UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

MOTOR PARA JOGOS 2D UTILIZANDO HTML5

MARCOS HARBS

BLUMENAU
2013

2013/2-16

MARCOS HARBS

MOTOR PARA JOGOS 2D UTILIZANDO HTML5

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

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

BLUMENAU
2013

2013/2-16

MOTOR PARA JOGOS 2D UTILIZANDO HTML5

Por

MARCOS HARBS

Trabalho 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. Aurélio Faustino Hoppe, M.Sc – FURB

Membro: ____
Prof. Maurício Capobianco Lopes, Dr – FURB

Blumenau, 10 de dezembro de 2013

Dedico este trabalho a todos os meus familiares e amigos, assim como todos os educadores que ajudaram na formação de meu caráter.

AGRADECIMENTOS

Aos meus pais por terem me dado a oportunidade de estar vivo.

Aos meus amigos que me apoiaram.

À minha namorada pela paciência.

Ao meu orientador, Dalton Solano dos Reis, por ter acreditado na conclusão deste trabalho e me conduzido durante o percurso.

As convicções são inimigas mais perigosas da verdade do que as mentiras.

Friedrich Nietzsche

RESUMO

Este trabalho apresenta a implementação de um motor de jogos 2D em HTML5 e Javascript, um editor web para este motor e uma aplicação para exemplificar o uso destas duas implementações. O motor de jogos disponibiliza funcionalidades como criação de cena, criação de camadas, gerenciamento de objetos, gerenciamento de recurso de imagens e áudio, detecção de colisão, física de corpos rígidos (utilizando a biblioteca Box2DJS) e uma arquitetura reutilizável orientada a componentes. Para a criação do motor foi utilizada a linguagem Javascript juntamente com o elemento *canvas* do HTML5. Este trabalho também aborda técnicas para abstração da parte comum de um jogo e apresenta os resultados obtidos com os testes de desempenho e memória realizados através dos principais navegadores do mercado.

Palavras-chave: Motor de jogos. HTML5. Box2DJS. Javascript.

ABSTRACT

This work describes the implementation of a 2D game engine in HTML5 and Javascript, a web editor for this engine and an application that illustrate the use of these two implementations. The game engine provides functionalities such as creating scenes, creating layers, objects management, assets management, collision detection, rigid body physics (using the Box2DJS) and a reusable component oriented architecture. To create the engine was used the Javascript language with the HTML5 canvas element. This work addresses techniques to abstract the common part of developing a game and introduce the results of performances and memory tests that were executed in the major browsers.

Key-words: Game engine. HTML5. Box2DJS. Javascript.

LISTA DE ILUSTRAÇÕES

Figura 1 – Visão interna do Kinect 20

Figura 2 – Peças do Tangram 21

Figura 3 – Impact *game engine* 22

Figura 4 – Unity 3D 23

Quadro 1 – Características dos trabalhos correlatos 23

Figura 5 – Diagrama de casos de uso do motor de jogos 25

Figura 6 – Diagrama de pacotes do motor de jogos 26

Figura 7 – Diagrama de classes do pacote `component` 27

Figura 8 – Diagrama de classes do pacote `gameobject` 28

Figura 9 – Diagrama de classes do pacote `game` 29

Figura 10 – Diagrama de classes do pacote `system` 30

Figura 11 – Diagrama de classes do pacote `collide` 31

Figura 12 – Diagrama de classes do pacote `geometric` 31

Figura 13 – Diagrama de classes do pacote `utils` 31

Figura 14 – Diagrama de casos de uso do editor de jogos 32

Figura 15 – Diagrama de pacotes do editor de jogos 33

Figura 16 – Diagrama de classes do pacote `br.com.furb.html5.game.editor.controller` 34

Figura 17 – Diagrama de classes do pacote `br.com.furb.html5.game.editor.managedbean` 35

Figura 18 – Diagrama de classes do pacote `br.com.furb.html5.game.editor.model` 36

Figura 19 – Diagrama de classes do pacote `br.com.furb.html5.game.editor.utils` 37

Figura 20 – Diagrama de sequencia do *loop* principal do motor de jogos 38

Figura 21 – Camadas do editor e motor de jogos 39

Quadro 2 – Código fonte de `Game` 42

Quadro 3 – Código fonte de `Layer` 43

Quadro 4 – Implementações do método `createBodyShape` 44

Quadro 5 – Eventos do objeto `Component` 45

Quadro 6 – Normalização de coordenada do *mouse* 46
Quadro 7 – Exemplo de arquivo gerado pelo editor..48
Figura 22 – Tela inicial da ferramenta de edição 49
Figura 23 – Criação de um novo projeto 50
Figura 24 – Adicionando um novo objeto do jogo..50
Figura 25 – Criação de uma instância de um objeto do jogo 51
Figura 26 – Adicionando um novo componente 52
Figura 27 – Criação de uma nova instância de um componente 53
Figura 28 – Tela de criação de um componente..54
Quadro 8 – Código fonte de `MoveTangramComponent` 55
Figura 29 – Execução do jogo Tangram...56
Figura 30 – Tela para manutenção de *assets* 57
Figura 31 – Tela de documentação do motor 58
Figura 32 – Cenário de testes com o jogo Tangram e Kinect...60
Figura 33 – Cenário de testes com o jogo Space Invaders 61
Figura 34 – Cenário de testes de desempenho com 200 objetos 62
Figura 35 – Gráfico comparando o desempenho dos navegadores na primeira análise...........63
Figura 36 – Relatório de execução do *profile* da aplicação...64
Figura 37 – Gráfico comparando o desempenho dos navegadores na segunda análise 66
Figura 38 – Consumo de memória do motor de jogos 67
Quadro 9 – Características dos trabalhos correlatos e do trabalho desenvolvido 67
Quadro 10 – Caso de uso UC01 73
Quadro 11 – Caso de uso UC02 73
Quadro 12 – Caso de uso UC03 73
Quadro 13 – Caso de uso UC04 73
Quadro 14 – Caso de uso UC05 74
Quadro 15 – Caso de uso UC06 74
Quadro 16 – Caso de uso UC07 74
Quadro 17 – Caso de uso UC08 74
Quadro 18 – Caso de uso UC09 74
Quadro 19 – Caso de uso UC10 75
Quadro 20 – Caso de uso UC11 75
Quadro 21 – Caso de uso UC12 75
Quadro 22 – Caso de uso UC13 75

Quadro 23 – Caso de uso UC14 75
Quadro 24 – Caso de uso UC15 76
Quadro 25 – Caso de uso UC16 76
Quadro 26 – Caso de uso UC17 77
Quadro 27 – Caso de uso UC18 77
Quadro 28 – Caso de uso UC19 77

LISTA DE TABELAS

Tabela 1 – Quantidade de quadros por segundo por número de objetos na cena nos principais navegadores na primeira análise 62
Tabela 2 – Quantidade de quadros por segundo por número de objetos na cena nos principais navegadores na segunda análise 65

LISTA DE SIGLAS

CSS – *Cascading Style Sheets*

DOM – *Document Object Model*

FPS – *First Person Shooter*

HTML – *HyperText Markup Language*

HTML5 – *HyperText Markup Language 5*

JSF – *Java Server Faces*

JSON – *JavaScript Object Notation*

MVC – *Model View Controller*

POJO – *Plain Old Java Objects*

RF – Requisito Funcional

RNF – Requisito Não Funcional

UML – *Unified Modeling Language*

W3C – *World Wide Web Consortium*

WHATWG – *Web Hypertext Application Technology Working Group*

SUMÁRIO

1 INTRODUÇÃO.......... 14
1.1 OBJETIVOS DO TRABALHO.......... 15
1.2 ESTRUTURA DO TRABALHO.......... 15
2 FUNDAMENTAÇÃO TEÓRICA.......... 17
2.1 JOGOS ELETRÔNICOS.......... 17
2.2 MOTORES DE JOGOS.......... 17
2.3 TECNOLOGIAS USADAS NO DESENVOLVIMENTO DO MOTOR DE JOGOS.......... 18
2.3.1 HTML5.......... 18
2.3.2 JAVASCRIPT.......... 19
2.3.3 BOX2DJS.......... 19
2.3.4 KINECT.......... 19
2.4 TANGRAM.......... 20
2.5 TRABALHOS CORRELATOS.......... 21
2.5.1 Impact.......... 21
2.5.2 Unity 3D.......... 22
2.5.3 Comparação entre os trabalhos correlatos.......... 23
3 DESENVOLVIMENTO.......... 24
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO.......... 24
3.2 ESPECIFICAÇÃO.......... 24
3.2.1 Casos de uso do motor de jogos.......... 25
3.2.2 Diagramas de classes do motor de jogos.......... 25
3.2.2.1 Pacote `component`.......... 26
3.2.2.2 Pacote `gameobject`.......... 28
3.2.2.3 Pacote `game`.......... 28
3.2.2.4 Pacote `system`.......... 29
3.2.2.5 Pacote `collide`.......... 30
3.2.2.6 Pacote `geometric`.......... 31
3.2.2.7 Pacote `utils`.......... 31
3.2.3 Casos de uso do editor de jogos.......... 32
3.2.4 Diagramas de classes do editor de jogos.......... 32

3.2.4.1 Pacote `br.com.furb.html5.game.editor.controller` 33
3.2.4.2 Pacote `br.com.furb.html5.game.editor.managedbean` 34
3.2.4.3 Pacote `br.com.furb.html5.game.editor.model` 36
3.2.4.4 Pacote `br.com.furb.html5.game.editor.utils` 36
3.2.5 Diagramas de sequência 37
3.2.6 Diagramas de arquitetura 39
3.3 IMPLEMENTAÇÃO 40
3.3.1 Técnicas e ferramentas utilizadas 40
3.3.2 O motor de jogos 40
3.3.2.1 Arquitetura do motor de jogos 41
3.3.2.2 Estrutura do motor de jogos 42
3.3.3 O editor de jogos 47
3.3.4 Operacionalidade da implementação 48
3.4 RESULTADOS E DISCUSSÃO 58
3.4.1 Testes de funcionalidades 59
3.4.1.1 Tangram 59
3.4.1.2 Space Invaders 60
3.4.2 Testes de desempenho 61
3.4.3 Comparativo entre o trabalho desenvolvido e os correlatos 67
4 CONCLUSÕES 69
4.1 EXTENSÕES 70
REFERÊNCIAS BIBLIOGRÁFICAS 71
APÊNDICE A – Detalhamento dos casos de uso da aplicação 73

1 INTRODUÇÃO

No livro *Theory of Fun for Game Design* segundo Koster (2005 apud GREGORY, 2009, p. 8), um jogo é uma experiência interativa que fornece ao jogador uma sequência de desafios cada vez mais difíceis que promovem uma aprendizagem ao jogador. Na academia utiliza-se o termo teoria dos jogos, na qual múltiplos agentes selecionam estratégias e táticas a fim de maximizar seus ganhos, dentro de um conjunto de regras bem definidas do jogo. Quando usado no contexto de entretenimento baseado em computador, a palavra jogo normalmente nos remete a imagem de um mundo virtual tridimensional com um humanóide, animal, ou veículo como personagem principal sob controle do jogador (GREGORY, 2009, p. 8).

O termo motor de jogos surgiu em meados da década de 1990 em referência aos jogos de tiro em primeira pessoa (*First Person Shooter* - FPS) como o Doom. Com os jogos baseados em computador tornando-se populares os desenvolvedores começaram a montar a arquitetura do jogo de forma reutilizável, separando os componentes de software como o sistema de renderização de modelos tridimensionais, sistema de detecção de colisão e sistema de gerenciamento de áudio, das regras de negócio específicas do jogo. O valor desta separação logo se tornou evidente, quando os desenvolvedores começaram a reaproveitar esta arquitetura em diferentes produtos (GREGORY, 2009, p. 11).

