```

Nas linhas 5 e 6 é obtida a instância única do gerenciador de mensagens e chama-se o método `sendMessage`, responsável por enviar uma mensagem através do *WebSocket*. A mensagem enviada por parâmetro a esse método é o termo da ação determinada pelo interpretador, que é obtida através do método `getActionTerm`. No Quadro 17 pode ser visto o método responsável por chamar a execução de um ciclo de raciocínio.

Quadro 17 - Implementação do método `run` da classe `FishAgent`

```
1       public void run() {
2             showInfo("Reasoning cycle...");
3             getTS().reasoningCycle();
4       }
```

Na linha 3 chama-se o sistema de transição através do método `getTS` da classe `AgArch` e por meio dele pode-se chamar o método `reasoningCycle`, responsável pela execução de um ciclo de raciocínio. No Quadro 18 é demonstrado o método `createAgent` da classe `AgentFactory`, responsável pela criação de agentes através de um nome e tipo.

Quadro 18 - Implementação do método `createAgent` da classe `AgentFactory`

```
1   public static AgArch createAgent(AgentType type, String name) {
2         switch (type) {
3         case SHARK:
4               return new FishAgent(name, "shark.asl");
5         case SARDINE:
6               return new FishAgent(name, "sardine.asl");
7         default:
8         case UNKNOWN:
9               return null;
10        }
11  }
```

Na linha 1 pode-se observar os parâmetros `type` e `name`, o primeiro indica o tipo do agente a ser criado, nesse caso há apenas dois tipos, tubarão e sardinha. O segundo indica o nome do agente a ser criado que é o mesmo nome do peixe criado no Aquário Virtual, sendo responsável por diferenciar agentes do mesmo tipo e associar o peixe do aquário com o agente. Nas linhas 4 e 6 pode ser visto a criação dos dois tipos de agente existentes através de uma nova instância da classe `FishAgent`, que recebe o nome do novo agente no primeiro parâmetro e o arquivo ASL contendo a codificação de sua mente no segundo parâmetro. No Quadro 19 pode ser visto a implementação do método `manage` da classe `ActionManager`, responsável por gerenciar as ações enviadas pelos personagens do Aquário Virtual.

Quadro 19 - Implementação do método `manage` da classe `ActionManager`

```
1       public void manage(ActionType actionType, JSONObject jsonObject)
2   {
3               switch (actionType) {
4               case CREATE_AGENT:
5                       String agentTypeDescription =
6   jsonObject.getString("agentType");
7                       AgentType agentType =
8   AgentType.getAgentTypeByDescription(agentTypeDescription);
9                       String agentName = jsonObject.getString("name");
10                      AgArch newAgent = AgentFactory.createAgent(agentType,
11  agentName);
12                      AgentManager.getInstance().addAgent(newAgent);
13                      break;
14              case PERCEPTION:
15                      agentName = jsonObject.getString("name");
16                      JSONArray perceptionsArray =
17  jsonObject.getJSONArray("perceptions");
18                      FishAgent agent = (FishAgent)
19  AgentManager.getInstance().getAgent(agentName);
20                      List<Literal> perceptions =
21  convertPerceptions(perceptionsArray);
22                      agent.configureAgent();
23                      agent.setPerceptions(perceptions);
24                      agent.run();
25                      break;
26              default:
27              case UNKNOWN:
28                      break;
29              }
30      }
```

Na linha 1 pode-se observar que o método recebe os parâmetros `actionType` e `jsonObject`, dos quais o primeiro indica o tipo de ação suportada pelo Reasoner e o segundo a mensagem enviada pelo Aquário Virtual no formato JSON. A primeira ação suportada é a criar agente, nas linhas 5 e 6 pode ser visto que é pega a descrição do tipo de agente através do objeto JSON que posteriormente nas linhas 7 e 8 é convertida para seu respectivo objeto do tipo `AgentType`. Na linha 9 é pego o nome do personagem através do objeto JSON. Nas linhas 10 e 11 é chamada a fábrica de agentes passando o tipo de agente e seu nome por parâmetro, armazenando a instância do agente criado e posteriormente na linha 12 é adicionado o novo agente na classe gerenciadora de agentes.

A segunda ação suportada é a percepção. Na linha 15 é obtido o nome do personagem que teve as percepções através do objeto JSON e nas linhas 16 e 17 também através do objeto JSON é pego o *array* de percepções. Com o nome do agente obtido na linha 15, pode-se buscar seu respectivo agente através do gerenciador de agentes, conforme é observado nas linhas 18 e 19. Nas linhas 20 e 21 o `array` de percepções em formato JSON é convertido para o formato suportado pelo interpretador Jason, em seguida na linha 22 o agente é configurado através do método `configureAgent`. Na linha 23 ele recebe as novas percepções e por fim na

linha 24 é invocado o método `run`, responsável por raciocinar essas percepções. No Quadro 20 pode-se observar o método `manage` da classe `MessageManager`, responsável por gerenciar as mensagens recebidas.

Quadro 20 - Implementação do método `manage` da classe `MessageManager`

```
1  public void manage(String message) {
2              if (HAND_SHAKE.equalsIgnoreCase(message)) {
3                    sendMessage(HAND_SHAKE);
4                    return;
5              }
6
7              jsonObject = new JSONObject(message);
8              String actionTypeDescription =
9  jsonObject.getString("action");
10             ActionType actionType =
11 ActionType.getActionTypeByDescription(actionTypeDescription);
12             ActionManager.getInstance().manage(actionType, jsonObject);
13       }
```

Na linha 1 o método recebe a mensagem enviada através do *WebSocket* e caso for uma mensagem de *hand shake*, é enviada outra mensagem de *hand shake* para o Aquário Virtual, conforme pode ser visto nas linhas 2 e 3, indicando que o Reasoner está pronto para receber mensagens. Na linha 7 a mensagem é codificada para o formato JSON, nas linhas 8 e 9 através desse objeto JSON é obtido a descrição do tipo de ação e nas linhas 10 e 11 essa descrição é utilizada para obter o objeto do tipo `ActionType` correspondente. Por fim os objetos do tipo `ActionType` e JSON são enviados para o gerenciador de ação através do método `manage`, conforme observa-se na linha 12. No Quadro 21 é demonstrada a implementação da mente da sardinha através da linguagem AgentSpeak.

Quadro 21 - Implementação da mente da sardinha

```
1  /* Agent Sardine */
2
3  /* Initial beliefs and rules */
4  /* Crenças e regras iniciais */
5
6  predator("Tubarão").
7
8  /* Initial goals */
9  /* Objetivos iniciais */
10 +!explore.
11
12 /* Initial plans */
13 /* Planos iniciais */
14
15 +onPercept(Perceiver, Perceived, PerceivedType) :
16 predator(PerceivedType)
17       <- flee(Perceiver, Perceived).
18
19 +onPercept(Perceiver, Perceived, PerceivedType) : not
20 predator(PerceivedType)
21       <- explore(Perceiver).
22
23 +onCollide(Perceiver, Perceived, PerceivedType)
24       <- explore(Perceiver).
```

AgentSpeak é uma linguagem baseada na arquitetura BDI, por consequência disso a estrutura de codificação de um agente é dividida entre crenças, objetivos e planos. Na linha 6 pode ser visto a definição de uma crença, que nesse caso significa que `Tubarão` é um predador. Na linha 10 pode ser visto a definição de um objetivo, indicando que ele deve começar explorando o ambiente a sua volta. A partir da linha 15 observa-se os planos do agente, o primeiro plano conta com o gatilho `onPercept` que recebe três informações por parâmetro, o primeiro é referente ao nome do peixe perceptor, o segundo ao nome do peixe percebido e o terceiro ao tipo do peixe percebido. Esse plano será raciocinado quando o agente receber uma percepção com esse gatilho e seus três parâmetros, nesse caso será verificado se o tipo do peixe percebido é um predador, caso for, o interpretador Jason irá comunicar que a ação a ser executada é fugir, conforme pode ser visto na linha 17.

O segundo plano é referente ao mesmo gatilho do primeiro, porém esse plano trata da ação a ser executada caso o tipo do peixe percebido não for um predador, nesse caso a ação é explorar, conforme pode ser visto na linha 21. O último plano é referente ao gatilho `onCollide` que representa o evento de colisão entre dois peixes e possui os mesmos parâmetros que os gatilhos anteriores. No caso da sardinha, esse plano não teve nada de especial implementado, ao colidir com qualquer tipo de peixe a ação deve ser explorar, conforme pode ser visto na linha 24. No Quadro 22 é demonstrado a implementação da mente do tubarão através da linguagem AgentSpeak.

Quadro 22 - Implementação da mente do tubarão

```
1  /* Agent Shark */
2
3  /* Initial beliefs and rules */
4  /* Crenças e regras iniciais */
5
6  prey("Sardinha").
7
8  /* Initial goals */
9  /* Objetivos iniciais */
10 +!explore.
11
12 /* Initial plans */
13 /* Planos iniciais */
14
15 +onPercept(Perceiver, Perceived, PerceivedType) : prey(PerceivedType)
16       <- pursue(Perceiver, Perceived).
17
18 +onPercept(Perceiver, Perceived, PerceivedType) : not
19 prey(PerceivedType)
20       <- explore(Perceiver).
21
22 +onCollide(Perceiver, Perceived, PerceivedType) : prey(PerceivedType)
23       <- eat(Perceiver, Perceived).
24
25 +onCollide(Perceiver, Perceived, PerceivedType) : not
26 prey(PerceivedType)
27       <- explore(Perceiver).
```

Na linha 6 observa-se a única crença do tubarão, indicando que `Sardinha` é sua presa e na linha 10 pode ser visto o objetivo inicial que é explorar. A partir da linha 15 observa-se os planos do tubarão, o primeiro possui o gatilho `onPercept` que indica o evento de percepção. Ele possui os mesmos parâmetros que o os planos da sardinha e verifica se o peixe percebido é uma presa, caso for, a ação perseguir é gerada, conforme observa-se na linha 16.

O segundo plano é referente ao mesmo gatilho do primeiro, porém esse plano trata da ação a ser executada caso o tipo de peixe percebido não for uma presa, nesse caso a ação a ser executada é explorar, conforme pode-se observar na linha 20. O terceiro plano é referente ao gatilho `onCollide` que representa o evento de colisão entre dois peixes e também possui os mesmos parâmetros que o gatilho `onPercept`. Caso o tipo de peixe percebido pela colisão for uma presa, a ação comer é gerada, conforme pode ser visto na linha 23. O último plano é referente ao mesmo gatilho do plano anterior, porém esse trata da ação a ser executada caso o tipo de peixe percebido pela colisão não for uma presa, nesse caso a ação será explorar, conforme observa-se na linha 27.

### 3.3.5 Aquário Virtual

Nesta seção será descrita a implementação do Aquário Virtual, abordando suas principais funções e exemplos de mensagens trocadas com o Reasoner. Aquário Virtual é uma

aplicação web que possui a finalidade de simular um ecossistema marinho, auxiliando no estudo de conceitos biológicos relacionados ao mesmo.

A classe `VisEdu` é a principal classe do Aquário Virtual, se responsabilizando pelo gerenciamento da aplicação, portanto optou-se por armazenar o *WebSocket* nela. No Quadro 23 pode ser visto o trecho de código que chama a inicialização do *WebSocket*.

Quadro 23 - Implementação da função `create` da classe `VisEdu`

```
1       this.create = function (world) {
2             Game.loadAPI(new ThreeJSCustomHandler());
3             window.addEventListener('resize', this.onResize, false);
4             VisEdu.aspect = world.width() / (world.height());
5             this.setup();
6             Game.init(world, this.scene);
7             this.setupFactory();
8             this.createWebSocket("ws://localhost:8081/visedu-
9 reasoner/jason");
10            this.afterLoad();
11      }
```

Nas linhas 8 e 9 observa-se a chamada da função `createWebSocket` passando por parâmetro o *Uniform Resource Identifier* (URI) do *Servlet* principal do Reasoner, responsável por configurar a outra ponta do `WebSocket`, conforme mostra o Quadro 13. No Quadro 24 pode ser visto a implementação da função `createWebSocket`.

Quadro 24 - Implementação da função `createWebSocket` da classe `VisEdu`

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

Na linha 4 observa-se a instanciação do objeto `WebSocket` que irá se comunicar com o Reasoner e a partir da linha 10 pode ser visto que os eventos `onmessage`, `onopen`, `onclose` e `onerror` do *WebSocket* são redirecionados respectivamente para as funções `onMessage`,

onOpen, onClose e onError da classe VisEdu. No Quadro 25 pode ser visto a implementação da função onClose.

Quadro 25 - Implementação da função onClose da classe VisEdu

```
1       this.onClose = function(evt) {
2             this.isOpen = false;
3             console.log("onClose: " + evt.data);
4       }
```

Essa função é chamada quando o evento indicando o fechamento do *WebSocket* é disparado. Na linha 2 indica-se para o atributo isOpen da classe VisEdu que o *WebSocket* está fechado, enquanto que na linha 3 é escrito no console do navegador o motivo do fechamento. No Quadro 26 pode ser visto a implementação da função onOpen.

Quadro 26 - Implementação da função onOpen da classe VisEdu

```
1       this.onOpen = function(evt) {
2             if (this.timeOfInstantiation != null) {
3                   var now = Date.now();
4                   console.log("Time to establish connection: " +
5 Math.abs(now - this.timeOfInstantiation) / 1000);
6                   this.timeOfInstantiation = null;
7             }
8             this.webSocket.send(this.HAND_SHAKE);
9       }
```

Essa função é chamada quando o evento indicando a abertura do *WebSocket* é disparado. Na linha 8 observa-se o envio da mensagem de *hand shake* para o Reasoner. No Quadro 27 pode ser visto a implementação da função onMessage.

Quadro 27 - Implementação da função onMessage da classe VisEdu

```
 1       this.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       }
```

Essa função é chamada quando uma mensagem chega para o *WebSocket*. Na linha 2 é verificado se a mensagem recebida é de *hand shake*, caso for, é indicado para o atributo isOpen que o *WebSocket* está disponível para uso. Na linha 14 observa-se a conversão da mensagem para o formato JSON, da qual é retirada a parte da ação a ser tomada pela aplicação, na linha seguinte essa ação é enviada para a função executeAction. No Quadro 28

pode ser visto a implementação da função `executeAction`, responsável por executar a ação recebida.

Quadro 28 - Implementação da função `executeAction` da classe `VisEdu`