Um motor de jogos normalmente é desenvolvido para um gênero específico de jogo. Existem vários gêneros de jogos, dentre eles pode-se citar: *First Person Shooter* (jogos de tiro em primeira pessoa como Doom), *Platformers* (jogos de aventura como Donkey Kong), *Fighting Games* (jogos de luta), *Racing Games* (jogos de corrida), *Real-Time Strategy* (jogos de estratégia em tempo real como Age of Empires), *Massively Multiplayer Online Games* (jogos *online* com grande quantidade de jogadores conectados simultaneamente como em World of Warcraft), entre outros (GREGORY, 2009, p. 13-24). Outro gênero de jogos são os educacionais, onde a pedagogia utilizada é a exploração auto dirigida ao invés da instrução explícita e direta. Um exemplo deste gênero de jogos é o Tangram, que possui uma grande variedade de representações geométricas, além de se poder formar letras. Este jogo é utilizado em algumas escolas para ensino de formas geométricas, polígonos e frações (OSCAR JÚNIOR, 2003, p. 30). Outra particularidade dos motores de jogos é que eles são implementados em diversas linguagens e com a ascensão da web hoje também é possível desenvolver jogos para web utilizando *HyperText Markup Language 5* (HTML5) (PEREIRA, 2012, p. 12).

O HTML5 é uma cooperação entre o *World Wide Web Consortium* (W3C) e a *Web Hypertext Application Technology Working Group* (WHATWG). O HTML5 possui algumas premissas, como: novos recursos devem ser baseados em *HyperText Markup Language* (HTML), *Cascading Style Sheets* (CSS), *Document Object Model* (DOM) e Javascript, reduzir a necessidade de *plug-ins* externos como Flash, melhor tratamento de erro, deve ser independente de dispositivo e o processo de desenvolvimento deve ser visível para o público (W3SCHOOLS, 2013a). O *canvas* fornece toda a potência gráfica necessária para implementar qualquer coisa, de processadores de texto até vídeo games (GEARY, 2012, p. xv).

Diante do exposto, desenvolveu-se um motor de jogos em HTML5 para criação de jogos em duas dimensões. Além do motor foi criada uma ferramenta de edição de jogo que irá utilizar o motor criado. Esta ferramenta funciona nos principais navegadores do mercado, permitindo que o usuário crie um jogo de forma mais visual e programe novas funcionalidades utilizando a linguagem Javascript.

1.1 OBJETIVOS DO TRABALHO

O objetivo do trabalho desenvolvido é criar um motor para facilitar a criação de jogos em duas dimensões utilizando HTML5 e Javascript, juntamente com uma ferramenta para edição de jogos que irá funcionar no navegador, permitindo a criação do jogo de forma visual por pessoas com pouco conhecimento de programação.

Os objetivos específicos do trabalho são;

a) disponibilizar um motor de jogos 2D;
b) disponibilizar um editor gráfico que funcione nos principais navegadores e que permita ao usuário alterar o jogo de forma visual;
c) utilizar o editor gráfico e o motor de jogos para desenvolver um jogo.

1.2 ESTRUTURA DO TRABALHO

O trabalho está desenvolvido em quatro capítulos. O segundo capítulo do presente trabalho refere-se à fundamentação teórica necessária para o seu entendimento.

O terceiro capítulo contém a descrição de como ocorreu o desenvolvimento do motor de jogos e do editor, os casos de uso envolvidos, os diagramas de classe, diagrama de sequência e documentações necessárias. Ainda, são apresentados os resultados e discussões ocorridas durante o desenvolvimento.

Ao final, o quarto capítulo apresenta as conclusões do presente trabalhando e sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Na seção 2.1 é apresentado o conceito de jogos eletônicos. A seção 2.2 trata sobre motores de jogos. A seção 2.3 apresenta as tecnologias usadas no desenvolvimento do motor de jogos. A seção 2.4 apresenta o jogo Tangram. Na última seção serão descritos alguns trabalhos correlatos.

2.1 JOGOS ELETRÔNICOS

O termo geral jogo abrange vários tipos de jogos como: jogos de tabuleiro, jogos de cartas, jogos de cassino, jogos de guerra, jogos de computador, entre vários outros tipos (GREGORY, 2009, p. 8).

Na maioria dos jogos de computador, um subconjunto do mundo real ou de um mundo imaginário é modelado matematicamente, de modo que este mundo possa ser manipulado pelo computador. Este modelo é uma simplificação do modelo real, em função da dificuldade em modelar todos os detalhes do mundo real para o mundo virtual. Esta aproximação e simplificação são ferramentas que um desenvolvedor de jogos possui, e quando usadas corretamente podem proporcionar ao jogador uma experiência única (GREGORY, 2009, p. 9).

Todos os jogos de computador interativos são simulações temporais, o que significa que o modelo do mundo virtual é dinâmico. O estado do mundo do jogo muda ao longo do tempo conforme os eventos e a história do jogo acontecem. O jogo também deve responder aos comandos de entrada do jogador em tempo real (GREGORY, 2009, p. 9).

2.2 MOTORES DE JOGOS

Um motor de jogos é uma abstração dos detalhes relacionados às tarefas comuns no desenvolvimento de jogos, como renderização, física e entrada de dados, permitindo aos desenvolvedores focar nos detalhes que tornam seus jogos únicos (WARD, 2008).

O termo motor de jogos surgiu em meados dos anos noventa em referência aos jogos de FPS como o Doom. O Doom foi projetado com uma separação bem definida entre componentes comuns (como sistema de renderização de modelos 3D, sistema de detecção de colisão e sistema de áudio) e a arte do jogo, o mundo do jogo e as regras do jogo que são responsáveis pela experiência do jogador. O valor desta separação tornou-se logo visível quando os desenvolvedores começaram a criar novas armas, personagens, mundos, veículos e novas regras, com somente o mínimo de alteração necessária no motor do jogo (GREGORY, 2009, p. 11).

Um motor de jogos normalmente consiste de um conjunto de ferramentas. Motores de jogos são sistemas de software e geralmente construídos em camadas. Normalmente as camadas mais altas dependem das camadas mais baixas, mas o contrário não é verdade. Às vezes uma camada mais baixa depende de uma camada acima dela, sendo isso conhecido como dependência circular. Este comportamento deve ser evitado, pois isso provoca um alto acoplamento entre os sistemas da arquitetura do motor, tornando o software instável e diminuindo a reusabilidade do código (GREGORY, 2009, p. 28).

O alto acoplamento é um grande problema, mas que pode ser resolvido adotando uma arquitetura baseada em componentes. Em uma arquitetura baseada em componentes, ao invés de se usar herança para conseguir definir novos tipos de comportamento para um objeto do jogo, usa-se componentes para estender os comportamentos de um objeto. Desta maneira pode-se adicionar e remover componentes a um objeto e cada componente define um conjunto de comportamentos. Por exemplo, se o programador deseja que o objeto sofra ação da gravidade basta então adicionar o componente `GravityComponent` ao objeto, não sendo necessário que o objeto herde de uma classe pai que tenha o comportamento desejado, permitindo que a arquitetura possua um baixo acoplamento e uma grande reusabilidade (AEM, 2010).

2.3 TECNOLOGIAS USADAS NO DESENVOLVIMENTO DO MOTOR DE JOGOS

As seções seguintes apresentam as tecnologias adotadas no desenvolvimento do motor de jogos. São elas: HTML5, Javascript, Box2DJS e Kinect utilizando a biblioteca Zigfu.

2.3.1 HTML5

A HTML é a linguagem de marcação utilizada para criação de páginas web. A HTML foi primeiramente projetada para descrever semanticamente documentos científicos. Apesar disso, com o passar dos anos seu *design* e adaptações permitiram que seu uso descrevesse outros tipos de documentos, tais como páginas Web.

O HTML5 será o novo padrão para HTML. Sua implementação continua em progresso, por isso, nem todos os navegadores suportam os novos elementos do HTML5. O HTML5 possui vários elementos e entre eles destacam-se o *canvas*, utilizado para desenhar; os elementos de áudio e vídeo para reprodução de mídia; melhor suporte para armazenamento local dos dados; entre outros (W3SCHOOLS, 2013a).

2.3.2 JAVASCRIPT

Javascript é uma linguagem da Web. Iniciou como um meio de manipular alguns tipos selecionados de elementos em uma página Web (como imagens e campos de formulário) e ganhou destaque com o passar dos anos, devido à necessidade do usuário interagir com as páginas Web (STEFANOV, 2010, p. 1).

O núcleo da linguagem de programação Javascript é baseado nos padrões da linguagem ECMAScript (ECMA, 2011). A terceira versão do padrão foi oficialmente aceita em 1999 e é a implementação atualmente rodando através dos navegadores Web. A quarta versão foi abandonada e a quinta versão foi aprovada em dezembro de 2009, dez anos após a anterior (STEFANOV, 2010, p. 5).

Javascript é uma linguagem de programação leve, cujo código pode ser inserido em páginas HTML. O código Javascript inserido em páginas HTML pode ser executado por todos os navegadores modernos e, além disso, é uma linguagem que possui uma baixa curva de aprendizagem (W3SCHOOLS, 2013b).

2.3.3 BOX2DJS

A Box2DJS é um porte do motor de física Box2D para a linguagem Javascript (BOX2DJS, 2008). A Box2D é um motor para simulação de física de corpos rígidos, desenvolvida originalmente em C++ e possui código aberto. Ela possui várias funcionalidades: detecção de colisão contínua, colisão por categorias e grupos, polígonos convexos e círculos, contato fricção e restituição, física contínua com tempo de impacto, gerenciamento de objetos que podem durmir, pontos de junção, entre outras funcionalidades (BOX2D, 2013).

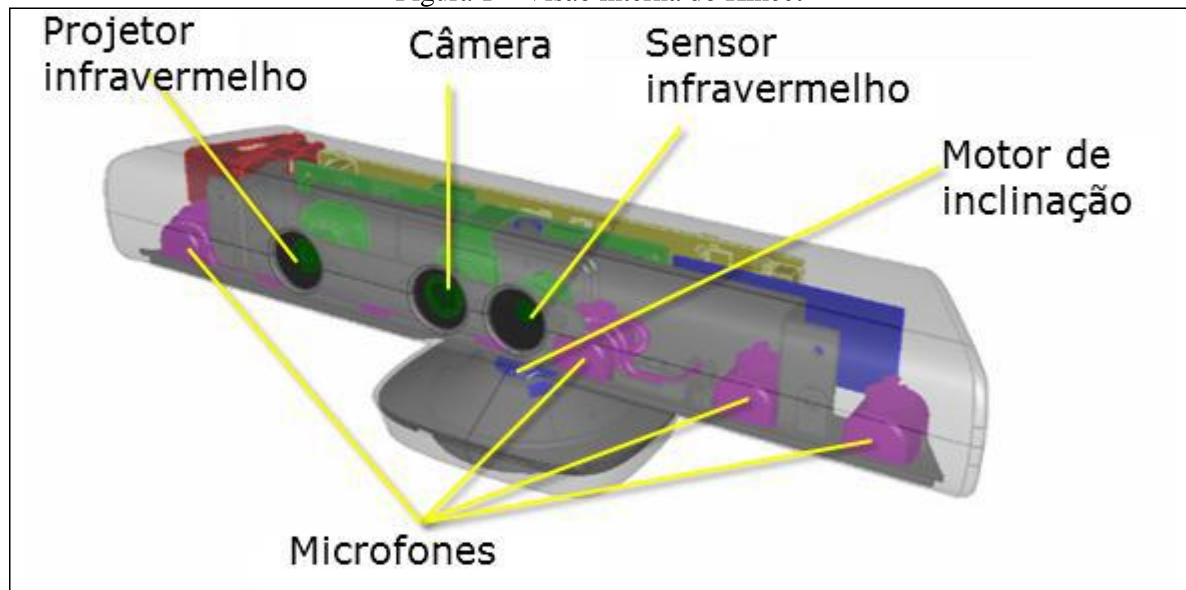
2.3.4 KINECT

O Kinect (Figura 1) é um dispositivo que retorna informações do ambiente real utilizando: um projetor infravermelho, uma câmera infravermelha, uma câmera de vídeo e quatro microfones. Para auxiliar a captura de dados ele também tem um motor de inclinação, e utiliza o Kinect *for* Windows SDK para acessar as informações coletadas destes dispositivos (MICROSOFT, 2012).