```
1       this.executeAction = function(action) {
2             var arrAction = action.split("(");
3             var actionString = arrAction[0];
4
5             switch(actionString) {
6                   case "explore":
7                         this.executeActionExplore(arrAction);
8                         break;
9                   case "flee":
10                        this.executeActionFlee(arrAction);
11                        break;
12                  case "pursue":
13                        this.executeActionPursue(arrAction);
14                        break;
15                  case "eat":
16                        this.executeActionEat(arrAction);
17                        break;
18                  default:
19            }
20
21            console.log("executeAction: " + action);
22      }
```

Na linha 2 a ação é quebrada para retirar somente a palavra que representa a ação, conforme pode ser visto na linha 3. Nas linhas seguintes a ação é verificada através de um `switch` e chama-se a função responsável pela execução da mesma, por fim é escrito a ação recebida no console do navegador. No Quadro 29 pode ser visto a implementação da função `executeActionExplore`, responsável pela execução da ação explorar.

Quadro 29 - Implementação da função `executeActionExplore` da classe `VisEdu`

```
1       this.executeActionExplore = function(arrAction) {
2             arrAction = arrAction[1].split(")");
3             arrAction = arrAction[0].split(",");
4             var perceptorName = StringUtils.replaceAll(arrAction[0],
5 "\"", "");
6             var objectsMap = Game.apiHandler.getObjectsMap();
7             var perceptorObject = objectsMap[perceptorName];
8             if(perceptorObject) {
9                   perceptorObject.storeReasonerMessage("Reasoner: " +
10 perceptorName + " deve explorar\n");
11            }
12      }
```

Nas linhas 4 e 5 é obtido o nome do peixe perceptor que é utilizado para buscar o objeto correspondente conforme pode ser visto na linha 6. Com esse objeto em mãos é chamada a função `storeReasonerMessage`, responsável por armazenar a ação para posteriormente ser exibida na área de texto. A ação explorar não demanda alterações no comportamento do peixe, pois é a que ele já executa normalmente. No Quadro 30 pode ser visto a implementação da função `executeActionFlee`, responsável por executar a ação fugir.

Quadro 30 - Implementação da função `executeActionFlee` da classe `VisEdu`

```
1       this.executeActionFlee = function(arrAction) {
2             arrAction = arrAction[1].split(")");
3             arrAction = arrAction[0].split(",");
4             var perceptorName = StringUtils.replaceAll(arrAction[0],
5 "\"", "");
6             var perceivedName = StringUtils.replaceAll(arrAction[1],
7 "\"", "");
8             var objectsMap = Game.apiHandler.getObjectsMap();
9
10            var perceptorObject = objectsMap[perceptorName];
11
12            if(perceptorObject) {
13                  perceptorObject.shouldFlee = true;
14                  perceptorObject.storeReasonerMessage("Reasoner: " +
15 perceptorName + " deve fugir de " + perceivedName + "\n");
16            }
17      }
```

A função encontra os objetos condizentes a seus nomes extraídos da mensagem recebida e utiliza o peixe perceptor através de seu atributo `shouldFlee` para indicar que ele deve fugir, conforme pode ser visto na linha 13. No Quadro 31 pode ser visto a implementação da função `executeActionPursue`, responsável por executar a ação perseguir.

Quadro 31 - Implementação da função `executeActionPursue` da classe `VisEdu`

```
1       this.executeActionPursue = function(arrAction) {
2             arrAction = arrAction[1].split(")");
3             arrAction = arrAction[0].split(",");
4             var perceptorName = StringUtils.replaceAll(arrAction[0],
5 "\"", "");
6             var perceivedName = StringUtils.replaceAll(arrAction[1],
7 "\"", "");
8             var objectsMap = Game.apiHandler.getObjectsMap();
9
10            var perceptorObject = objectsMap[perceptorName];
11
12            if(perceptorObject) {
13                  perceptorObject.shouldPursue = true;
14                  perceptorObject.storeReasonerMessage("Reasoner: " +
15 perceptorName + " deve perseguir " + perceivedName + "\n");
16            }
17      }
```

Através do atributo `shouldPursue` do objeto referente ao peixe preceptor, é indicado que ele deve perseguir, conforme pode ser visto na linha 13. No Quadro 32 pode ser visto a implementação da função `executeActionEat`, responsável pela ação comer.

Quadro 32 - Implementação da função `executeActionEat` da classe `VisEdu`

```
1       this.executeActionEat = function(arrAction) {
2             arrAction = arrAction[1].split(")");
3             arrAction = arrAction[0].split(",");
4             var perceptorName = StringUtils.replaceAll(arrAction[0],
5 "\"", "");
6             var perceivedName = StringUtils.replaceAll(arrAction[1],
7 "\"", "");
8             var objectsMap = Game.apiHandler.getObjectsMap();
9
10            var perceptorObject = objectsMap[perceptorName];
11            var perceivedObject = objectsMap[perceivedName];
12
13            if(perceivedObject) {
14
15      perceivedObject.piece.type.treeBehavior.removePiece(perceivedObjec
16 t.piece);
17                  DragAndDropController.fitTree();
18            }
19
20            if(perceptorObject) {
21                  perceptorObject.lastTimeToDeath = Date.now();
22                  perceptorObject.storeReasonerMessage("Reasoner: " +
23 perceptorName + " deve comer " + perceivedName + "\n");
24            }
25
26            Game.apiHandler.sardineEatenCount++;
27      }
```

Na linha 15 observa-se que o objeto do peixe percebido é utilizado para pegar a referência de sua peça e chama-se a função `removePiece` da respectiva classe responsável pelo comportamento da peça na árvore. Essa função remove a peça da árvore e chama outras funções responsáveis pela remoção do objeto na cena gráfica. Na linha 17 é chamada a função `fitTree` da classe controladora da árvore de peças, sua finalidade é retirar quaisquer engates sem peças pelo meio da árvore. Na linha 26 pode ser visto o incremento na quantidade de sardinhas comidas, esse atributo é usado para controlar a procriação dos tubarões.

A classe `ThreeJSCustomHandler` está entre as mais importantes da aplicação, pois ela é responsável pelo gerenciamento da parte gráfica, como por exemplo a configuração das câmeras, desenho dos objetos e consequentemente a animação dos mesmos. No Quadro 33 pode ser visto a implementação da função `addGameObject`, responsável por formar grafos de cena e mapear os objetos gráficos.

Quadro 33 - Implementação da função `addGameObject` da classe `ThreeJSCustomHandler`

```
1  ThreeJSCustomHandler.prototype.addGameObject = function(object, parent)
2  {
3        var threeObject = object.threeObject;
4        if (threeObject) {
5              parent.threeObject.add(threeObject);
6        }
7
8        // Mapeia os objetos através de seus nomes
9        if (threeObject.name != "") {
10             this.objectsMap[threeObject.name] = object;
11       }
12 }
```

Na linha 3 observa-se que o parâmetro `object` que é uma instância da classe `GameObject` possui a referência para o objeto correspondente no formato da biblioteca ThreeJS. Na linha 5 o objeto é adicionado como filho do objeto "pai", incrementando o grafo de cena. Na linha 9 é verificado se o objeto a ser mapeado possui nome, caso possuir, adiciona-se ele ao mapa `objectsMap` passando como chave seu próprio nome, conforme visto na linha 10. No Quadro 34 pode ser visto a implementação da função `removeGameObject`, responsável por remover objetos do grafo de cena e do mapa de objetos.

Quadro 34 - Implementação da função `removeGameObject` da classe `ThreeJSCustomHandler`

```
1  ThreeJSCustomHandler.prototype.removeGameObject = function(object,
2  parent) {
3        var threeObject = object.threeObject;
4        if (threeObject) {
5              parent.threeObject.remove(object.threeObject);
6        }
7
8        // Remove o objeto gráfico do mundo
9        if(Game.apiHandler.selectedFish && Game.apiHandler.selectedFish
10 == object) {
11             // Remove peixe selecionado
12             Game.apiHandler.selectedFish = null;
13             // Remove câmera auxiliar
14             Game.apiHandler.auxCamera = null;
15             // Limpa área de texto
16             $('#messages-area').val('');
17       }
18
19       delete this.objectsMap[object.name];
20 }
```

Na linha 5 o objeto é removido de seu objeto "pai", decrementando o grafo de cena, enquanto que na linha 19 ele é removido do mapa de objetos. Nas linhas 9 e 10 é verificado se existe um peixe selecionado e se ele é o objeto a ser removido, caso for, o peixe selecionado torna-se nulo (linha 12), a câmera secundária é removida (linha 14) e a área de texto com as mensagens do Reasoner é limpa (linha 16). No Quadro 35 pode ser visto a implementação da função `onRender`, que é sobrescrita na classe `ThreeJSCustomHandler` e responsável por atualizar os objetos gráficos a cada quadro.

Quadro 35 - Implementação da função `onRender` da classe `ThreeJSCustomHandler`

```
1  ThreeJSCustomHandler.prototype.onRender = function() {
2        this.controls.update();
3        VisEdu.stats.update();
4
5        // Itera sobre os objetos presentes no mapa
6        $.each(ThreeJSCustomHandler.prototype.objectsMap, function(index,
7  value){
8              var threeObject = value.threeObject;
9              if(threeObject) {
10                   if(threeObject.name != "") { // Meus objetos possuem
11 nomes
12
13       value.update(ThreeJSCustomHandler.prototype.objectsMap);
14                   }
15             }
16       })
17 }
```

Na linha 6 observa-se a iteração pelos objetos gráficos armazenados no mapa `objectsMap` e na linha 13 pode ser visto a chamada da função `update`, responsável pela atualização do objeto atual da iteração. No Quadro 36 pode ser visto a implementação da função `beforeRender`, responsável pela execução de tarefas que necessitam ser feitas antes de iniciar a chamada da função `onRender`.

Quadro 36 - Implementação da função `beforeRender` da classe `ThreeJSCustomHandler`

```
1  ThreeJSCustomHandler.prototype.beforeRender = function() {
2        var left = 0;
3        var bottom = 0;
4        var width = Game.canvas.width;
5        var height = Game.canvas.height;
6
7        this.renderer.setViewport(left, bottom, width, height);
8        this.renderer.setScissor(left, bottom, width, height);
9        this.renderer.enableScissorTest(true);
10       this.renderer.setClearColor(VisEdu.clearColor);
11       Game.camera.threeObject.aspect = width / height;
12       Game.camera.threeObject.updateProjectionMatrix();
13       this.renderer.render(Game.scene.threeObject,
14 Game.camera.threeObject);
15
16       if (this.auxCamera) {
17             left = Math.floor(width - (width / 4));
18             height = Math.floor(height / 4);
19             this.renderer.setViewport(left, bottom, width, height);
20             this.renderer.setScissor(left, bottom, width, height);
21             this.renderer.enableScissorTest(true);
22             this.renderer.setClearColor(VisEdu.backgroundColor);
23             this.auxCamera.aspect = left / height;
24             this.renderer.render(Game.scene.threeObject,
25 this.auxCamera);
26       }
27 }
```

No caso dessa aplicação é necessário que a configuração das câmeras seja feita pela função `beforeRender`. A configuração da câmera principal se torna necessária sempre, pois

ela deve estar habilitada durante toda a execução da aplicação. A configuração da segunda é feita somente quando uma peça de peixe é selecionada, habilitando a câmera responsável pela visão do peixe. Na linha 16 é verificado se o atributo `auxCamera` possui valor definido, caso positivo, a segunda câmera pode ser configurada.

Na linha 7 observa-se a função `setViewPort`, responsável pela definição do tamanho da janela de visualização da câmera principal. Se tratando de uma janela de visualização, trabalha-se com um ambiente de duas dimensões, ou seja, x e y, com x representando a largura e y a altura. O elemento `canvas` da linguagem HTML tem seu ponto inicial no canto inferior esquerdo. Os parâmetros `left`, `bottom`, `width` e `height` representam respectivamente o valor em x relativo ao início da largura da tela, o valor em y relativo ao início da altura da tela, a largura da tela e a altura da tela.

Na linha 8 observa-se a função `setScissor`, responsável por definir o tamanho da área que será aplicado o teste da tesoura. Esse teste tem por objetivo eliminar da rotina de desenho fragmentos de objetos gráficos ou até mesmo objetos inteiros que não estão dentro de determinada área (OPENGL, 2015a). Nesse caso é a mesma área de visualização da câmera, pois não faz sentido gastar recursos computacionais para desenhar algo que nem pode ser visto. A ativação deste teste é feita na linha 9 através da função `enableScissorTest`.

Na linha 10 pode ser visto a função `setClearColor`, responsável por definir a cor de fundo da câmera caso não possua nenhum objeto gráfico a ser desenhado. Na linha 11 observa-se a definição do atributo `aspect` da câmera principal, que representa a proporção entre a altura e a largura da visualização da câmera e é o principal responsável por imagens “esticadas”. Para a alteração do atributo `aspect` refletir no ambiente virtual é necessário atualizar a matriz de projeção da câmera, isto é feito através do método `updateProjectionMatrix`, conforme pode ser visto na linha 12.

Por fim, para adicionar a maioria das configurações anteriores de fato na câmera, utiliza-se a função `render`, responsável por desenhar tudo o que existe na cena presente no primeiro parâmetro para a visualização da câmera passada pelo segundo parâmetro, conforme pode ser visto nas linhas 13 e 14.

Foram criados três componentes customizados para esta aplicação, são eles, `AquariumRenderComponent`, `FishRenderComponent` e `PanoramicBackgroundRenderComponent`. Ambos herdam da classe `RenderableComponent` do Motor de Jogos e representam respectivamente o componente responsável por construir o objeto gráfico do aquário, o componente responsável por construir o objeto gráfico dos peixes

e o componente responsável por construir o objeto gráfico do fundo panorâmico. No Quadro 37 pode ser visto a implementação da função `genThreeObject` da classe `AquariumRenderComponent`.

Quadro 37 - Implementação da função `genThreeObject` da classe `AquariumRenderComponent`

```
1 AquariumRenderComponent.prototype.genThreeObject = function(){
2       return this.createAquarium();
3 }
```

Essa função é responsável por retornar um objeto tridimensional no formato da biblioteca ThreeJS e é sobrescrita da classe `RenderableComponent`. Na linha 2 observa-se a chamada à função `createAquarium` que de fato irá criar o objeto tridimensional, sua implementação pode ser vista no Quadro 38.

Quadro 38 - Implementação da função `createAquarium` da classe `AquariumRenderComponent`

```
 1 AquariumRenderComponent.prototype.createAquarium = function() {
 2       var aquarium = new THREE.Object3D();
 3
 4       var aquariumGeometry = new THREE.SphereGeometry(this.diameter,
 5 30, 30);
 6
 7       aquarium.add(this.createSandGround());
 8       aquarium.add(this.createSupport());
 9       aquarium.add(this.createAquariumFront(aquariumGeometry));
10       aquarium.add(this.createAquariumBack(aquariumGeometry));
11       this.addPlants(aquarium);
12       this.addRocks(aquarium);
13
14       return aquarium;
15 }
```

Na linha 2 observa-se a instanciação do objeto do tipo `Object3D` que serve como um container para outros objetos que irão compor o aquário, criando um grafo de cena, conforme será visto a seguir. Nas linhas 4 e 5 é criado a geometria tridimensional do vidro do aquário, ele possui o formato esférico, por isso a instanciação do objeto `SphereGeometry`. Nas linhas 7 a 12 pode ser visto funções que adicionam novos objetos ao container `aquarium`. No Quadro 39 pode ser visto a implementação da função `createSandGround`, responsável pela criação do chão de areia do aquário.

Quadro 39 - Implementação da função `createSandGround` da classe `AquariumRenderComponent`

```
1  AquariumRenderComponent.prototype.createSandGround = function() {
2        var circleGeometry = new THREE.CircleGeometry(87, 60);
3        var sandTexture = THREE.ImageUtils.loadTexture(
4  './resources/sandTexture.jpg' );
5        sandTexture.minFilter = THREE.LinearFilter;
6        var material = new THREE.MeshBasicMaterial( { map: sandTexture,
7  overdraw: true } );
8        var sandGround = new THREE.Mesh( circleGeometry, material );
9        sandGround.rotateX(-(Math.PI / 2));
10       sandGround.position.y = -49;
11       return sandGround;
12 }
```

Na linha 2 é instanciado um objeto do tipo `CircleGeometry`, que representa a geometria tridimensional com formato circular. Nas linhas 3 e 4 é criado a textura de areia a partir da imagem `sandTexture.jpg`, na linha 5 é aplicado um filtro linear a essa imagem e nas linhas 6 e 7 é criado o material a ser aplicado no objeto que representa o chão de areia. Com a geometria e o material criado é possível instanciar uma malha tridimensional, conforme pode ser visto na linha 8 através do construtor da classe `Mesh`. No Quadro 40 é demonstrado a implementação da função `createSupport`, responsável por criar o suporte do aquário.

Quadro 40 - Implementação da função `createSupport` da classe `AquariumRenderComponent`

```
1  AquariumRenderComponent.prototype.createSupport = function() {
2        var supportHeight = 60;
3        var geometry = new THREE.CylinderGeometry(93, 37, supportHeight,
4  50, 50, false) ;
5        var material = new THREE.MeshBasicMaterial( { map:
6  THREE.ImageUtils.loadTexture( './resources/supportTexture.jpg' ),
7  overdraw: true } );
8        var support = new THREE.Mesh( geometry, material ) ;
9        support.position.y = -80;
10       support.add(this.createSupportBase(material, supportHeight));
11       return support;
12 }
```

Nas linhas 3 e 4 é criada a geometria do suporte, seu formato é cônico, porém é utilizado a classe `CylinderGeometry`, deixando a parte de baixo mais fina gerando um cone. Nas linhas 5, 6 e 7 pode ser visto a criação do material utilizado pelo suporte, que é do tipo `MeshBasicMaterial`, esse tipo de material é o mais básico e fácil de ser utilizado em uma malha tridimensional. Na linha 10 observa-se a chamada à função `createSupportBase`, que retorna a base do suporte que na mesma linha é adicionada ao suporte recém criado, alimentando o grafo de cena. No Quadro 41 pode ser visto a implementação da função `createSupportBase`.

Quadro 41 - Implementação da função `createSupportBase` da classe `AquariumRenderComponent`

```
1  AquariumRenderComponent.prototype.createSupportBase = function(material,
2  supportHeight) {
3        var geometry = new THREE.CylinderGeometry(70, 70, 10, 50, 10,
4  false);
5        var supportBase =  new THREE.Mesh( geometry, material ) ;
6        supportBase.position.y = -(supportHeight / 2);
7        return supportBase
8  }
```

A base do suporte possui o formato cilíndrico e utiliza a classe `CylinderGeometry`, com a largura da parte de cima e de baixo igual, conforme pode ser observado nas linhas 3 e 4. Na linha 6 pode ser visto o deslocamento da base em metade da altura do suporte em relação ao eixo y, pois quando ele é criado, sua posição inicial é no meio do objeto pai, que nesse caso é o suporte. No Quadro 42 demonstra-se a implementação da função `createAquariumFront`, responsável pela criação da parte de fora do vidro do aquário.

Quadro 42 - Implementação da função `createAquariumFront` da classe `AquariumRenderComponent`

```
 1  AquariumRenderComponent.prototype.createAquariumFront =
 2  function(geometry) {
 3        // Cria mapa de textura cúbica
 4        var backgroundPath = "./resources/background/";
 5        var textureCubeMap = [ backgroundPath + "posx.jpg",
 6  backgroundPath + "negx.jpg",
 7                               backgroundPath + "posy.jpg",
 8  backgroundPath + "negy.jpg",
 9                               backgroundPath + "posz.jpg",
10  backgroundPath + "negz.jpg" ];
11
12        // Carrega a textura cúbica
13        var textureCube = THREE.ImageUtils.loadTextureCube(
14  textureCubeMap, THREE.CubeRefractionMapping );
15
16        // Cria material refratado para visão de fora do aquário
17        var refractedMaterial = new THREE.MeshLambertMaterial( { color:
18  0x40ffdf, envMap: textureCube, refractionRatio: 0.98, reflectivity:0.8,
19  transparent: true, opacity: 0.5 } );
20
21        var aquariumFront = new THREE.Mesh( geometry, refractedMaterial
22  );
23        return aquariumFront;
24  }
```

Nas linhas 4 a 10 é criado um mapa com seis imagens representando as seis faces de um cubo, esse mapa é utilizado para criar uma textura cúbica igual à do fundo panorâmico, conforme pode-se observar nas linhas 13 e 14. Na linha 14 também pode ser visto a classe `CubeRefractionMapping`, representando o tipo de mapeamento dessa textura, que nesse caso é um mapeamento de refração.

Nas linhas 17, 18 e 19 é construído o material refratado, que leva em consideração a cor do material, o fundo panorâmico, a taxa de reflexão e a opacidade, gerando a refração do

ambiente e reproduzindo um efeito de vidro. O tipo de material utilizado é o `MeshLambertMaterial`, recomendado para superfícies não brilhantes e que nos permite a criação da refração. No Quadro 43 pode ser visto a implementação da função `createAquariumBack`, responsável pela criação da parte de dentro do vidro do aquário.

Quadro 43 - Implementação da função `createAquariumBack` da classe `AquariumRenderComponent`

```
1  AquariumRenderComponent.prototype.createAquariumBack =
2  function(geometry) {
3        // Cria material de água para visão de dentro do aquário
4        var materialBack = new THREE.MeshLambertMaterial( { color:
5  0x40ffdf, transparent: true, opacity: 0.5 } );
6        materialBack.side = THREE.BackSide;
7        var aquariumBack = new THREE.Mesh( geometry, materialBack );
8        return aquariumBack;
9  }
```

O procedimento de criação da parte de dentro do aquário é bem parecida com a parte de fora, porém no lado de dentro não existe refração, possui somente a cor da água com transparência, permitindo a visualização do ambiente em volta. No Quadro 44 demonstra-se a implementação da função `addPlants`, responsável por centralizar a inclusão de plantas no aquário.

Quadro 44 - Implementação da função `addPlants` da classe `AquariumRenderComponent`

```
1  AquariumRenderComponent.prototype.addPlants = function(aquarium) {
2        THREE.Loader.Handlers.add( /\.dds$/i, new THREE.DDSLoader() );
3        var loader = new THREE.OBJMTLLoader();
4        loader.load( "./resources/models/plant/plant.obj",
5  "./resources/models/plant/plant.mtl", function ( object ) {
6              object.position.y = -50;
7              aquarium.add( object );
8        }, ThreeUtils.onProgress, ThreeUtils.onError );
9  }
```

Na linha 2 pode ser visto a adição de uma instância da classe `DDSLoader`, responsável por gerenciar o carregamento de texturas internamente na biblioteca ThreeJS. Na linha 3 é instanciado o objeto responsável por carregar as informações contidas nos arquivos OBJ e MTL, nas linhas seguintes pode-se observar a chamada da função `load` que inicia o carregamento.

O terceiro parâmetro da função `load` é uma função que irá receber o objeto tridimensional quando o mesmo terminar de ser carregado, adicionando-o ao objeto principal do aquário que serve como container. O fluxo da aplicação segue normalmente ao chamar a função `load`, porém quando o objeto terminar de ser carregado, essa função será executada, simulando um trabalho concorrente. No Quadro 45 pode ser visto a implementação da função `addRocks`.

Quadro 45 - Implementação da função `addRocks` da classe `AquariumRenderComponent`

```
1  AquariumRenderComponent.prototype.addRocks = function(aquarium) {
2        this.addArchRock(aquarium);
3        this.addSmallBlueRock(aquarium);
4        this.addBigBlueRock(aquarium);
5  }
```

Essa função é responsável por centralizar a inclusão de rochas no aquário e nos próximos quadros é possível observar a inclusão das três rochas vistas nas linhas 2, 3 e 4. No Quadro 46 pode ser visto a implementação da função `addArchRock`, responsável por adicionar a rocha arcada ao aquário.

Quadro 46 - Implementação da função `addArchRock` da classe `AquariumRenderComponent`

```
1  AquariumRenderComponent.prototype.addArchRock = function(aquarium) {
2        THREE.Loader.Handlers.add( /\.dds$/i, new THREE.DDSLoader() );
3        var loader = new THREE.OBJMTLLoader();
4        loader.load( "./resources/models/arch_rock/arch_rock.obj",
5  "./resources/models/arch_rock/arch_rock.mtl", function ( object ) {
6              object.position.x = -30;
7              object.position.y = -50;
8              object.position.z = 30;
9              object.scale.x = 5;
10             object.scale.y = 5;
11             object.scale.z = 5;
12             aquarium.add( object );
13       }, ThreeUtils.onProgress, ThreeUtils.onError );
14 }
```

A inclusão da rocha arcada funciona da mesma forma que a inclusão da planta, o que muda são os arquivos OBJ e MTL a serem carregados e as transformações que esse objeto sofre. Nas linhas 9, 10 e 11 é possível observar o escalonamento do objeto, o qual aumenta-se em cinco vezes o seu tamanho original. Percebe-se que é aplicado para ambos os eixos, evitando distorcer o objeto. No Quadro 47 pode ser visto a implementação da função `addSmallBlueRock`, responsável por construir a pedra azul pequena.

Quadro 47 - Implementação da função `addSmallBlueRock` da classe `AquariumRenderComponent`

```
1  AquariumRenderComponent.prototype.addSmallBlueRock = function(aquarium)
2  {
3        THREE.Loader.Handlers.add( /\.dds$/i, new THREE.DDSLoader() );
4        var loader = new THREE.OBJMTLLoader();
5        loader.load( "./resources/models/rock/rock.obj",
6  "./resources/models/rock/rock.mtl", function ( object ) {
7              object.position.x = 40;
8              object.position.y = -45;
9              object.position.z = 50;
10             object.scale.x = 2;
11             object.scale.y = 2;
12             object.scale.z = 2;
13             aquarium.add( object );
14       }, ThreeUtils.onProgress, ThreeUtils.onError );
15 }
```

O carregamento desse objeto funciona igual a pedra anterior, novamente o que muda são as transformações sofridas e os arquivos a serem carregados. No Quadro 48 pode-se

observar a implementação da função `addBigBlueRock`, responsável pela construção da rocha azul grande.

Quadro 48 - Implementação da função `addBigBlueRock` da classe `AquariumRenderComponent`

```
1  AquariumRenderComponent.prototype.addBigBlueRock = function(aquarium) {
2        THREE.Loader.Handlers.add( /\.dds$/i, new THREE.DDSLoader() );
3        var loader = new THREE.OBJMTLLoader();
4        loader.load( "./resources/models/rock/rock.obj",
5  "./resources/models/rock/rock.mtl", function ( object ) {
6              object.position.x = 20;
7              object.position.y = -50;
8              object.position.z = -40;
9              object.scale.x = 5;
10             object.scale.y = 5;
11             object.scale.z = 5;
12             aquarium.add( object );
13       }, ThreeUtils.onProgress, ThreeUtils.onError );
14 }
```

Percebe-se que a forma de carregar o objeto e os arquivos a serem carregados são idênticos à pedra anterior, sendo diferenciada pelo posicionamento e pela escala, enquanto que a anterior teve seu tamanho aumentado em duas vezes, essa pedra teve seu tamanho aumentado em cinco vezes. No Quadro 49 pode ser visto a implementação da função `genThreeObject` da classe `FishRenderComponent`, responsável por gerar o objeto gráfico dos peixes.

Quadro 49 - Implementação da função `genThreeObject` da classe `FishRenderComponent`