Existem outras maneiras de interagir com o Kinect sem ser através do Kinect *for* Windows SDK, como, por exemplo, usando a Zigfu que fornece uma biblioteca Javascript para acessar dados do Kinect. Para utilizar esta biblioteca é necessário que o usuário instale um *plugin* em seu navegador. Este *plugin* pode ser instalado nos navegadores Google

Chrome, Mozilla Firefox, Internet Explorer e Safari, nas plataformas Windows ou Mac. É disponibilizada uma biblioteca Javascript que faz a comunicação entre a aplicação do usuário e o *plugin* instalado no navegador. Esta biblioteca é paga (cem dólares), porém pode ser usada de forma gratuíta mas uma imagem de propaganda fica sobre a aplicação do usuário neste caso (ZIGFU, 2013).

Figura 1 – Visão interna do Kinect



Fonte: adaptado de Microsoft (2012).

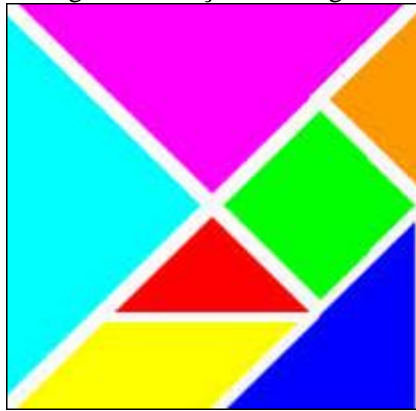
2.4 TANGRAM

O Tangram é um jogo que tem uma grande variedade de representações geométricas e transformações que se pode aplicar sobre estas representações, além de se poder formar letras e ainda por já existir um estudo pedagógico por trás deste jogo, que é utilizado em algumas escolas para o ensino de formas geométricas, polígonos e frações (OSCAR JÚNIOR, 2003, p. 30).

O Tangram, conhecido na China por volta do século VII a.C. como as sete tábuas da astúcia, é um jogo baseado em figuras. A versão mais contada sobre sua origem é a de que o monge Tai-Jin deu uma missão ao seu discípulo Lao-Tan, que consistia em percorrer o mundo e registrar numa placa de porcelana toda a beleza que encontrasse. Muito emocionado por ter sido escolhido para esta missão, o discípulo deixou cair a placa quadrada de porcelana, que quebrou-se em sete pedaços, como as peças do Tangram (LONGHI, 2004).

O Tangram é um quebra-cabeça geométrico (bidimensional) formado por sete peças (Figura 2), de formas geométricas simples (cinco triângulos, um quadrado e um trapézio) que juntas formam um quadrado. Combinando estas peças originam-se outras figuras, podendo representar uma grande variedade do mundo real.

Figura 2 – Peças do Tangram



Fonte: Skillarea (2013).

2.5 TRABALHOS CORRELATOS

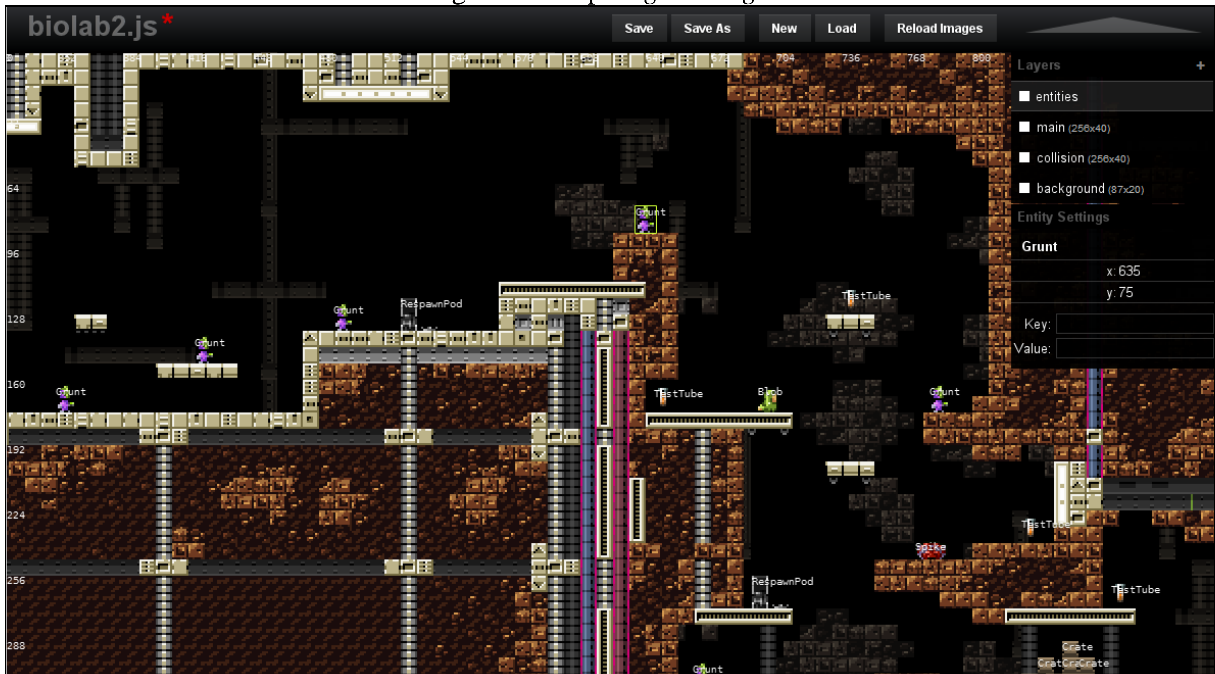
Existem muitos motores e editores de jogos no mercado, alguns livres e outros pagos, para várias linguagens e plataformas existentes. Estes motores de jogos possuem o objetivo de abstrair a parte comum do desenvolvimento de um jogo e fornecer um esqueleto básico para a criação de um novo jogo, fornecendo funções para: gerenciamento de áudio, gerenciamento de eventos, renderização de cenas, tratamento de colisão, entre outras funções. Dentre estes motores foram selecionados o Impact (IMPACT, 2013) e a Unity 3D (UNITY, 2013), os quais são descritos a seguir.

2.5.1 Impact

O Impact (Figura 3) é um motor de jogos em Javascript que permite desenvolver jogos em HTML5 para navegadores móveis e *desktop*. Impact pode ser executado em todos os navegadores compatíveis com HTML5: Firefox, Chrome, Safári, Opera e Internet Explorer 9. Isso também inclui o iPhone, iPod Touch e iPad. O Impact vem com um editor de níveis versátil que permite criar mundos no jogo facilmente (IMPACT, 2013).

O Impact abstrai várias partes do desenvolvimento do jogo, tornando o processo mais produtivo, como: criação de níveis, detecção de colisão, tratamento de colisões, carregamento de recursos como imagens e áudios, gerenciamento de animações e renderização dos objetos do jogo. Além disso, também disponibiliza ferramentas para depuração do jogo (IMPACT, 2013).

Figura 3 – Impact *game engine*



Fonte: Impact (2013).

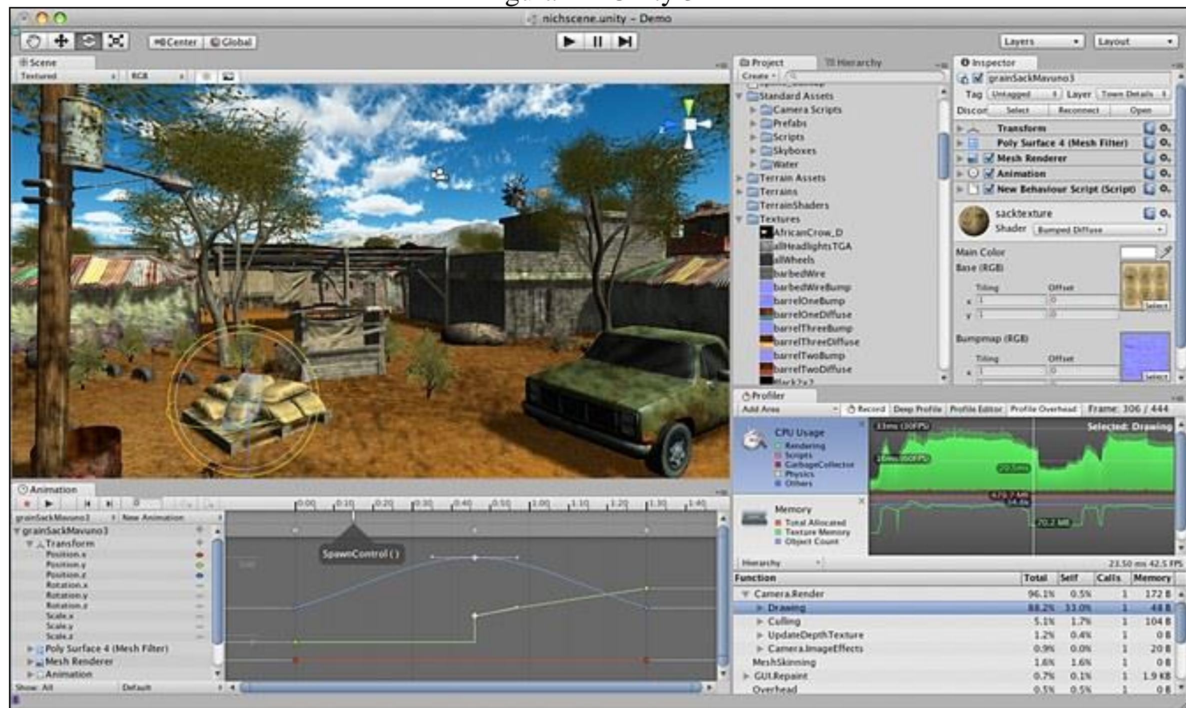
A Figura 3 representa a interface do Impact, onde se pode observar uma fase de um jogo sendo construída e ao lado direito um menu para acessar e modificar as entidades do jogo.

2.5.2 Unity 3D

A Unity 3D (Figura 4) é um ecossistema de desenvolvimento de jogos: um motor de renderização integrado com um conjunto de ferramentas e fluxos de trabalho para criar conteúdo 3D interativo, pode-se publicar em diferentes plataformas, com *assets* (recursos utilizados nos jogos como imagens e áudios) prontos na Asset Store e uma comunidade que compartilha conhecimento (UNITY, 2013).

A Unity 3D é um motor de jogos muito difundido no mercado e com várias funcionalidades como: um sistema de fluxo de trabalho que permite que se monte uma cena de forma rápida, carregamento de áudios e renderização de imagens, um sistema flexível de animação de personagens e disponibiliza várias funcionalidades referentes a parte de física de corpos rígidos e de inteligência artificial para o desenvolvedor. Possui também uma arquitetura orientada a componentes, permitindo ao desenvolvedor criar novos componentes para seu jogo (UNITY, 2013).