```
1  FishRenderComponent.prototype.genThreeObject = function(){
2        THREE.Loader.Handlers.add( /\.dds$/i, new THREE.DDSLoader() );
3        var container = new THREE.Object3D();
4
5        var name = this.piece.properties['name'];
6        var obj = this.piece.properties['obj'];
7        var mtl = this.piece.properties['mtl'];
8        var showVision = this.piece.properties['showVision'];
9        var visionRadius = this.piece.properties['visionRadius'];
10       var visionRange = this.piece.properties['visionRange'];
11
12       var loader = new THREE.OBJMTLLoader();
13       loader.load(obj, mtl, function ( object ) {
14             this.bbox = new THREE.Box3().setFromObject(object);
15             var size = this.bbox.size();
16             container.userData = { size: size };
17
18             var geometry = new THREE.CylinderGeometry(visionRadius, 1,
19 visionRange, 4, 4, false) ;
20             var material = new THREE.MeshBasicMaterial( { color:
21 0xff0000} );
22             var vision = new THREE.Mesh(geometry, material);
23
24             vision.translateZ((visionRange / 2) + (size.z / 2));
25
26             vision.rotateX(Math.PI / 2); //90 graus
27             vision.name = "vision";
28             vision.visible = showVision;
29
30             var camera = new THREE.PerspectiveCamera(45, VisEdu.aspect,
31 0.1, 5000);
32             camera.translateZ((visionRange / 2) + (size.z / 2));
33             camera.rotateY((Math.PI / 1.158));
34
35             container.add(vision);
36             container.add(object);
37             container.add(camera);
38       }, ThreeUtils.onProgress, ThreeUtils.onError );
39
40       container.name = name;
41
42       return container;
```

Nas linhas 5 a 10 são obtidas as propriedades da peça do peixe a ser criado, pois é a peça que contém as propriedades que os diferenciam. Na linha 14 é instanciado um objeto do tipo `Box3` com base no objeto carregado que representa sua *bounding box*, ela serve para calcular o tamanho do peixe em si, pois posteriormente o objeto que representa o peixe armazenará a malha tridimensional que representa graficamente sua a visão e a câmera que exibe sua visão, dificultando o cálculo do tamanho. Na linha 16 observa-se que o tamanho do peixe é armazenado no atributo `userData` do objeto, esse atributo permite o armazenamento de informações customizadas.

Para que o alcance da visão seja visualizada, constrói-se uma geometria em forma de cone, com um material básico de cor vermelha, conforme pode-se observar nas linhas 18 a 22.

Na linha 24 nota-se a função `translateZ`, responsável pelo deslocamento do objeto no eixo z, recebendo uma fórmula matemática. O deslocamento no eixo z evita a preocupação com a orientação do peixe em relação ao ambiente, pois ele sempre será deslocado para a direção que sua frente está apontada. Essa fórmula tem a finalidade de calcular o valor necessário para que a visão fique na frente do peixe, pois inicialmente ela fica no meio, por isso leva-se em consideração o tamanho do peixe e o alcance da visão.

Na linha 28 é indicado com base na propriedade `showVision` da peça, se a representação gráfica da visão deve ser exibida ou ocultada. Posteriormente nas linhas 30 e 31 observa-se a criação da câmera que irá exibir o que o peixe vê e que ficará embutida no mesmo, nota-se que ela é do tipo `PerspectiveCamera`, tipo responsável por proporcionar uma noção de profundidade, ideal para ambientes tridimensionais. No Quadro 50 pode ser visto a implementação da função `genThreeObject` da classe `PanoramicBackgroundRenderComponent`, responsável por construir o fundo panorâmico, que nada mais é do que um cubo com imagens em suas faces internas e que receberá o aquário em seu interior, gerando uma vista panorâmica.

Quadro 50 - Implementação da função `genThreeObject` da classe `PanoramicBackgroundRenderComponent`

```
1  PanoramicBackgroundRenderComponent.prototype.genThreeObject =
2  function(){
3        var backgroundPath = "./resources/background/";
4
5        // Cria mapa de textura cúbica
6        var textureCubeMap = [ backgroundPath + "posx.jpg",
7  backgroundPath + "negx.jpg",
8                               backgroundPath + "posy.jpg",
9  backgroundPath + "negy.jpg",
10                              backgroundPath + "posz.jpg",
11 backgroundPath + "negz.jpg" ];
12
13       // Carrega a textura cúbica
14       var textureCube = THREE.ImageUtils.loadTextureCube(
15 textureCubeMap, THREE.CubeRefractionMapping );
16
17       // Atribui a textura cúbica ao shader
18       var shader = THREE.ShaderLib[ "cube" ];
19       shader.uniforms[ "tCube" ].value = textureCube;
20
21       // Cria o material baseado em shader
22       var material = new THREE.ShaderMaterial( {
23             fragmentShader: shader.fragmentShader,
24             vertexShader: shader.vertexShader,
25             uniforms: shader.uniforms,
26             side: THREE.BackSide
27       } ), mesh = new THREE.Mesh( new THREE.BoxGeometry( this.size,
28 this.size, this.size ), material );
29
30       return mesh;
31 }
```

Na linha 18 observa-se a criação do *shader*, que basicamente é um trecho de código a ser executado diretamente pela *Graphics Processing Unit* (GPU), ou unidade de processamento gráfico (OPENGL, 2015b). A biblioteca ThreeJS possui uma lista de *shaders* codificados, como o responsável por construir um cubo, obtido através do *array* `ShaderLib` na linha 18. Na linha seguinte atribui-se a textura cúbica a esse *shader*, e em seguida cria-se o material baseado em *shader* do tipo `ShaderMaterial`.

O primeiro parâmetro do material é o *Fragment Shader*, programa responsável por definir as cores para cada vértice do objeto (OPENGL, 2015c). O segundo parâmetro é o *Vertex Shader*, programa responsável por gerar o mapeamento de vértices (OPENGL, 2015d). O terceiro parâmetro indica que dados como mapas de iluminação e sombra são uniformes e compartilhados entre o *Fragment Shader* e o *Vertex Shader* (THREEJS, 2015). Por fim, o quarto parâmetro indica de que lado o desenho será visualizado, nesse caso somente o lado de dentro, representado pelo atributo `BackSide`. No Quadro 51 é demonstrado a implementação da função `initialize` da classe `FishObject`, responsável por inicializar as propriedades de uma nova instância da classe.

Quadro 51 - Implementação da função `initialize` da classe `FishObject`

```
1  JSUtils.addMethod(FishObject.prototype, "initialize",
2        function(component, piece){
3              this.initialize(0, 0, 0, 0, 0, 0); // Sem importância os
4  valores aqui
5
6              ComponentUtils.addComponent(this, this.component =
7  component);
8
9              this.piece = piece;
10
11             var direction = (Math.floor((Math.random() * 2) + 1) == 1);
12 // true ou false
13             this.moveSpeed = this.piece.properties['moveSpeed'];
14             this.deltaRotation = this.moveSpeed * (Math.PI / 180);
15             if(direction) {
16                   this.deltaRotation *= -1;
17             }
18             this.deltaHeight = this.deltaRotation;
19
20             this.name = this.piece.properties['name'];
21             this.type = this.piece.type;
22
23             this.lastTime = Date.now();
24             this.lastTimeToDeath = Date.now();
25             this.deltaHeight = 0.0005;
26
27             var positionX = (Math.floor((Math.random() * 100) - 50));
28 // de -50 a 50
29             var positionZ = (Math.floor((Math.random() * 100) - 50));
30             this.threeObject.translateX(positionX);
31             this.threeObject.translateZ(positionZ);
32
33             this.shouldFlee = false;
34             this.shouldPursue = false;
35             this.shouldEat = this.piece.properties['shouldEat'];
36
37             this.pointZero = new THREE.Vector3(0, 0, 0);
38
39             this.frameCount = 0;
40
41             this.reasonerMessages = "";
42             this.lastMessage = "";
43
44             VisEdu.webSocket.send("{ \"action\" : \"createAgent\",
45 \"name\" : \"" + this.name + "\", \"agentType\" : \"" + this.type.name
46 + "\" }");
47             return this;
48       }
49 );
```

Na linha 11 é gerado de forma randômica um valor binário representando a direção inicial que o peixe deve se mover, se é para a esquerda ou direita. Com base nessa direção é multiplicado o valor da variação de rotação por -1, invertendo sua direção, conforme é observado na linha 16. Nas linhas 27 e 29 é gerado um valor randômico entre -50 e 50 para o eixo x e outro para o eixo z, responsáveis por criar os peixes em posições diferentes dentro do aquário. Nas linhas 44 a 46 pode ser visto o envio de uma mensagem para o Reasoner, essa

mensagem é responsável pela criação do agente no mesmo e percebe-se que ela armazena a ação `createAgent`, o nome do peixe a ter seu agente criado e seu tipo. No Quadro 52 pode ser visto a implementação da função `update`, responsável por atualizar as informações do peixe e gerar a animação.

Quadro 52 - Implementação da função `update` da classe `FishObject`

```
1  FishObject.prototype.update = function(objectsMap) {
2        if(this.isReady()) {
3              if(this.vision == null) {
4                    this.vision = this.threeObject.children[0];
5                    this.camera = this.threeObject.children[2];
6              }
7
8              if(Game.apiHandler.properties['speedMultiplier'] != 0) {
9                    if(this.frameCount >= 2) {
10                         if(Game.apiHandler.percept) {
11                               this.percept(objectsMap);
12                         }
13                         this.frameCount = 0;
14                   }
15                   this.move();
16             }
17       }
18       this.frameCount++;
19 }
```

Na linha 2 observa-se a chamada da função `isReady`, responsável por verificar se o carregamento do objeto tridimensional terminou e se ele está pronto para uso. Após isso é verificado se a visão do peixe ainda está nula, conforme visto na linha 3. Nas linhas 4 e 5 são atribuídos os objetos que representam a visão e a câmera de visão do peixe respectivamente, obtidos através do objeto principal que os armazena em forma de grafo de cena.

Na linha 8 é verificado o multiplicador de velocidade do mundo, caso for 0 o peixe não faz nada. Na linha 9 é verificado a contagem de quadros, caso for igual ou maior que dois é executada a percepção, pois se isso é feito a cada quadro há uma perda de desempenho considerável. Quanto maior for o intervalo de quadros para a execução da percepção, maior o desempenho, porém menor a quantidade de respostas do processo de raciocínio, podendo até mesmo influenciar na qualidade da animação. No Quadro 53 pode ser visto a implementação da função `percept`, responsável por perceber o ambiente em volta do peixe.

Quadro 53 - Implementação da função `percept` da classe `FishObject`

```
1  FishObject.prototype.percept = function(objectsMap) {
2        if(this.vision != null) {
3              var thisObject = this;
4              $.each(objectsMap, function(index, value){
5                    if(thisObject != value) {
6                          if(value.threeObject) {
7                                if(value.threeObject.name != "") { //
8  Meus objetos possuem nomes
9
10       if(thisObject.detectCollisionBBox(thisObject.threeObject.childre
11 n[0], value.threeObject.children[1])) { // Visão deste com o outro
12 peixe
13                                                VisEdu.webSocket.send("{
14 \"action\" : \"perception\", \"name\" : \"" +
15 thisObject.threeObject.name + "\",\"perceptions\" : [\"onPercept(\\\""
16 + thisObject.threeObject.name + "\\\",\\\"" + value.threeObject.name +
17 "\\\",\\\"" + value.type.name + "\\\")\"] }");
18                                                if(thisObject.shouldEat) {
19
20       if(thisObject.detectCollisionBBox(thisObject.threeObject.childre
21 n[1], value.threeObject.children[1])) {
22
23       VisEdu.webSocket.send("{ \"action\" : \"perception\", \"name\" :
24 \"" + thisObject.threeObject.name + "\",\"perceptions\" :
25 [\"onCollide(\\\"" + thisObject.threeObject.name + "\\\",\\\"" +
26 value.threeObject.name + "\\\",\\\"" + value.type.name + "\\\")\"]
27 }");
28                                                      }
29                                                }
30                                          }
31                                    }
32                              }
33                        }
34             });
35       }
36 }
```

Na linha 4 observa-se que é feita uma iteração sobre todos os objetos mapeados, pois é preciso verificar todos para garantir que todas as percepções sejam efetuadas. Nas linhas 10 e 11 é verificado se existe colisão entre a visão do peixe perceptor com o objeto atual da iteração, caso existir é enviada uma mensagem para o Reasoner informando a ação como percepção e passando as informações dessa percepção, tais como o nome do peixe perceptor, o nome e o tipo do objeto percebido, conforme pode ser visto nas linhas 13 a 17.

Caso a propriedade do peixe perceptor indique que ele deve comer, verifica-se a colisão entre ele e o objeto atual da iteração, caso houver uma colisão, é enviada outra mensagem para o Reasoner, informando que ocorreu uma colisão e as informações dos objetos envolvidos, conforme pode ser visto nas linhas 23 a 27. No Quadro 54 demonstra-se a implementação da função `detectCollisionBBox`, responsável por verificar se existe colisão entre dois objetos por meio de suas *bounding box*.

Quadro 54 - Implementação da função `detectCollisionBBox` da classe `FishObject`

```
1  FishObject.prototype.detectCollisionBBox = function(object1, object2) {
2        if(object1 && object2) {
3              var object1BBox = new THREE.Box3().setFromObject(object1);
4              var object2BBox = new THREE.Box3().setFromObject(object2);
5              var collision = object1BBox.isIntersectionBox(object2BBox);
6              if(collision) {
7                    return true;
8              }
9        }
10       return false;
11 }
```

Nas linhas 3 e 4 são criadas as *bounding box* dos objetos, aqui encontra-se uma etapa do processo de "renderização" que tem alto custo de processamento, pois toda vez que é necessário o seu uso deve-se recalcular as novas *bounding box*, prejudicando muito a performance das animações. A função `isIntersectionBox` pertencente ao objeto `Box3` da biblioteca ThreeJS e é responsável por indicar se há intersecção entre os dois objetos, conforme pode ser visto na linha 5.

No Quadro 55 é demonstrado a implementação da função `initialize` da classe `AquariumObject`, responsável por inicializar as propriedades de uma nova instância da classe.

Quadro 55 - Implementação da função `initialize` da classe `AquariumObject`

```
1  JSUtils.addMethod(AquariumObject.prototype, "initialize",
2        function(component, piece){
3              this.initialize(0, 0, 0, 0, 0, 0); // Sem importância os
4  valores aqui
5              ComponentUtils.addComponent(this, this.component =
6  component);
7              this.piece = piece;
8              this.type = this.piece.type;
9              this.name = this.piece.properties['name'];
10             this.lastTime = Date.now();
11             this.lastTimeBubbles = Date.now();
12             this.colorLevel = [0x40ffdf, 0x40eedf, 0x40dddf, 0x40ccdf,
13 0x40bbdf, 0x40aadf, 0x4099df, 0x4088df, 0x4077df, 0x4066df, 0x4055df];
14             this.currentColorLevel = 0;
15             this.aquariumFront = this.threeObject.children[2];
16             this.aquariumBack = this.threeObject.children[3];
17             this.bubbleCount = 0;
18             this.bubbleAnimationSpeed = 0.3;
19             return this;
20       }
21 );
```

Nas linhas 12 e 13 observa-se uma lista de cores em hexadecimal que representam os dez níveis de cor que a água pode apresentar, o primeiro valor indica a cor com mais quantidade de plâncton e a última com a menor quantidade. Na linha 14 é armazenado o nível atual da cor da água, por padrão ela começa esverdeada indicando muito plâncton, já que o aquário inicia vazio. Nas linhas 15 e 16 observa-se o armazenamento das partes de fora e de dentro do aquário, as quais tem a cor dos seus materiais alterada conforme o nível de

plâncton. No Quadro 56 pode ser visto a implementação da função `update`, responsável por atualizar o aquário a cada quadro.

Quadro 56 - Implementação da função `update` da classe `AquariumObject`

```
1  AquariumObject.prototype.update = function(objectsMap) {
2        if(Game.apiHandler.properties['speedMultiplier'] != 0) {
3              var currentTime = Date.now();
4              if(this.bubbleCount < 20 && currentTime >
5  (this.lastTimeBubbles + (3000 / this.getTimeSpeedMultiplier()))) {
6                    this.createBubble();
7                    this.lastTimeBubbles = Date.now();
8              }
9              this.animateBubbles();
10             if(currentTime > (this.lastTime + (20000 /
11 this.getTimeSpeedMultiplier()))) {
12                   var currentSardineCount =
13 Game.apiHandler.sardineCount;
14                   var currentSardineHalfCount = currentSardineCount / 2;
15                   this.updateColorLevel(currentSardineCount,
16 currentSardineHalfCount);
17                   this.verifyProcreation(currentSardineCount,
18 currentSardineHalfCount);
19                   this.lastTime = currentTime;
20             }
21       }
22 }
```

Na linha 2 é verificado o multiplicador de velocidade do mundo e caso for 0 o aquário não faz nada. Nas linhas 4 e 5 é verificado se a quantidade de bolhas não passou da quantidade estipulada (20) e se já passou o tempo necessário para a criação de uma nova bolha. Na linha 6 é chamada a função responsável por criar uma bolha, nas linhas 15 e 16 é chamada a função que atualiza o nível de cor da água e nas linhas 17 e 18 é chamada a função responsável por verificar a procriação dos peixes. No Quadro 57 pode ser visto a implementação da função `updateColorLevel`.

Quadro 57 - Implementação da função `updateColorLevel` da classe `AquariumObject`

```
1  AquariumObject.prototype.updateColorLevel =
2  function(currentSardineCount, currentSardineHalfCount) {
3        if(Math.round(currentSardineCount / 1.5) > this.currentColorLevel
4  && this.currentColorLevel < 10) {
5              this.currentColorLevel++;
6        } else {
7              if(Math.round(currentSardineCount / 1.5) <
8  this.currentColorLevel && this.currentColorLevel > 0) {
9                    this.currentColorLevel--;
10             }
11       }
12       this.aquariumFront.material.color.setHex
13 (this.colorLevel[this.currentColorLevel]);
14       this.aquariumBack.material.color.setHex
15 (this.colorLevel[this.currentColorLevel]);
16 }
```

Nas linhas 3 e 4 é verificado se a quantidade de sardinhas dividido por 1.5 é maior do que o nível de cor atual e se esse nível já não está no máximo, caso essas condições forem

verdadeiras o nível será incrementado resultando em uma cor mais limpa (menos plâncton). O valor 1.5 se dá pelo fato de permitir no máximo quinze sardinhas, portanto a cada duas (`Math.round` arredonda o valor) sardinhas a mais ou a menos no aquário o nível de cor é alterado. Nas linhas 7 e 8 é verificado a condição contrária à anterior e caso ela seja verdadeira o nível de cor é diminuído. Nas linhas 12 a 15 observa-se a atribuição da nova cor para os materiais da parte da frente e de fora do aquário. No Quadro 58 pode ser visto a implementação da função `verifyProcreation`, responsável por verificar a procriação dos peixes.

Quadro 58 - Implementação da função `verifyProcreation` da classe `AquariumObject`

```
1  AquariumObject.prototype.verifyProcreation =
2  function(currentSardineCount, currentSardineHalfCount) {
3        this.verifySardineProcreation(currentSardineCount,
4  currentSardineHalfCount);
5        this.verifySharkProcreation();
6  }
```

Essa função chama as funções responsáveis pela verificação da procriação das sardinhas e dos tubarões, conforme pode ser visto nas linhas 3, 4 e 5. No Quadro 59 é demonstrado a implementação função `verifySardineProcreation`, responsável por verificar a procriação das sardinhas.

Quadro 59 - Implementação da função `verifySardineProcreation` da classe `AquariumObject`

```
 1  AquariumObject.prototype.verifySardineProcreation =
 2  function(currentSardineCount, currentSardineHalfCount) {
 3        if(currentSardineCount > 1 && currentSardineCount < 15) {
 4              if(currentSardineCount % 2 == 1) {
 5                    currentSardineHalfCount--;
 6              }
 7              for(var i = 0; i < currentSardineHalfCount; i++) {
 8                    if(currentSardineCount + i > 14) {
 9                          break;
10                    }
11                    this.addSardine();
12              }
13        }
14  }
```

Na linha 3 é verificado se a quantidade de sardinhas é maior que um, pois considera-se que as sardinhas se reproduzem em duplas e se a quantidade de sardinhas é menor que 15, pois limita-se a quantidade de procriação para não prejudicar a performance da aplicação. Levando em consideração que elas se reproduzem em duplas, a quantidade de sardinhas que nascem é a metade da quantidade existente. Por exemplo, se existem quatro será gerado duas novas sardinhas e se existirem cinco também será gerado duas novas sardinhas, pois uma não possui par. No Quadro 60 pode ser visto a implementação da função `verifySharkProcreation`, responsável por verificar a procriação dos tubarões.

Quadro 60 - Implementação da função `verifySharkProcreation` da classe `AquariumObject`

```
1  AquariumObject.prototype.verifySharkProcreation = function() {
2        var currentSharkCount = Game.apiHandler.sharkCount;
3        if(currentSharkCount > 1 && currentSharkCount < 4 &&
4  Game.apiHandler.sardineEatenCount >= 5) {
5              this.addShark()
6              Game.apiHandler.sardineEatenCount = 0;
7        }
8  }
```

O funcionamento da procriação de tubarões é mais simples, porém funciona de forma semelhante à procriação das sardinhas. A quantidade de tubarões é limitada em quatro (linha 3), esse valor foi adotado com base em testes de utilização, dos quais verificou-se que esse valor gera um bom equilíbrio no ecossistema, levando em consideração o limite de quinze sardinhas. A procriação se efetua quando os tubarões tenham comido cinco ou mais sardinhas, então um novo tubarão nasce e a contagem de sardinhas devoradas é zerada. No Quadro 61 pode ser visto um exemplo de mensagem enviada para o Reasoner.

Quadro 61 - Exemplo de mensagem enviada para o Reasoner

```
1  {
2        "action" : "perception",
3        "name" : "Tubarão 1",
4        "perceptions" : ["onPercept(\"Tubarão 1\",\"Sardinha
5  1\",\"Sardinha\")"]
6  }
```

Conforme visto nas linhas 2 a 5, a mensagem é composta pelo identificador da ação que deve ser executada pelo Reasoner (*tag* `action`), o identificador do objeto que realizou a percepção (*tag* `name`) e uma lista de mensagens com as percepções identificadas (*tag* `perceptions`). Após realizar um ciclo de raciocínio, caso o Reasoner determine alguma ação a ser executada pelo objeto na simulação, é criada uma mensagem contendo a descrição da ação e posteriormente enviada para a aplicação via *WebSocket*. Um exemplo de mensagem enviada pelo Reasoner pode ser observado no Quadro 62.

Quadro 62 - Exemplo de mensagem enviada pelo Reasoner