Figura 4 – Unity 3D



Fonte: Demozcene (2010).

A Figura 4 representa o ambiente de desenvolvimento da Unity 3D. Pode-se visualizar a cena atual em que o usuário está trabalhando. Ao lado é possível visualizar a estrutura do projeto organizada em forma de árvore e ao lado alguns atributos do objeto selecionado na cena.

2.5.3 Comparação entre os trabalhos correlatos

O Quadro 1 mostra a comparação das características dos trabalhos correlatos. Ambos os trabalhos suportam simulações físicas, possuem um editor de cena, ferramentas de *debug* e exportam o jogo para dispositivos móveis. A Impact é preparada para jogos em duas dimensões, enquanto a Unity 3D para jogos em três dimensões. A Unity 3D possui uma arquitetura orientada a componentes e suporte nativo a *joystick*.

Quadro 1 – Características dos trabalhos correlatos

Características	**Unity 3D**	**Impact**
Suporte 2D		X
Suporte 3D	X	
Orientada a componentes	X	
Suporte a física	X	X
Editor de cena	X	X
Ferramentas de debug	X	X
Geração para dispositivos móveis	X	X
Código fonte aberto		
Sem necessidade de licença		
Possui integração com Kinect		
Possui integração com Joystick	X	

3 DESENVOLVIMENTO

Neste capítulo são apresentadas as etapas do desenvolvimento do motor de jogos 2D e da ferramenta de edição, assim como da aplicação de testes e demonstração. São abordados os principais requisitos, a especificação, a implementação e os resultados e discussão.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O motor de jogos 2D deverá:

a) criar e editar uma cena em duas dimensões (Requisito Funcional - RF);
b) criar e editar um novo objeto de cena (RF);
c) adicionar ou remover objetos da cena em tempo de execução (RF);
d) adicionar uma animação a um objeto e parar ou executar a animação em tempo de execução (RF);
e) criar e editar novos componentes (RF);
f) adicionar ou remover componentes a um objeto de cena em tempo de execução (RF);
g) tratar eventos de entrada de teclado, mouse, Kinect e *joysticks* (RF);
h) possuir uma ferramenta para editar a cena (RF);
i) criar e editar novas camadas (RF);
j) criar diferentes visões de câmeras (RF);
k) adicionar imagens e sons a um objeto do jogo (RF);
l) criar e editar um projeto de um jogo (RF);
m) a ferramenta para edição de cena deverá ser web (Requisito Não Funcional - RNF);
n) implementar os comportamentos dos objetos utilizando Javascript (RNF);
o) utilizar HTML5 e executar nos diversos navegadores do mercado que suportam HTML5 (RNF);
p) renderizar uma cena com até cinquenta objetos mantendo o limite mínimo de trinta quadros por segundo (RNF).

3.2 ESPECIFICAÇÃO

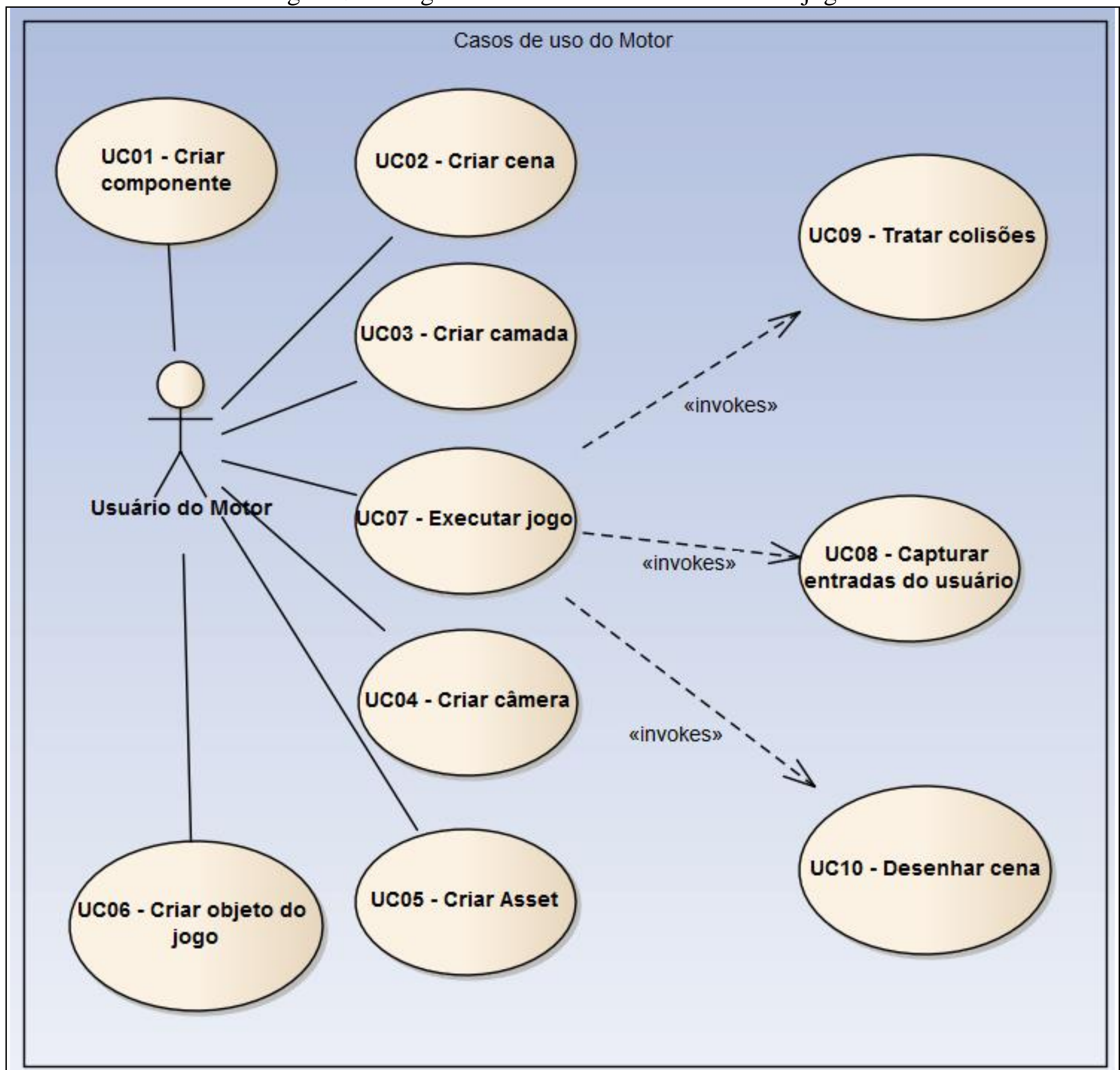
A especificação deste trabalho foi desenvolvida com a ferramenta Enterprise Architect, utilizando os conceitos do paradigma de orientação a objetos e por meio de diagramas da *Unified Modeling Language* (UML). Assim, nas próximas seções são apresentados os

diagramas de caso de uso, diagramas de classes, diagrama de sequência e diagrama de arquitetura.

3.2.1 Casos de uso do motor de jogos

Nesta seção são descritos os casos de uso das funcionalidades disponibilizadas pelo motor de jogos, ilustrados pela Figura 5. O detalhamento de cada caso de uso pode ser visto no Apêndice A.

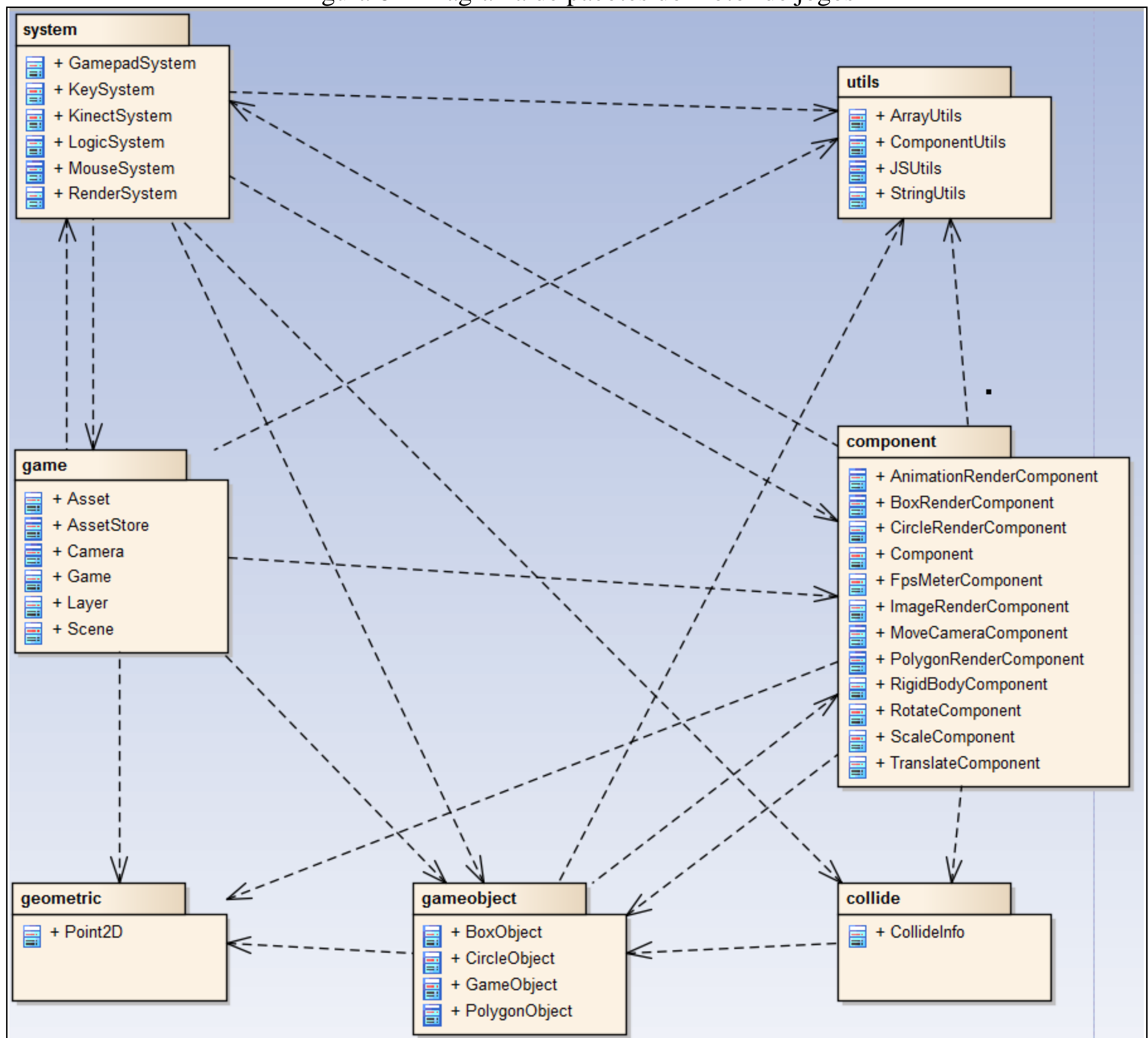
Figura 5 – Diagrama de casos de uso do motor de jogos



3.2.2 Diagramas de classes do motor de jogos

Nesta seção são descritas as classes e estruturas que constituem o motor de jogos. Na Figura 6 são exibidas as dependências entre os pacotes do motor. As classes destes pacotes serão detalhadas a seguir.

Figura 6 – Diagrama de pacotes do motor de jogos



3.2.2.1 Pacote `component`

O pacote denominado `component` possui as classes necessárias para definir um componente dentro do motor de jogos assim como componentes previamente disponibilizados, como pode ser visto na Figura 7.

Figura 7 – Diagrama de classes do pacote `component`

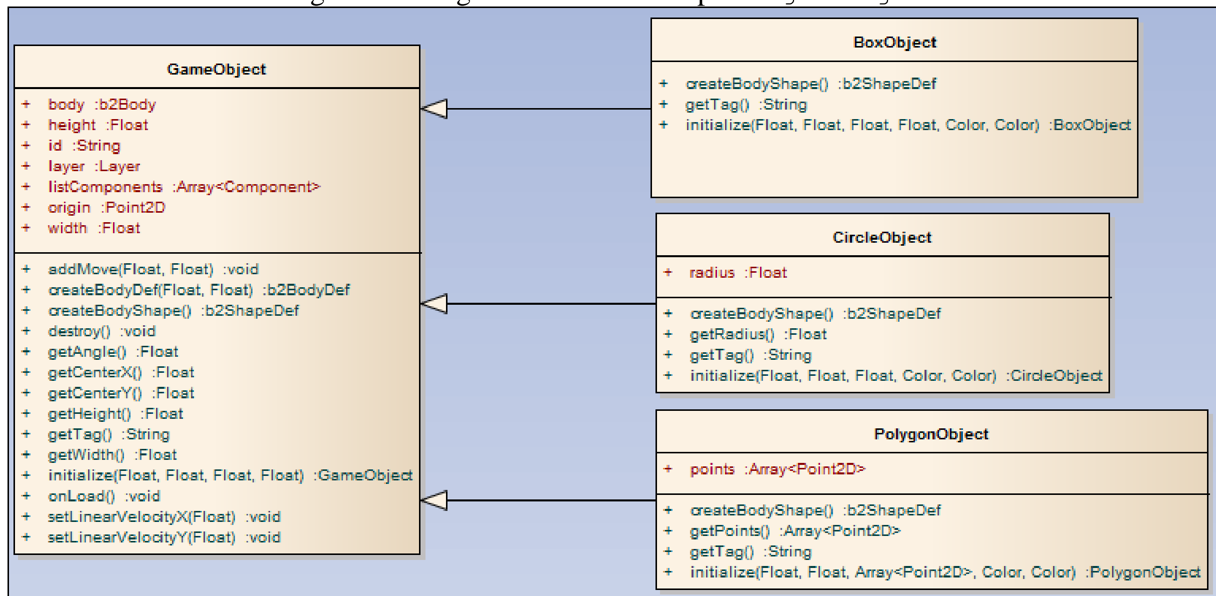
A classe `Component` é a classe que define os atributos e métodos comuns a todos os componentes dentro do motor de jogos. As classes que herdarem dela devem sobrescrever os eventos para programar a funcionalidade desejada. A classe `FpsMeterComponent` define um componente que quando associado ao jogo, mede a quantidade de vezes que o jogo está sendo desenhado e exibe a taxa de desenho na tela do jogo. A classe `RigidBodyComponent` define um componente que encapsula todo o comportamento da física de um objeto, cria o corpo físico de um objeto e o associa ao mundo da Box2DJS. A classe `TranslateComponent` define um componente que realiza a translação de um objeto. A classe `RotateComponent` define um componente que realiza a rotação de um objeto. A classe `ScaleComponent` define um objeto que realiza a escala de um objeto. A classe `AnimationRenderComponent` define um componente que recebe uma imagem e parâmetros de repetição para desenhar animações que podem ser associadas a um objeto. A classe `MoveCameraComponent` define um componente que quando associado ao jogo permite ao usuário escalar, rotacionar ou transladar a câmera atual do jogo. A classe `ImageRenderComponent` define um componente que desenha uma imagem estática na tela do jogo. A classe `BoxRenderComponent` define um componente que associado a um `BoxObject` realiza o desenho do mesmo na tela do jogo. A classe `CircleRenderComponent` define um componente que associado a um `CircleObject` desenha

o mesmo na tela do jogo. A classe `PolygonRenderComponent` define um componente que associado a um `PolygonObject` desenha o mesmo na tela.

3.2.2.2 Pacote `gameobject`

O pacote denominado `gameobject` possui a classe necessária para definir um objeto do jogo dentro do motor assim como alguns objetos disponibilizados previamente, como pode ser visto na Figura 8.

Figura 8 – Diagrama de classes do pacote `gameobject`

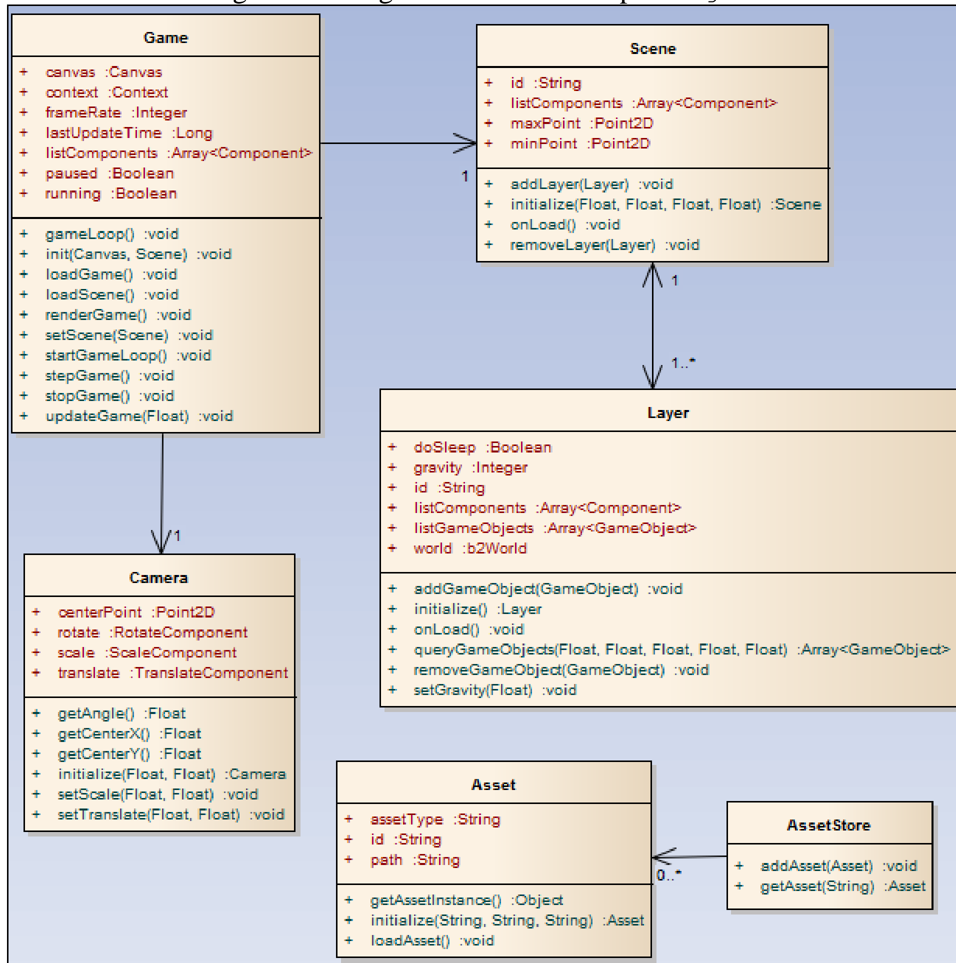


A classe `GameObject` define um objeto do jogo dentro do motor de jogos. Ela possui os atributos e métodos comuns a todos os objetos do jogo e encapsula dentro dela um atributo `b2Body` que representa um corpo rígido para o motor de física Box2DJS. As classes `BoxObject`, `CircleObject` e `PolygonObject` definem implementações concretas de objetos do jogo do tipo retangular, circular e polígono.

3.2.2.3 Pacote `game`

O pacote denominado `game` possui as classes (Figura 9) necessárias para gerenciar a execução do jogo e da cena atual.

Figura 9 – Diagrama de classes do pacote `game`

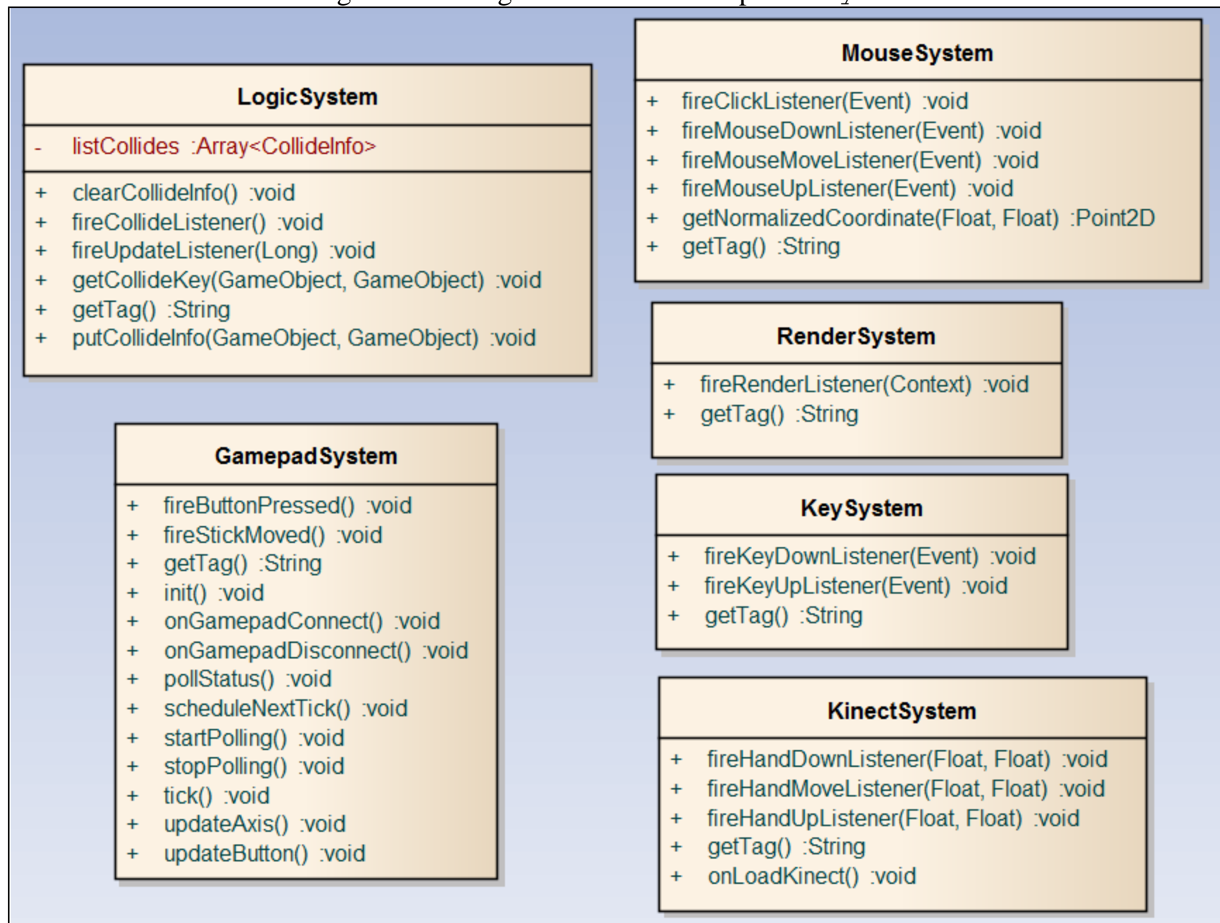


A classe `Asset` representa um recurso de imagem ou áudio que pode ser utilizado pelo jogo. A classe `AssetStore` é apenas uma classe para agrupar e facilitar o acesso a objetos do tipo `Asset`. A classe `Layer` define uma camada de uma cena. É ela que irá agrupar os objetos do jogo. Esta classe encapsula um atributo `b2World` que representa um mundo para a Box2DJS, ou seja, todo tratamento de física que ocorre em uma camada é independente das outras. A classe `Scene` define uma cena do jogo e pode ser formada por uma ou mais camadas. A classe `Camera` define uma câmera para a visualização do jogo pelo usuário. Esta câmera pode ser transladada, rotacionada ou escalada para mudar o modo que o usuário vê o jogo sem modificar a posição dos objetos. A classe `Game` é a classe responsável por gerenciar a cena atual sendo desenhada e o *loop* principal do jogo.