```
1  {
2        "action" : "pursue(\"Tubarão 1\",\"Sardinha 1\")"
3  }
```

Na mensagem acima pode-se observar que existe somente um identificador (*tag* `action`), representando a ação que o objeto deve executar na simulação, nesse caso a ação é `pursue`, ou seja o Tubarão 1 deve perseguir a Sardinha 1.

### 3.3.6 Operacionalidade da implementação

O primeiro passo para se ter a simulação de um aquário com peixes é a inclusão do aquário, para isso é necessário arrastar a peça `Aquário` contida na aba `Fábrica de Peças`

para o engate correspondente na árvore de peças, logo abaixo da peça `Mundo`, conforme pode ser visto na Figura 29.

Figura 29 - Inclusão do aquário


O segundo passo é a inclusão de peixes, para isso é necessário arrastar ou a peça `Tubarão` que representa o predador ou a peça `Sardinha` que representa a presa, também contidas na aba `Fábrica de Peças`, para o engate correspondente na árvore de peças, conforme pode ser visto na Figura 30.

Figura 30 - Inclusão de peixes no aquário


Para remover um peixe do aquário deve-se arrastar a peça correspondente a ele na árvore de peças para a lixeira, conforme pode ser visto na Figura 31.

Figura 31 - Remoção de um peixe


Para remover o aquário é o mesmo procedimento acima, porém com a peça `Aquário`, todas as peças de peixes que estão incluídas no aquário também serão removidas. Para habilitar a câmera secundária responsável por exibir a visão do peixe deve-se clicar na peça correspondente ao peixe na árvore de peças, isso faz com que a peça seja selecionada e a câmera apareça no canto inferior direito, conforme observa-se na Figura 32.

Figura 32 - Habilitação da câmera secundária


A câmera secundária por padrão mostra uma visão mais amigável aos olhos humanos, com ângulo de visão e alcance semelhantes a visão humana. Porém, pode-se habilitar a visão "real" do peixe, ou seja, como o peixe visualiza o ambiente ao seu redor. Essa visão está relacionada diretamente com as propriedades `Raio da visão` e `Alcance da visão` contidas

na aba `Propriedades da Peça`. Para habilitar essa visão “real”, pode-se ativar a propriedade `Visão sincronizada`, conforme mostra a Figura 33.

Figura 33 - Habilitação da visão real do peixe


Percebe-se que o fundo da visão ficou azul e que não aparece mais a extremidade do aquário e nem a paisagem ao seu redor, isso se dá pelo fato de que o alcance da visão não chega até a extremidade do aquário e nem ao fundo panorâmico. A visão também pode ficar distorcida dependendo do valor do raio de visão e a qualquer momento pode-se voltar à visão amigável desabilitando a propriedade `Visão sincronizada`. Para desabilitar a câmera secundária basta cancelar a seleção do peixe atual, clicando novamente em cima da peça, como mostrado na Figura 34.

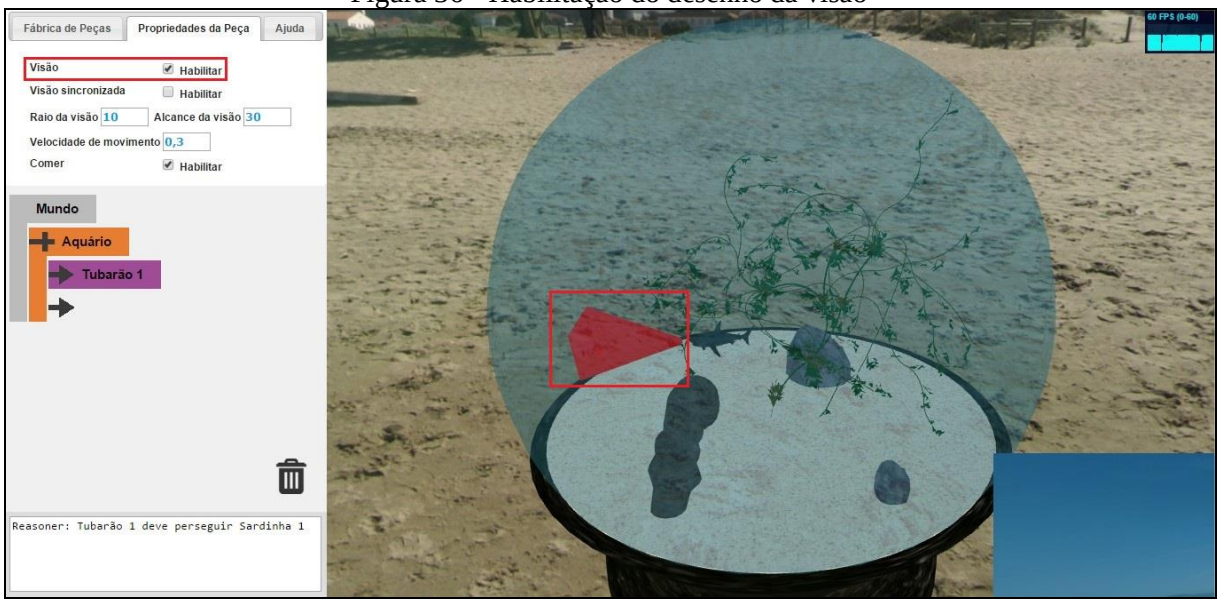
Figura 34 - Desativação da câmera secundária

Percebe-se que a câmera no canto inferior direito sumiu, além da área de texto voltar a ficar em branco. Ao selecionar o peixe, igual feito para habilitar a câmera secundária, também é habilitado a área de texto responsável por informar as ações determinadas a esse peixe pelo Reasoner, conforme pode ser visto na Figura 35.

Figura 35 - Habilitação da área de texto

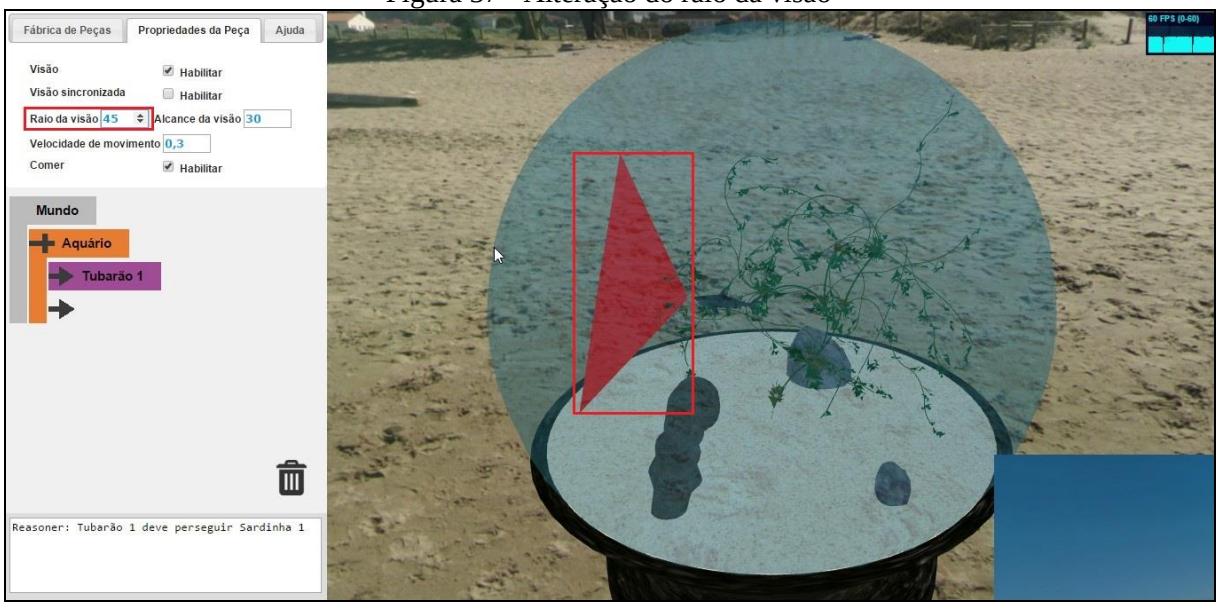

Percebe-se que há uma mensagem indicando que o `Tubarão 1` deve perseguir a `Sardinha 1`, mas se olharmos na árvore de peças e no aquário, há somente o `Tubarão 1`. Isso parece estranho porque a área de texto vai armazenando todas as ações enviadas pelo Reasoner para esse peixe, essa em especial ocorreu antes de remover a `Sardinha 1`. A aplicação permite habilitar o desenho do objeto correspondente à visão do peixe, para isso deve-se habilitar a propriedade `Visão` na aba `Propriedades da Peça`, conforme mostrado na Figura 36.

Figura 36 - Habilitação do desenho da visão



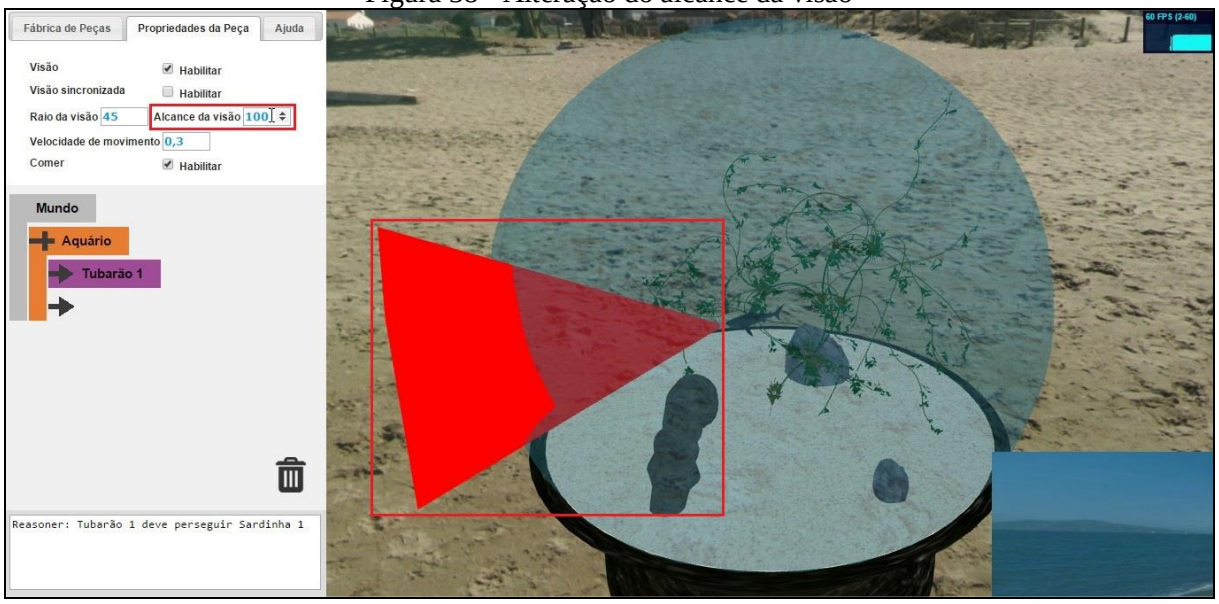

Nota-se que um objeto de cor vermelha e forma parecida com um cone foi desenhado a frente do tubarão, esse objeto está relacionado com as propriedades `Raio da visão` e `Alcance da visão` e representa a visão "real" do peixe na câmera secundária. Para alterar o raio de sua visão deve-se alterar a propriedade `Raio da visão` e apertar a tecla *enter* para confirmar a alteração, conforme visto na Figura 37.

Figura 37 - Alteração do raio da visão




Percebe-se que foi inserido o valor 45 e a área de visão aumentou, mas pode-se também alterar o alcance dessa área de visão, para isso é preciso alterar a propriedade `Alcance da visão`, conforme pode ser visto na Figura 38.

Figura 38 - Alteração do alcance da visão




Percebe-se que foi inserido o valor 100 e o alcance aumentou. A aplicação também permite a alteração da velocidade de movimento individual para cada peixe, para isso deve-se alterar a propriedade `Velocidade de movimento` do peixe selecionado, como mostra a Figura 39.

Figura 39 - Alteração da velocidade de movimento individual de um peixe


Nota-se que aumentou-se a velocidade de 0.3 para 0.6, isso significa que a velocidade do tubarão dobrou, pois esse valor representa o valor de deslocação no eixo z para cada quadro. A aplicação também permite controlar se o tubarão pode ou não comer as sardinhas, para isso pode-se alterar a propriedade `Comer`, conforme mostra a Figura 40.

Figura 40 - Alteração da propriedade `Comer`


Por padrão essa propriedade já vem habilitada, ou seja, os tubarões podem comer as sardinhas, fazendo com que elas sejam removidas do aquário e também da árvore de peças. Em alguns casos opta-se por desabilitar para que a população do aquário não diminua e que fique mais fácil de perceber comportamentos como o de perseguir. Somente peixes predadores possuem essa opção, nesse caso somente o tubarão. O restante das propriedades são comuns entre predadores e presas. A peça `Mundo` permite que seja alterada a velocidade de todos os peixes do aquário através da propriedade `Multiplicador de velocidade`, mas para poder visualizá-la, a peça mundo deve estar selecionada e consequentemente a aba `Propriedades da Peça`, conforme pode ser visto na Figura 41.

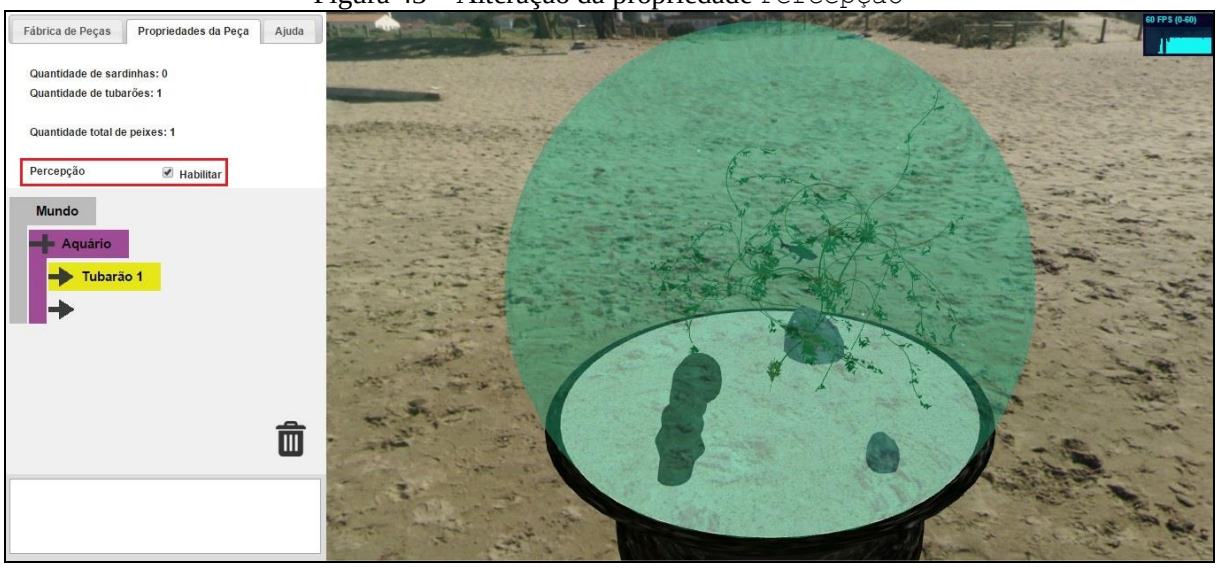
Figura 41 - Alteração da velocidade de movimento de todos os peixes

Percebe-se que o multiplicador por padrão possui o valor 1, esse valor é multiplicado pela velocidade individual de cada peixe, portanto uma velocidade de 0.3 multiplicada por 1 gera a velocidade 0.3, não alterando a mesma. Para dobrar a velocidade de movimento de todos os peixes é necessário alterar o valor do multiplicador de movimento para 2, pois 0.3 multiplicado por 2 gera uma velocidade de 0.6. Esse valor também pode ser 0 ou negativo, quando o valor é 0 os peixes param de se movimentar e quando o valor é negativo a animação começa a executar de trás para frente. A aplicação fornece uma maneira fácil de visualizar a quantidade de peixes no aquário, conforme mostra a Figura 42.

Figura 42 - Visualização da quantidade de peixes


Ela exibe a quantidade de peixes por tipo e o total, para isso deve-se selecionar a peça `Aquário` e consequentemente a aba `Propriedades da Peça`. Ainda nas propriedades da peça `Aquário` observa-se a propriedade `Percepção`, conforme pode ser visto na Figura 43.

Figura 43 – Alteração da propriedade `Percepção`



A propriedade `Percepção` é responsável por ativar ou não a percepção dos peixes, ou seja, se a percepção estiver ativa os peixes irão perceber o ambiente a sua volta, caso contrário só ficarão nadando sem nenhuma inteligência.

## 3.4 RESULTADOS E DISCUSSÕES

O objetivo principal de criar um aquário virtual que simulasse um ecossistema marinho através de animação comportamental foi contemplado, simulando uma cadeia alimentar pequena que possui o tubarão como predador e a sardinha como presa. O objetivo específico de estender o trabalho de Feltrin (2014) também foi contemplado. Utilizou-se ele como base para o desenvolvimento de uma nova versão do Reasoner, que se adapta melhor a simulações com animação em tempo real e que necessita de somente uma instância de *WebSocket*, enquanto que a versão antiga instanciava um para cada agente. Com uma única instância de *WebSocket* deixou-se de gastar recursos referentes a sua instanciação para cada ser virtual e não se notou aumento considerável no tempo médio de resposta em comparação ao trabalho de Feltrin (2014).

Outro objetivo específico contemplado foi o de permitir a inserção de agentes dotados de representações gráficas. O Aquário Virtual permite a inclusão de tubarões e sardinhas tridimensionais, cada qual com seu tipo de mente, criando um agente correspondente no modelo BDI implementado pelo interpretador Jason.

Por fim, o último objetivo específico do trabalho, que é referente a inclusão de funcionalidades para permitir a geração de animação comportamental, também foi contemplado. O aquário virtual oferece um ambiente propício para a inclusão de tubarões e sardinhas, dos quais são animados de forma autônoma e que percebem o ambiente a sua volta, podendo gerar animações específicas para comportamentos como explorar, fugir, perseguir e comer.

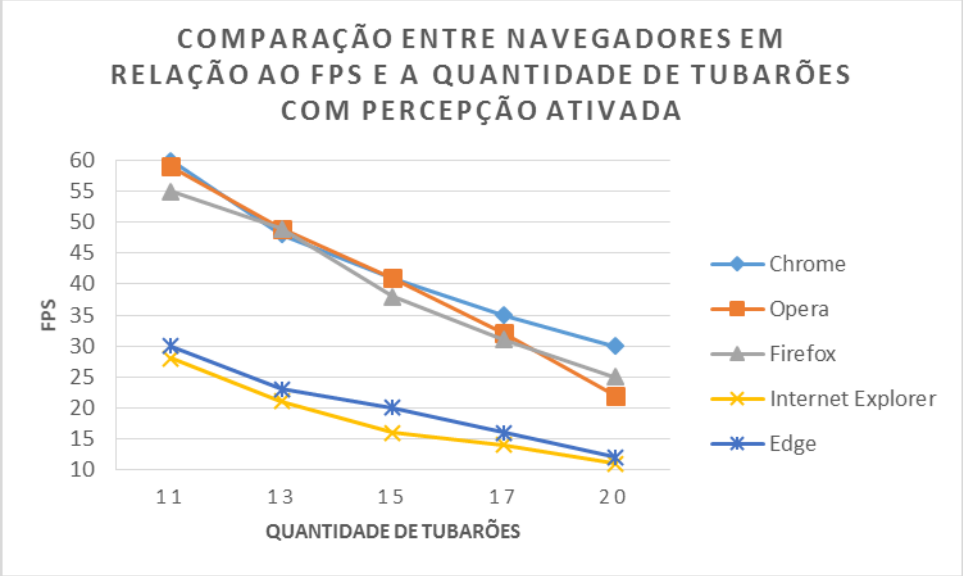
A ideia da aplicação mostrou-se interessante e promissora, pois foram relatados elogios por parte de acadêmicos do curso de Ciências Biológicas da FURB, conforme Apêndice A e também de um dos desenvolvedores do interpretador Jason, conforme Apêndice B. Com relação à propriedade referente ao multiplicador de velocidade presente na peça `Mundo`, o valor negativo não representa o comportamento de retroceder a simulação e sim a inversão do movimento das animações.

### 3.4.1 Teste de desempenho

O principal teste de desempenho a ser executado está relacionado com a quantidade de quadros por segundo (*Frames Per Second* - FPS) que a aplicação executa e para tal utilizou-se as estatísticas da biblioteca ThreeJS. Os testes aqui demonstrados são baseados nos dois tipos de peixe que podem ser adicionados ao aquário virtual, sendo que o tubarão possui 976 vértices e 1.687 faces enquanto que a sardinha possui 370 vértices e 632 faces. Percebe-se que são objetos tridimensionais relativamente "pesados" para animações desse tipo.

O primeiro teste consiste em executar a aplicação nos navegadores suportados, com algumas quantidades de tubarões que variam de 60 a 30 FPS no navegador Chrome. Escolheu-se o Chrome por ser o navegador utilizado durante o desenvolvimento. Na Figura 44 pode ser visto o gráfico comparativo entre eles.

Figura 44 - Gráfico comparativo de navegadores em relação ao FPS e a quantidade de tubarões com percepção ativada

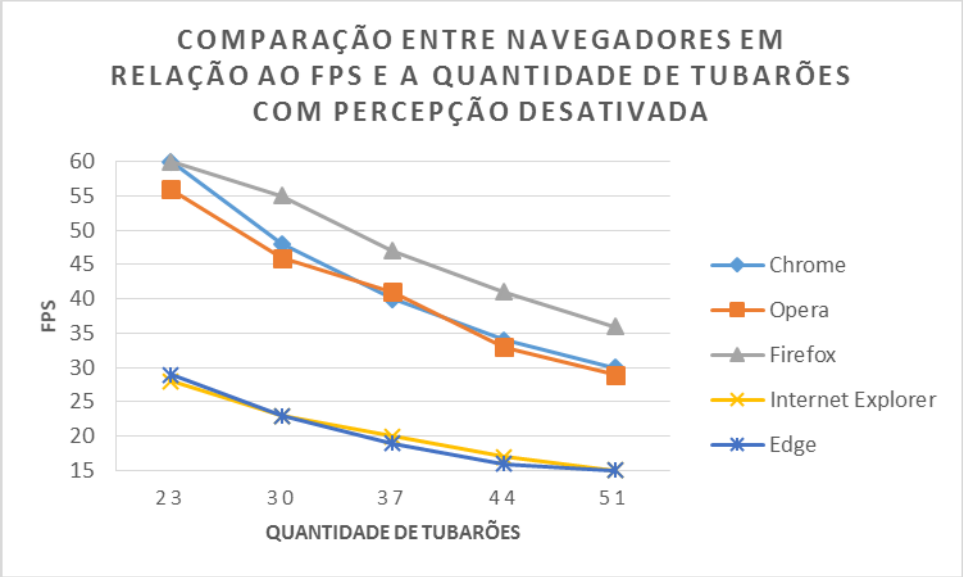



Nota-se que os navegadores Chrome, Firefox e Opera possuem os melhores desempenhos, mantendo-se entre 60 e 20 FPS com as quantidades de tubarões estipuladas, com o Chrome mostrando ser mais estável e possuir melhores resultados. Os navegadores Edge e Internet Explorer possuem os piores resultados, enquanto que os outros três conseguem executar a animação de onze tubarões dotados de percepção em torno de 60 FPS, esses dois mal conseguem manter em torno de 30 FPS, sendo que o pior resultado foi do Internet Explorer.

O segundo teste foi executado da mesma forma que o primeiro, porém dessa vez desabilitou-se a percepção, pois junto com ela encontram-se as chamadas a função que executa a detecção de colisão por *bounding box* e acredita-se que ela seja a principal

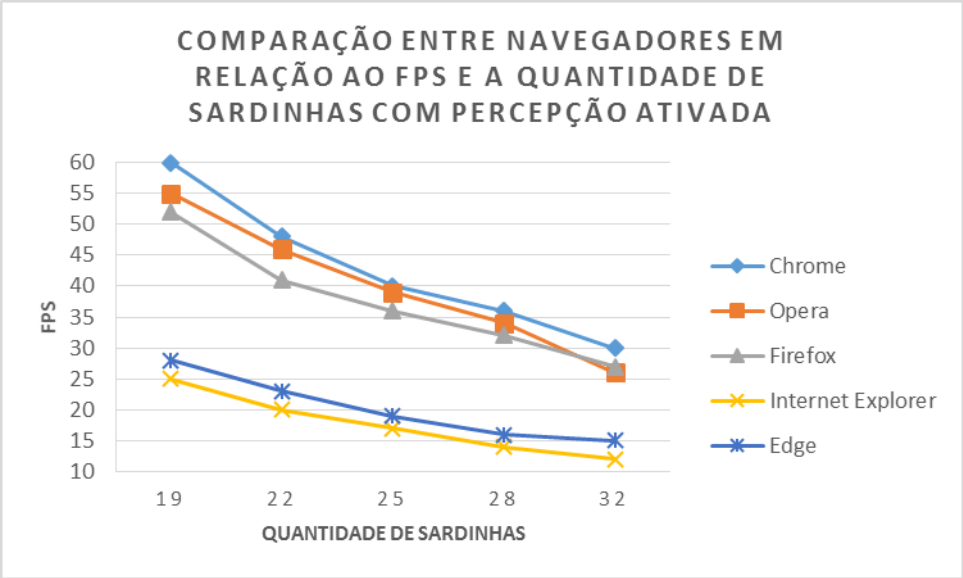
responsável pela perda de desempenho. Na Figura 45 pode ser visto o novo gráfico comparativo.

Figura 45 - Gráfico comparativo de navegadores em relação ao FPS e a quantidade de tubarões com percepção desativada



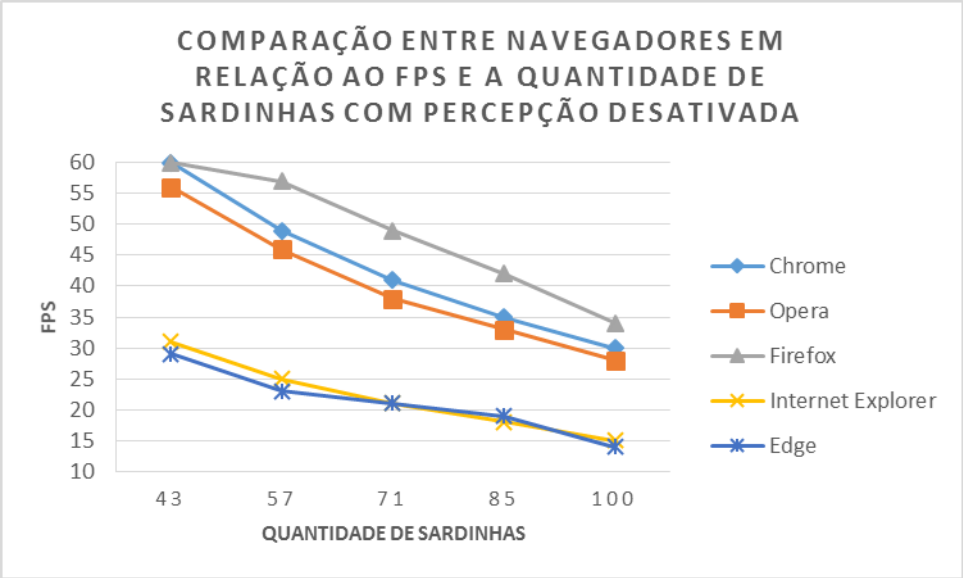

Percebe-se que o desempenho da aplicação melhorou muito, permitindo mais que o dobro de tubarões. Em relação aos navegadores, o comportamento geral não mudou muito, com exceção do Firefox que demonstrou um desempenho superior em relação aos demais e em relação a si mesmo com a percepção ativada. O terceiro teste foi executado da mesma forma que os anteriores, porém ele trata da quantidade de sardinhas com a perpção ativada. Seu gráfico comparativo pode ser visto na Figura 46.

Figura 46 - Gráfico comparativo de navegadores em relação ao FPS e a quantidade de sardinhas com percepção ativada



Percebe-se que novamente os navegadores Chrome, Firefox e Opera se mostram superiores, com destaque para o Chrome. Entre os piores continuam o Edge e o Internet Explorer, nota-se que eles estão com os resultados praticamente duas vezes piores que os demais. O quarto e último teste de FPS acontece da mesma forma que os anteriores e trata da quantidade de sardinhas com a percepção desativada, seu gráfico comparativo pode ser visto na Figura 47.

Figura 47 - Gráfico comparativo de navegadores em relação ao FPS e a quantidade de sardinhas com percepção desativada




Novamente o navegador Firefox demonstrou um comportamento diferenciado e superior aos demais com a percepção desativada. Percebe-se que a quantidade total de sardinhas mais que triplicou em relação a sua quantidade com a percepção ativada, reforçando a suspeita de que a detecção de colisão via *bounding box* utilizada demonstra muita perda de desempenho.

Foi feito também um teste de consumo de memória somente para o navegador Chrome, utilizando sua própria ferramenta chamada Profiles. Esse teste aborda o consumo de memória em *Megabytes* (MB) com a inclusão de tubarões e sardinhas. A tabela contendo os resultados de consumo de memória para a inclusão de tubarões pode ser visto na Tabela 1.

Tabela 1 - Quantidade de tubarões adicionados e seus respectivos valores de memória consumida

| quantidade de tubarões | memória consumida (MB) |
|---|---|
| 0 | 23.1 |
| 2 | 26.9 |
| 4 | 30.0 |
| 6 | 33.0 |
| 8 | 35.9 |
| 10 | 38.9 |

Levando em consideração que a aplicação com o aquário vazio consome 23.1 MB de memória, que foram incluídos 10 tubarões e a memória consumida aumentou em 15.8 MB. Divide-se essa quantidade de memória encontrada pelos 10 tubarões e pode-se chegar a uma estimativa de que cada tubarão incluído aumenta a memória consumida em 1.58 MB. Através dessa afirmativa chega-se a fórmula Mc = Qt x 1.58 + 23.1, onde Mc representa o total de memória consumida e o Qt a quantidade de tubarões. A tabela contendo os resultados de consumo de memória para a inclusão de sardinhas pode ser visto na Tabela 2.

Tabela 2 - Quantidade de sardinhas adicionadas e seus respectivos valores de memória consumida

| quantidade de sardinhas | memória consumida (MB) |
|---|---|
| 0 | 23.1 |
| 2 | 24.6 |
| 4 | 26.2 |
| 6 | 27.3 |
| 8 | 28.3 |
| 10 | 29.4 |

Levando em consideração novamente que a aplicação com o aquário vazio consome 23.1 MB de memória, que foram adicionadas 10 sardinhas e a memória consumida aumentou em 6.3 MB. Divide-se essa quantidade de memória encontrada pelas 10 sardinhas e pode-se chegar a uma estimativa de que cada sardinha inserida aumenta a memória consumida em 0.63 MB. Através dessa afirmativa chega-se a fórmula Mc = Qs x 0.63 + 23.1, onde Mc representa o total de memória consumida e o Qs a quantidade de sardinhas. Considerando as duas fórmulas é possível calcular o valor total de memória consumida para ambos os tipos de peixe em apenas uma fórmula, Mc = Qt x 1.58 + Qs x 0.63 + 23.1.

Para a realização dos testes optou-se por armazenar o Reasoner na máquina local, com o intuito de isolar interferências externas do canal de comunicação, focando na diferença apresentada pelos navegadores. Foi medido o tempo de conexão que a aplicação web leva para se conectar com o Reasoner via *WebSocket* para cada navegador utilizado, esse tempo envolve a criação do *WebSocket* e o envio e recebimento da mensagem de *hand shake* (ver Quadro 20). A comparação do tempo de conexão com o Reasoner entre os navegadores pode ser visto no Quadro 63.

Quadro 63 - Comparação do tempo de conexão com o Reasoner entre os navegadores

| navegador | Opera | Internet Explorer | Chrome | Edge | Firefox |
|---|---|---|---|---|---|
| tempo | 0.171 | 0.212 | 0.239 | 0.288 | 0.291 |

Nota-se que o navegador Opera é o mais rápido para estabilizar a conexão com o Reasoner, seguido do Internet Explorer, do Chrome, do Edge e o mais lento que é o Firefox. Além do tempo para estabilizar a conexão com o Reasoner foi coletado o tempo médio de raciocínio em segundos para cada navegador utilizado, conforme pode ser visto na Tabela 3.

Tabela 3 - Tempo médio de raciocínio por quantidade de peixes

| quantidade de peixes | Opera | Firefox | Chrome | Internet Explorer | Edge |
|---|---|---|---|---|---|
| 3 | 0.004 | 0.004 | 0.004 | 0.009 | 0.007 |
| 6 | 0.004 | 0.006 | 0.004 | 0.017 | 0.022 |
| 9 | 0.004 | 0.015 | 0.003 | 0.041 | 0.057 |
| 12 | 0.009 | 0.033 | 0.008 | 0.095 | 10.24 |
| 15 | 0.1 | 0.056 | 0.114 | 0.533 | 19.677 |

O tempo de raciocínio engloba o envio da mensagem via *WebSocket* para o Reasoner, seu raciocínio e a devolução da mensagem com a ação a ser tomada também via *WebSocket*. A quantidade de peixes inclui os dois tipos e possui a mesma quantidade de agentes no modelo BDI interpretado pelo Jason. Cada amostra teve um tubarão e duas sardinhas adicionadas no aquário, sendo coletadas cinco amostras para o cálculo da média.

Nota-se que o navegador Opera segue com o melhor resultado, seguido do Firefox que teve o pior desempenho em relação ao tempo para estabilizar a conexão. O Chrome mantém sua terceira posição, o Internet Explorer que teve o segundo melhor tempo para estabilizar a conexão teve somente o quarto melhor resultado nesse teste. Por fim o Edge apresentou um resultado muito ruim a partir de 12 peixes, com tempo acima de 10 segundos.

### 3.4.2 Comparativo entre o trabalho desenvolvido e seus correlatos

Nessa seção será apresentada uma comparação entre as principais características do trabalho desenvolvido com as características identificadas dos trabalhos correlatos, conforme pode ser visto no Quadro 64.

Quadro 64 – Comparação com os trabalhos correlatos

| aspectos/trabalhos | MASSIVE | Fish School and Obstacles | STEVE | VISEDU – Aquário Virtual |
|---|---|---|---|---|
| gera animação comportamental | X | X | X | X |
| possui ambiente tridimensional | X | | X | X |
| desenvolvido para a web | | X | | X |
| possui fins educativos | | | X | X |
| relacionado à biologia | | X | | X |
| permite interação com o ambiente | X | X | X | X |
| possui tipos variados de mente | X | X | | X |
| desacoplado do modelo de IA | | | X | X |
| gera comportamento imprevisível | X | X | | X |

A característica principal do MASSIVE" é gerar animação comportamental para multidões, permitindo a criação de agentes para um ambiente tridimensional e que interajam com o ambiente a sua volta. Ele também permite a criação de diversos tipos de mentes e seu raciocínio pode produzir comportamentos imprevisíveis. Comparado com o trabalho desenvolvido, percebe-se que suas características são semelhantes em relação a animação comportamental, apesar de seus objetivos serem diferentes. No quesito performance o MASSIVE se mostra muito superior.

O Fish School and Obstacles é o que mais se assemelha ao trabalho desenvolvido, pois trabalha com o conceito de animação comportamental e foi projetado para funcionar no ambiente web. É relacionado a biologia, mais precisamente a simulação de peixes e permite que seus peixes interajam com o ambiente. Possui mais de um tipo de mente e permite a geração de animações imprevisíveis. O trabalho desenvolvido aparenta ser superior, levando em consideração o ambiente mais realístico e por ter uma quantidade de comportamentos maior.

A maior semelhança entre o STEVE e o trabalho proposto é seu objetivo de auxiliar na educação das pessoas. Apesar de já ser um trabalho relativamente antigo, ele demonstra visualização gráfica bonita e um ambiente tridimensional. Outro ponto interessante é que ele mescla a interação com as pessoas e com o ambiente para gerar animações e tornar o estudo das pessoas mais atrativo. O trabalho desenvolvido aparenta ser mais atrativo para os alunos, pois possui um ambiente tridimensional mais realístico e gera animações imprevisíveis.

# 4 CONCLUSÕES

O objetivo principal de criar um aquário virtual que simulasse um ecossistema marinho mesmo que de forma mínima através de animação comportamental foi atendido, permitindo a simulação de uma cadeia alimentar pequena que possui o tubarão como predador e a sardinha como presa, que por sua vez se alimenta de plâncton. A aplicação se mostrou um ótimo ambiente para a inserção de agentes dotados de representação gráfica, possibilitando a geração de comportamentos específicos como explorar, perseguir, fugir e comer, sem a necessidade de um animador ficar especificando como eles devem ocorrer.