3.2.2.4 Pacote `system`

O pacote denominado `system` possui as classes responsáveis por disparar os eventos do motor de jogos aos componentes. Este pacote possui cinco classes como pode ser visto na Figura 10.

Figura 10 – Diagrama de classes do pacote `system`

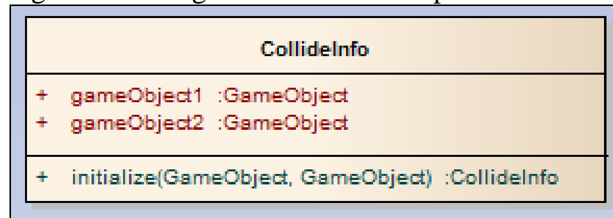


A classe `LogicSystem` é responsável por disparar os eventos de atualização dos componentes através do método `fireUpdateListener(deltaTime)` e também eventos de colisão entre objetos através do método `fireCollideListener()`. A classe `MouseSystem` é responsável por se registrar nos eventos de mouse disparados pelo navegador e propagá-los para os componentes. A classe `KinectSystem` é responsável por se registrar nos eventos disparados pela biblioteca Zigfu (que faz a comunicação com o Kinect, ver seção 2.3.4) e propagá-los para os componentes. A classe `GamepadSystem` é responsável por se registrar nos eventos disparados pelo *joystick* e propagá-los para os componentes. A classe `KeySystem` se registra nos eventos de teclado disparados pelo navegador e é responsável por propagar estes eventos aos componentes. A classe `RenderSystem` é responsável por disparar os eventos de renderização nos componentes, através do método `fireRenderListener(Context)`.

3.2.2.5 Pacote `collide`

O pacote denominado `collide` contém apenas a classe chamada `CollideInfo` como pode ser visto na Figura 11. Esta classe possui a responsabilidade de guardar um par de objetos que estão colidindo.

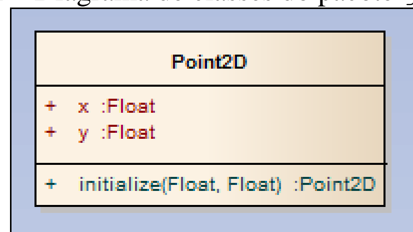
Figura 11 – Diagrama de classes do pacote `collide`



3.2.2.6 Pacote `geometric`

O pacote denominado `geometric` possui apenas a classe `Point2D` como pode ser visto na Figura 12. Esta classe serve para representar um ponto em um espaço de duas dimensões.

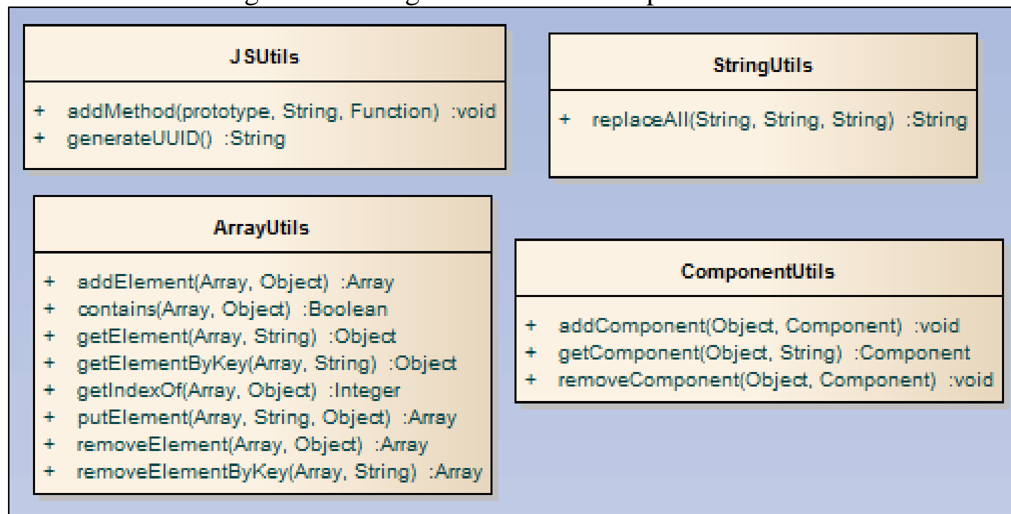
Figura 12 – Diagrama de classes do pacote `geometric`



3.2.2.7 Pacote `utils`

O pacote denominado `utils` possui classes utilitárias para operações Javascript, como pode ser visto na Figura 13.

Figura 13 – Diagrama de classes do pacote `utils`

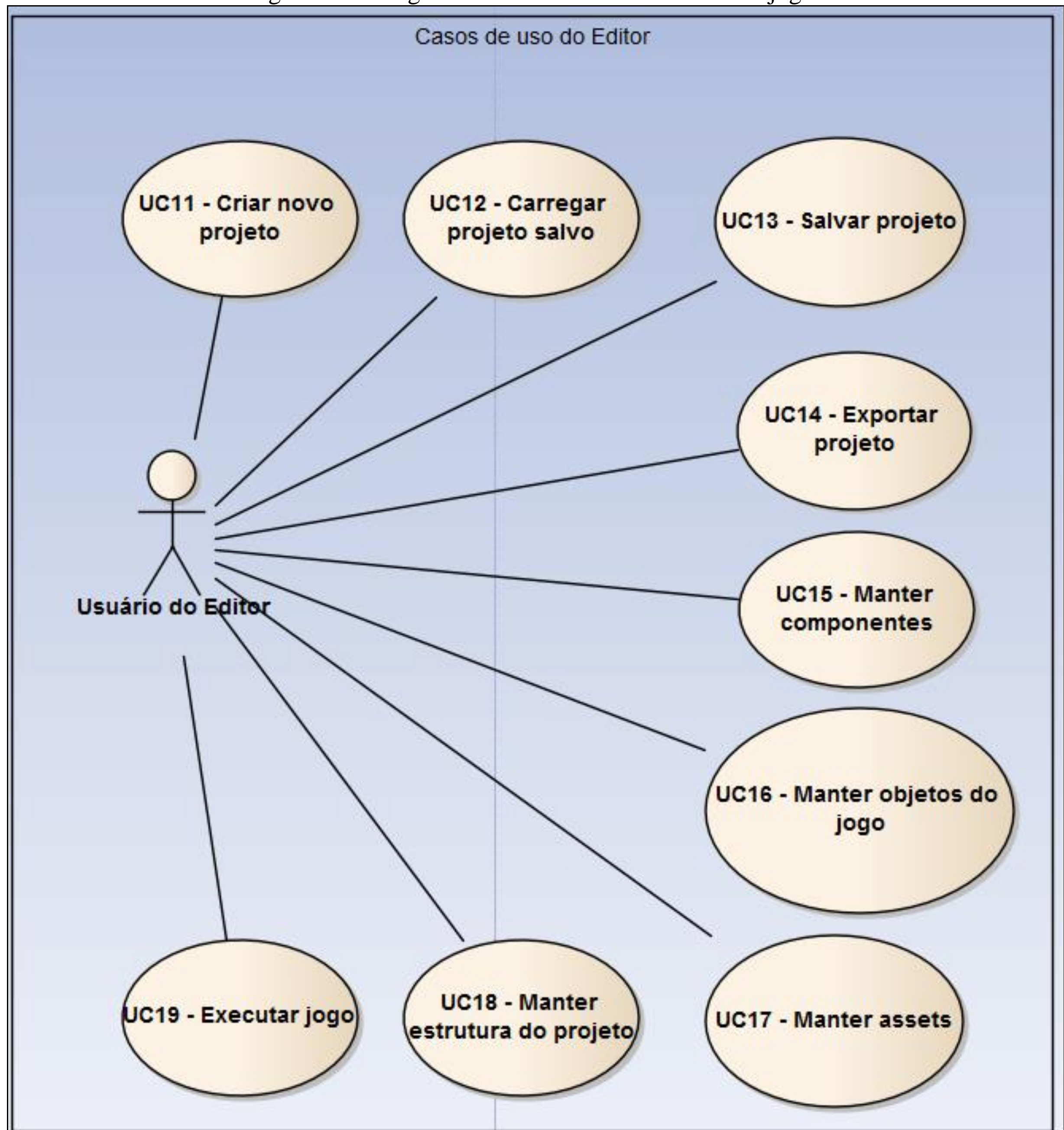


A classe `JSUtils` possui método utilitários para geração de um identificador único e para simular sobrecarga de métodos em Javascript. A classe `StringUtils` possui apenas um método que serve para substituir todas as ocorrências de um texto dentro de outro texto. A classe `ArrayUtils` possui métodos auxiliares para operações em listas. A classe `ComponentUtils` possui métodos para adicionar, remover e buscar componentes em algum objeto.

3.2.3 Casos de uso do editor de jogos

Nesta seção são descritos os casos de uso das funcionalidades disponibilizadas pelo editor de jogos, ilustrados pela Figura 14. O detalhamento de cada caso de uso pode ser visto no Apêndice A.

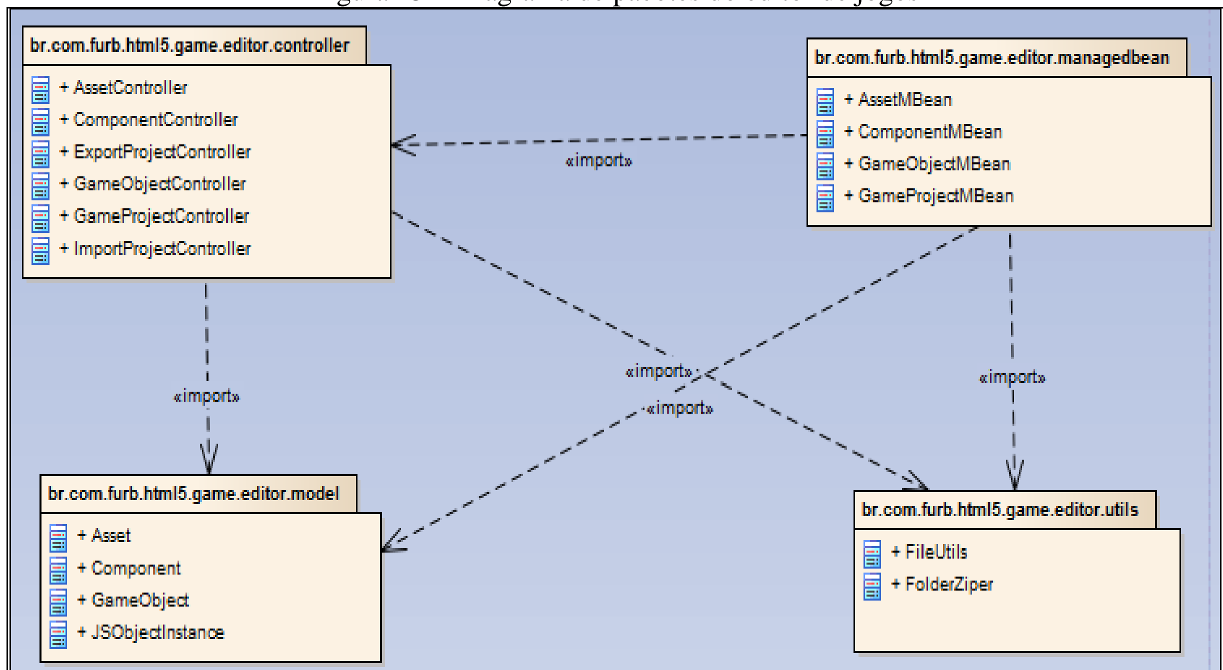
Figura 14 – Diagrama de casos de uso do editor de jogos



3.2.4 Diagramas de classes do editor de jogos

Nesta seção são descritas as classes e estruturas que constituem o editor de jogos. Na Figura 15 são exibidas as dependências entre os pacotes do editor. As classes destes pacotes serão detalhadas a seguir.