As ferramentas utilizadas pela aplicação web se mostraram adequadas, a combinação do HTML5 com o Javascript possibilitaram o dinamismo que um simulador necessita. O Motor de Jogos desenvolvido por Harbs (2013) e estendido por Koehler (2015) disponibilizou a estrutura básica para o desenvolvimento desse trabalho, abstraindo muitas funções e gerando mais produtividade. A utilização da biblioteca gráfica ThreeJS se mostrou eficiente, permitindo que a representação gráfica do ambiente e dos agentes fosse próxima da realidade e sem deixar a desejar em relação ao desempenho. Embora a detecção de colisão via *bounding box* ter se mostrado uma rotina com alto consumo de processamento, há outras maneiras mais performáticas de implementar a detecção de colisão, não tirando assim os demais méritos da biblioteca.

As ferramentas utilizadas pela aplicação servidora também se mostraram adequadas, o interpretador Jason (2014b) faz bem seu papel de interpretar o modelo BDI, permitindo a implementação de crenças, objetivos e planos através da linguagem AgentSpeak, além de permitir a integração com a linguagem Java. O Reasoner estendido de Feltrin (2014) se mostrou funcional, porém sua utilização preocupa em casos que trabalham com muitos agentes, devido ao tempo de raciocínio aumentar consideravelmente com o aumento do número de agentes, podendo gerar atrasos perceptíveis em simulações em tempo real.

Acredita-se que as principais contribuições desse trabalho são disponibilizar um ambiente tridimensional que permita a inclusão de agentes dotados de inteligência artificial e representação gráfica, possibilitar um leque enorme de extensões e possibilitar a execução de diversas medições de desempenho em relação a esses agentes. Suas principais limitações estão na quantidade de peixes dotados de percepção inseridos no aquário e no tempo médio de raciocínio para muitos agentes. Caso deseja-se fazer uma simulação com muitos peixes a aplicação apresenta muita lentidão, referente a baixa taxa de FPS e ao atraso no raciocínio, fazendo com que as ações não sejam executadas em tempo real.

## 4.1 EXTENSÕES

São sugeridas as seguintes extensões para a continuidade do trabalho:

a) criar modelos mentais mais elaborados;
b) utilizar outras técnicas de IA para interpretação do raciocínio no Reasoner, permitindo a comparação de diferentes técnicas de IA em um mesmo ambiente;
c) integrar o Reasoner com outros interpretadores do modelo BDI, permitindo a comparação de desempenho entre eles;
d) utilizar outras técnicas para detectar colisão, diminuindo o custo da detecção via *bounding box* e permitindo a comparação de desempenho entre elas. Uma sugestão é a detecção de colisão via *Ray Caster*;
e) utilizar outras bibliotecas gráficas, permitindo a comparação de resultados entre elas;
f) utilizar ferramentas voltadas para a web que permitam o desenvolvimento do aquário virtual de forma concorrente, permitindo a comparação de desempenho com a implementação atual;
g) implementar animações mais reais, como a animação das nadadeiras ao se movimentar e a animação da boca ao comer;
h) adicionar mais formas de vida e interação ao ecossistema, como outras cadeias alimentares;
i) implementar ou integrar com um modelo BDI em Javascript, caso exista, evitando a troca de mensagens pela rede com o Reasoner;
j) incluir um aquário de água doce e implementar seu ecossistema, possibilitando trabalhar com os dois tipos no mesmo ambiente;
k) implementar comportamentos cooperativos e competitivos entre os seres vivos do aquário, como a formação de cardumes por parte das presas, com o intuito de se defenderem ou a cooperação entre os predadores para conseguirem mais alimentos. Pode-se até mesmo desenvolver um comportamento cooperativo e um competitivo de caça, possibilitando a comparação de qual é mais eficiente;
l) gamificar o aquário, ou seja, trazer aspectos de jogos, como por exemplo executar determinadas fases para ganhar pontos, aumentar de nível, liberar novos desafios, novos seres vivos, entre outros;
m) seguindo a linha de gamificação, poder-se-ia implementar uma espécie de jogo multi-jogador que os alunos pudessem aplicar seus conhecimentos em biologia,

para que depois de um determinado tempo eles pudessem comparar seu desempenho com o dos outros colegas, servindo até mesmo como uma forma de avaliação;

n) implementar novas formas de interação com o aquário, como novas propriedades. Uma sugestão é o controle de oxigênio da água, caso o nível de oxigênio estiver muito baixo ou alto, os peixes começam a apresentar problemas e até mesmo morrer.

# REFERÊNCIAS


DAJOZ, Roger. **Princípios de ecologia**. 7. Ed. Porto Alegre: Artmed, 2005.

FELTRIN, Gustavo R. **VISEDU-SIMULA 1.0**: Visualizador de material educacional, módulo de animação comportamental. 2014. 90 f. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

GRUPO DE PESQUISA EM COMPUTAÇÃO GRÁFICA, PROCESSAMENTO DE IMAGENS E ENTRETENIMENTO DIGITAL. **VISEDU**: Visualizador de material educacional. Blumenau, [200-?]. Disponível em: <www.inf.furb.br/gcg/visedu>. Acesso em: 31 mar. 2015.

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.

INFORMATION SCIENCES INSTITUTE. **Virtual Environments for Training**. Los Angeles, 2000. Disponível em: <http://www.isi.edu/isd/VET/vet.html>. Acesso em: 8 abr. 2015.

JASON. **Description**. [S.l.], 2014a. Disponível em: <http://jason.sourceforge.net/wp/description/>. Acesso em: 7 abr. 2015.

_____. **About Jason**. [S.l.], 2014b. Disponível em: <http://jason.sourceforge.net/wp/>. Acesso em: 7 abr. 2015.

KOEHLER, William F. **VISEDU-CG 4.0**: Visualizador de material educacional. 2015. 89 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

MAGNENAT-THALMANN, Nadia; THALMANN, Daniel. **Handbook of virtual humans**. Chichester: John Wiley & Sons, 2004.

MASSIVE. **Frequently asked questions.** [S.l.], 2014a. Disponível em: <http://massivesoftware.com/faq.html>. Acesso em: 6 abr. 2015.

_____. **About MASSIVE**. [S.l.], 2014b. Disponível em: <http://massivesoftware.com/about.html>. Acesso em: 6 abr. 2015.

_____. **Products**. [S.l], 2014c. Disponível em: <http://massivesoftware.com/products.html>. Acesso em: 8 abr. 2015.

_____. **Massive 8.0**. [S.l.], 2015. Disponível em: <http://massivesoftware.com/massive8.0-press-release.html>. Acesso em: 6 nov. 2015.

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: 1 abr. 2015.

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

OPENGL. **Scissor Test**. [S.l.], 2015a. Disponível em: <https://www.opengl.org/wiki/Scissor_Test>. Acesso em: 6 nov. 2015.

_____. **Shader**. [S.l.], 2015b. Disponível em: <https://www.opengl.org/wiki/Shader>. Acesso em: 6 nov. 2015.

_____. **Fragment Shader**. [S.l.], 2015c. Disponível em: <https://www.opengl.org/wiki/Fragment_Shader>. Acesso em: 6 nov. 2015.

_____. **Vertex Shader**. [S.l.], 2015d. Disponível em: <https://www.opengl.org/wiki/Vertex_Shader>. Acesso em: 6 nov. 2015.

PEREIRA, Ana M. **Tecnologia x Educação**. 2011. 44 f. Trabalho de Conclusão de Curso (Especialização em Docência do Ensino Superior) – Universidade Candido Mendes, Rio de Janeiro.

REYNOLDS, Craig. **Behavioral animation**. [S.l.], abr. 1997. Disponível em: <http://www.red3d.com/cwr/behave.html>. Acesso em: 8 abr. 2015.

RICKEL, Jeff; JOHNSON, W. Lewis. **STEVE**: A Pedagogical Agent for Virtual Reality. Marina del Rey, 1998. Disponível em: <http://www.isi.edu/isd/VET/agents98-distribution.pdf>. Acesso em: 2 abr. 2015.

ROSA, Thomas da. **Simulador de animais vivos**: Meios alternativos. 2008. 59 f. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SCHMICKL, Thomas. **Fish school and obstacles**. [S.l.], 2002. Disponível em: <http://zool33.uni-graz.at/schmickl/Self-organization/Group_behavior/Fish_school_and_obstacles/fish_school_and_obstacles.html>. Acesso em: 3 abr. 2015.

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

SHIMIZU, Tamio. **Simulação em computador digital**. São Paulo: Editora da Universidade de São Paulo, 1975.

SOUZA, Renata B. **O uso das tecnologias na educação**. [S.l.], [200-?] Disponível em: <https://www.grupoa.com.br/revista-patio/artigo/5945/o-uso-das-tecnologias-na-educacao.aspx>. Acesso em: 24 mar. 2015.

TAVARES, Emanuel. **Simulação de Ecossistema**. São Paulo, 2013. Disponível em: <http://www.computacaonatural.com.br/simulacao-de-ecossistema/>. Acesso em: 24 mar. 2015.

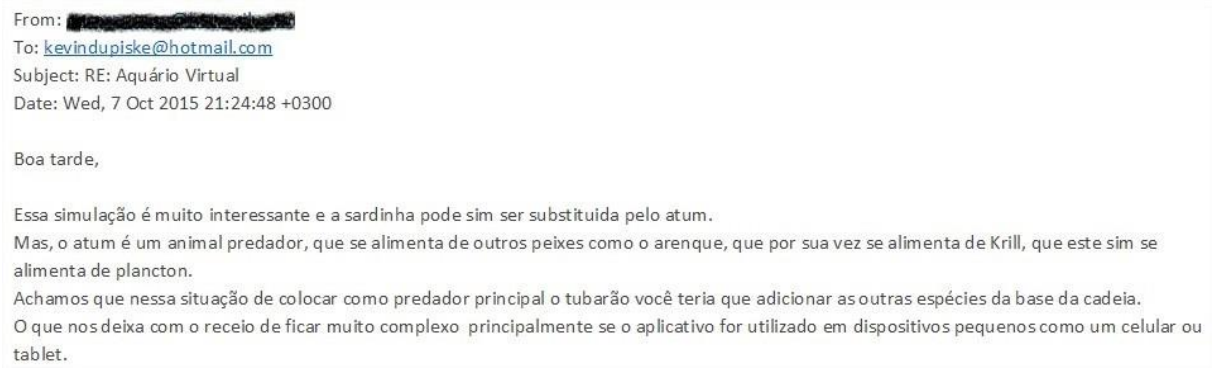
THREEJS. **ShaderMaterial**. [S.l.], 2015. Disponível em: <http://threejs.org/docs/#Reference/Materials/ShaderMaterial>. Acesso em: 6 nov. 2015.

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

## APÊNDICE A – Contato com acadêmicos de Ciências Biológicas da FURB

Durante o desenvolvimento da aplicação, entrou-se em contato com os acadêmicos Quirino Hugo Schmitz e Aurora Rupp do curso de Ciências Biológicas da FURB, sendo que o Quirino já trabalhou com aquários marinhos e a Aurora é uma entusiasta da biologia marinha. Eles sugeriram muitas coisas interessantes, porém o tempo não permitiu que todas fossem implementadas. Percebeu-se que eles viram com bons olhos uma aplicação com o intuito de simular um ecossistema de aquário marinho, conforme pode ser visto na Figura 48, que exibe um trecho de um dos e-mails trocados, onde a acadêmica Aurora diz que essa simulação é muito interessante, referindo-se à simulação de uma cadeia alimentar entre tubarão, sardinha e plâncton.

Figura 48 - Trecho de e-mail sobre a aplicação enviado por uma acadêmica de Ciências Biológicas



From: [redacted]
To: kevindupiske@hotmail.com
Subject: RE: Aquário Virtual
Date: Wed, 7 Oct 2015 21:24:48 +0300

Boa tarde,

Essa simulação é muito interessante e a sardinha pode sim ser substituida pelo atum.
Mas, o atum é um animal predador, que se alimenta de outros peixes como o arenque, que por sua vez se alimenta de Krill, que este sim se alimenta de plancton.
Achamos que nessa situação de colocar como predador principal o tubarão você teria que adicionar as outras espécies da base da cadeia.
O que nos deixa com o receio de ficar muito complexo principalmente se o aplicativo for utilizado em dispositivos pequenos como um celular ou tablet.

## APÊNDICE B – Contato com um dos desenvolvedores do interpretador Jason

Durante o desenvolvimento da aplicação, entrou-se em contato com um dos desenvolvedores do interpretador Jason, o Dr. Jomi Fred Hübner, professor da Universidade Federal de Santa Catarina (UFSC). Ele elogia a ideia da aplicação e responde alguns questionamentos sobre o uso do interpretador e sobre a implementação da mente dos agentes através da linguagem AgentSpeak, conforme pode ser visto na Figura 49.

Figura 49 – Trecho de e-mail trocado com um dos desenvolvedores do interpretador Jason