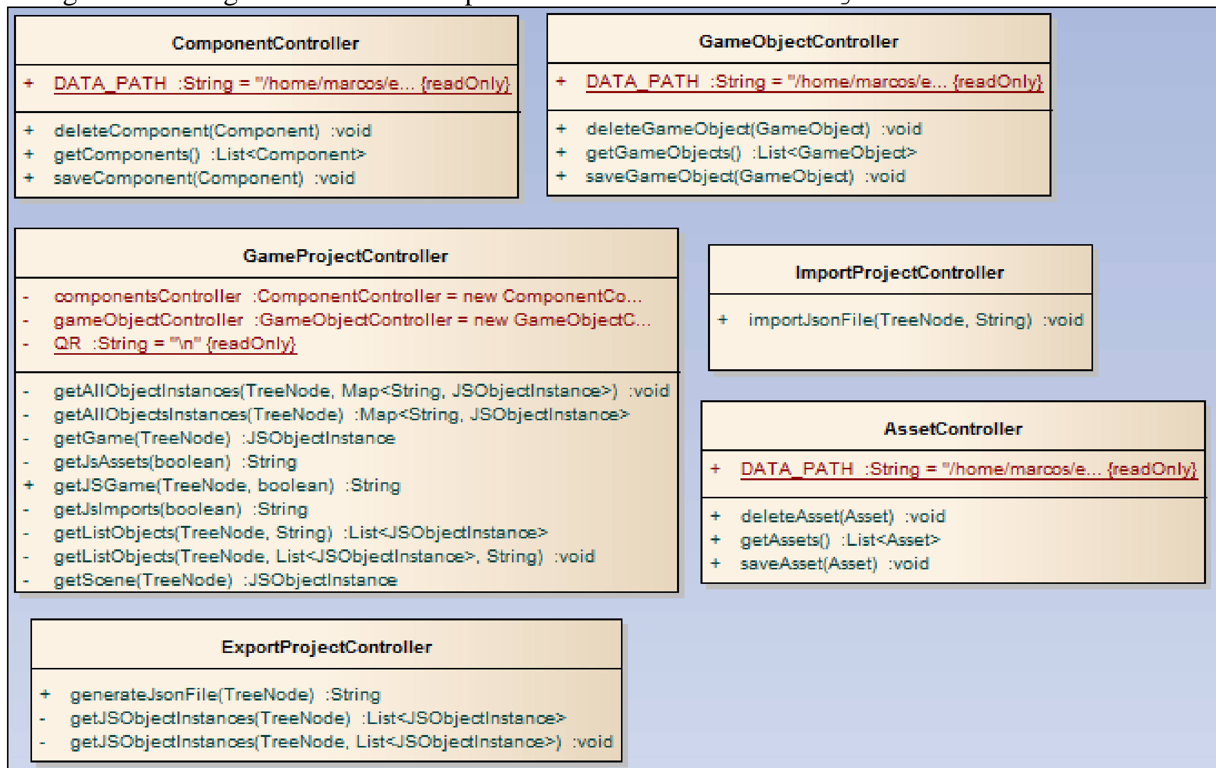
Figura 15 – Diagrama de pacotes do editor de jogos



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

O pacote denominado `br.com.furb.html5.game.editor.controller` possui as classes (Figura 16) controladoras do editor, responsáveis por realizar as ações disparadas pelo usuário no editor.

Figura 16 – Diagrama de classes do pacote `br.com.furb.html5.game.editor.controller`



A classe `ComponentController` é responsável por executar as ações de buscar, excluir e salvar componentes no servidor. A classe `GameObjectController` é responsável por buscar, excluir e salvar objetos do jogo no servidor. A classe `AssetController` é responsável por buscar, excluir e salvar imagens ou áudios no servidor. A classe `ImportProjectController` é responsável por receber um arquivo de dados JSON e recriar a estrutura da árvore do projeto. A classe `ExportProjectController` é responsável por gerar um arquivo de dados JSON que represente a estrutura de árvore do projeto. A classe `GameProjectController` é responsável por gerar todo o HTML necessário para a execução do jogo no editor a partir da árvore do projeto.

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

O pacote denominado `br.com.furb.html5.game.editor.managedbean` possui as classes responsáveis por atender as requisições feitas pelas tela web do editor e instanciar os controladores, como pode ser visto na Figura 17.

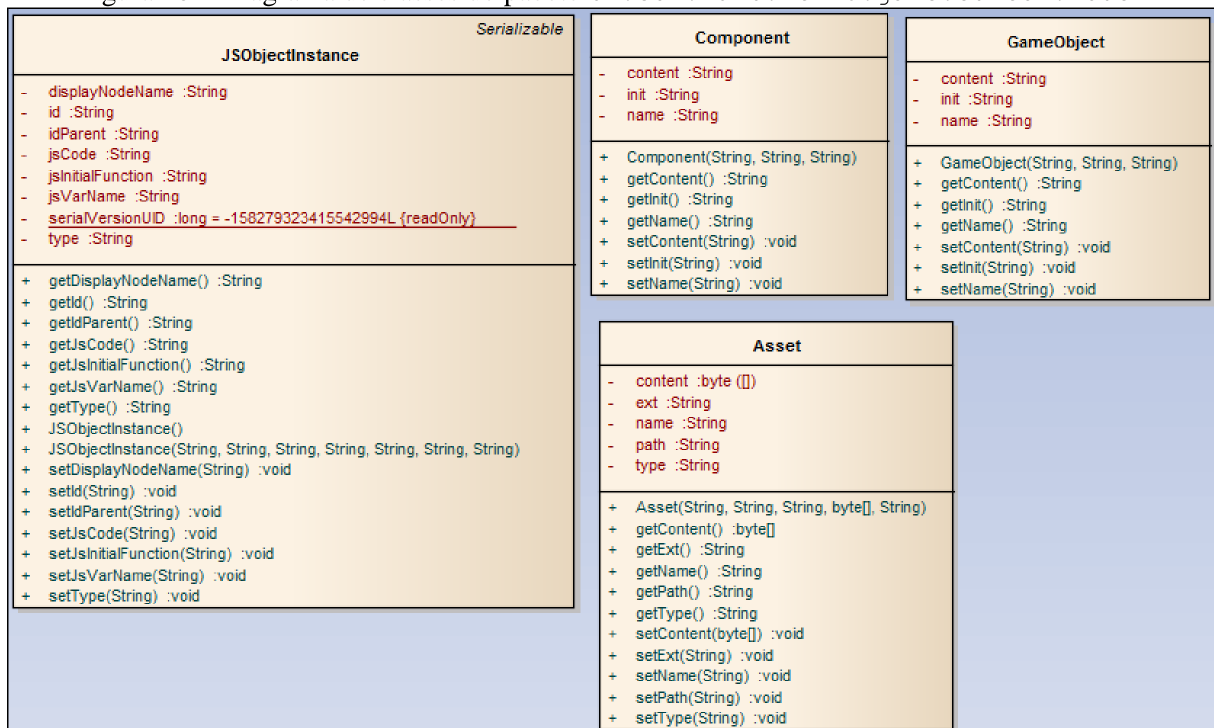
Figura 17 – Diagrama de classes do pacote `br.com.furb.html5.game.editor.managedbean`

A classe `AssetMBean` é responsável por atender as requisições da tela de cadastro e edição de *assets*. A classe `ComponentMBean` é responsável por atender as requisições da tela de cadastro e edição de componentes. A classe `GameObjectMBean` é responsável por atender as requisições da tela de cadastro e edição de objetos do jogo. A classe `GameProjectMBean` é responsável por atender as ações feitas na árvore de projeto do editor.

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

O pacote denominado `br.com.furb.html5.game.editor.model` possui as entidades utilizadas pelo editor como pode ser visto na Figura 18.

Figura 18 – Diagrama de classes do pacote `br.com.furb.html5.game.editor.model`

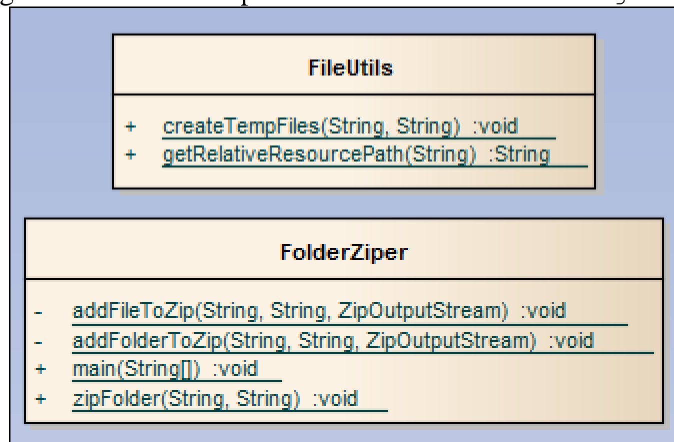


A classe `Component` possui os atributos e métodos necessários para representar um *script* que contém o código fonte de um componente. A classe `GameObject` possui os atributos e métodos necessários para representar um *script* no servidor que contenha o código fonte de um objeto do jogo. A classe `Asset` representa algum recurso (imagem ou áudio) que pode ser usado pelo usuário no jogo. A classe `JSObjectInstance` representa instâncias de objetos que são adicionadas pelo usuário na árvore de projeto do editor.

3.2.4.4 Pacote `br.com.furb.html5.game.editor.utils`

O pacote denominado `br.com.furb.html5.game.editor.utils` possui as classes utilitárias usadas pelo editor como pode ser visto na Figura 19.

Figura 19 – Diagrama de classes do pacote `br.com.furb.html5.game.editor.utils`

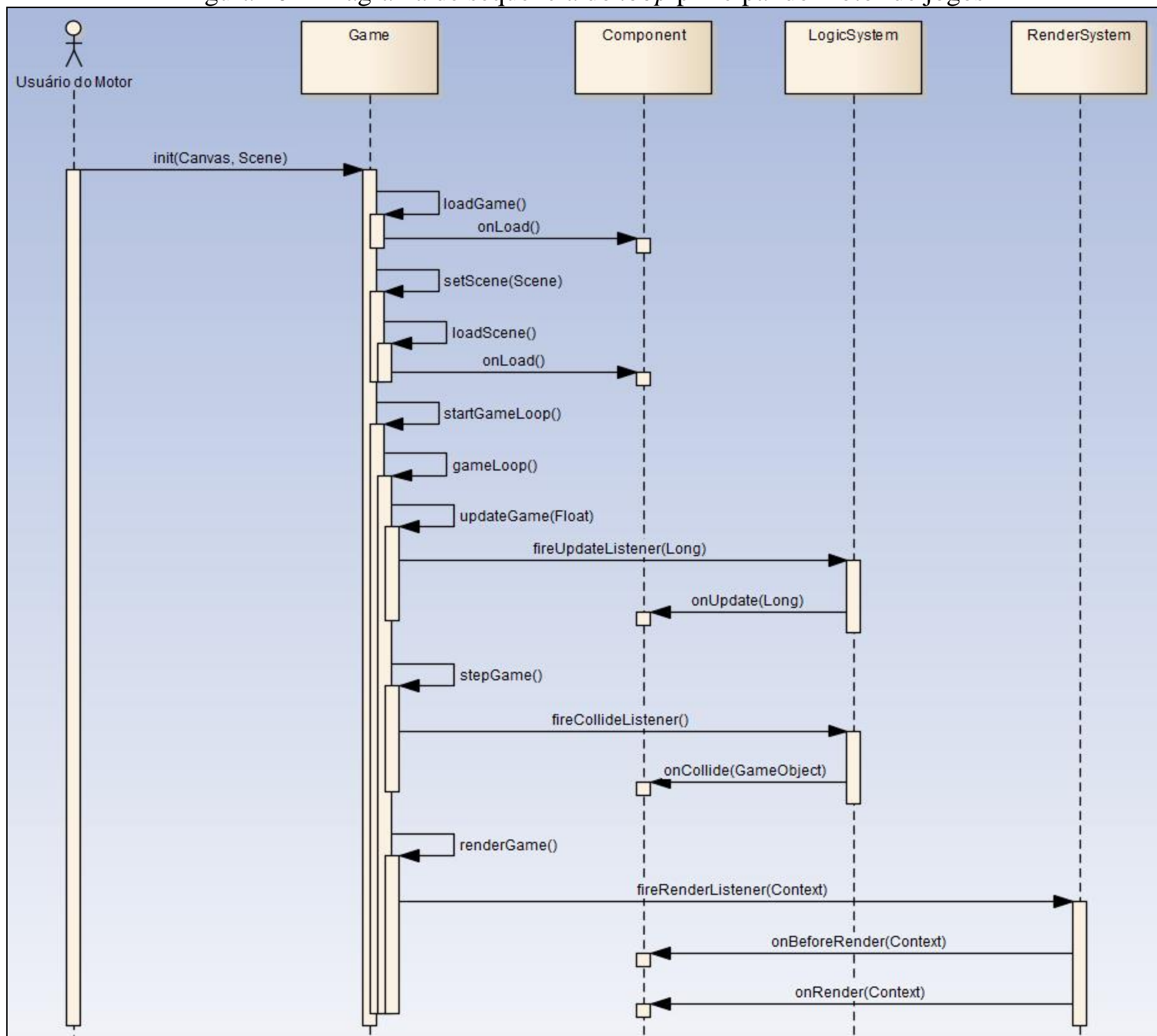


A classe `FileUtils` possui métodos para arquivos de um diretório de origem para um diretório temporário no servidor. A classe `FolderZiper` server para realizar a compactação de um diretório e todos seus subdiretórios em um arquivo `.zip`.

3.2.5 Diagramas de sequência

Esta seção apresenta um diagrama de sequência que descreve o *loop* principal do jogo gerenciado pelo motor. A Figura 20 apresenta este diagrama.

Figura 20 – Diagrama de sequencia do *loop* principal do motor de jogos

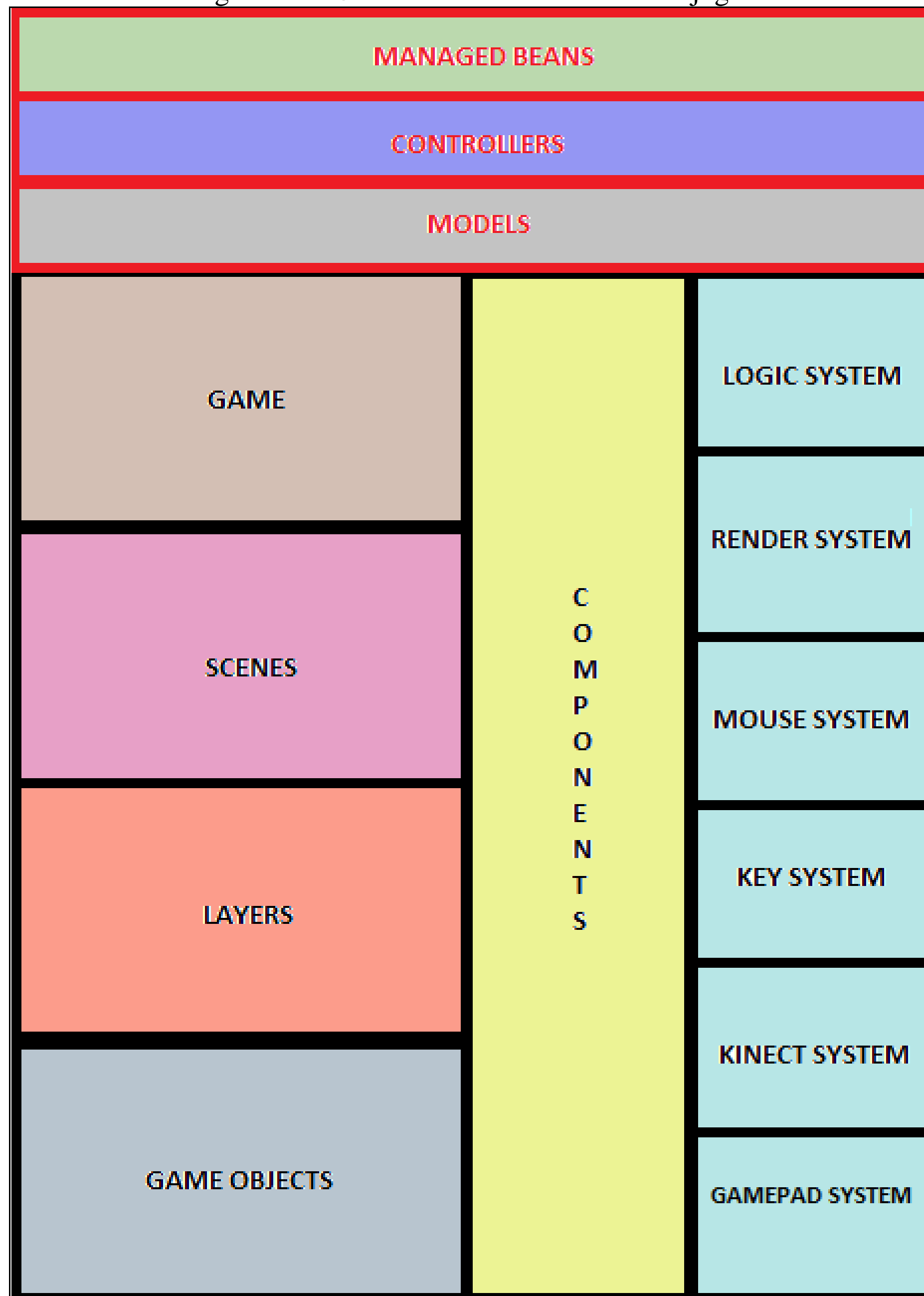


No diagrama apresentado o usuário acessa o jogo e o método de inicialização (`init`) do jogo é executado, este método primeiramente faz o carregamento de todos os componentes associados ao jogo. Depois configura a cena que será executada através do método `setScene` e carrega então todos os componentes da cena (`loadScene`) e então executa o *loop* do jogo (`gameLoop`). Este *loop* se divide em três partes, a primeira parte é a de atualização (`updateGame`), onde todos os componentes recebem o evento de atualização. Após isso são realizadas as verificações do mundo da Box2DJS através do método `stepGame` e para cada camada é disparado o evento de colisão para os objetos que estão colidindo através do método `fireCollideListener`. A terceira e última parte é a de desenhar a cena atualizada, que consiste em chamar o evento de desenhar em todos os componentes utilizando o método `fireRemderListener`.

3.2.6 Diagramas de arquitetura

Esta seção apresenta o diagrama das camadas da arquitetura do motor e editor de jogos. A Figura 21 apresenta este diagrama.

Figura 21 – Camadas do editor e motor de jogos



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

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

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas na implementação do motor e do editor de jogos, bem como, detalhes das principais classes e rotinas implementadas durante o desenvolvimento do motor de jogos.

3.3.1 Técnicas e ferramentas utilizadas

O desenvolvimento do motor de jogos utilizou a linguagem Javascript e elementos do HTML5, principalmente o *canvas*. Como ambiente de desenvolvimento para o motor de jogos foi utilizado o editor de texto Sublime Text 2.0.2 (SUBLIME, 2013). Para abstrair o comportamento de física dos objetos foi integrado, junto ao motor de jogos, um motor de simulações físicas conhecido como Box2DJS. O motor de jogos também possui interações com o Kinect, criada utilizando a biblioteca Zigfu (ZIGFU, 2013) e com *joysticks* utilizando a Gamepad API (GAMEPAD, 2013).

O desenvolvimento do editor de jogos utilizou a linguagem Java, com o *framework Java Server Faces* 2.2 (JSF) para desenvolvimento web em Java. Utilizou-se também uma biblioteca de componentes para JSF conhecida como Primefaces 3.5 (PRIMEFACES, 2011). Para o ambiente de desenvolvimento usou-se o Eclipse IDE for Java EE Developers, que já possui uma série de plugins configurados para facilitar o desenvolvimento de aplicações web com JSF. Como servidor de aplicação foi usado o JBoss 6.1.0 (JBOSS, 2013).

Os navegadores utilizados durante o desenvolvimento foram o GoogleChrome 32.0.1678.0 dev-m Aura, Internet Explorer 10.0.9200.16721, Opera 17.0 e Mozilla Firefox 25.0.

Para execução e depuração do motor e do editor de jogos utilizou-se um computador *desktop* com processador AMD FX(tm)-6100 Six-Core, memória de 8 GB DDR3 e uma placa de vídeo AMD Radeon HD 6850.

3.3.2 O motor de jogos

Essa seção descreve os conceitos utilizados no motor de jogos e sua arquitetura. Também mostra como foi criada a abstração da arquitetura orientada a componentes.

3.3.2.1 Arquitetura do motor de jogos

O motor de jogos é constituído por um conjunto de cenas. Uma cena é uma agrupadora de camadas e delimita a área em que os objetos irão responder a física. Uma cena possui ao menos uma camada, mas pode ter várias. Uma camada no motor de jogos representa uma agrupadora de objetos do jogo, onde serão adicionados os objetos. Cada camada também possui seu próprio mundo de simulações físicas. Isso significa que os objetos irão colidir e reagir fisicamente apenas com objetos que estão na mesma camada. As camadas são desenhadas na ordem em que elas estão adicionadas nas cenas. Os objetos do jogo devem sempre ser associados a uma camada. Os objetos do jogo são objetos que podem ser controlados pelo jogador ou pelo computador, ou ainda apenas objetos que formam o cenário do jogo. Cada objeto do jogo possui um ponto que representa sua posição dentro do mundo em duas dimensões da camada. Estes objetos também possuem um corpo físico que é criado e adicionado ao mundo de simulaçoes físicas da camada em que ele está. Basicamente estes corpos podem ser retângulos, circunferências ou algum polígono. O jogo também pode possuir um conjunto de câmeras, porém apenas uma pode estar ativa. A câmera serve para modificar o modo como o jogador está visualizando a cena atual, porém sem modificar as posições dos objetos do jogo. A câmera pode ser rotacionada, escalada ou transladada. O jogo pode possuir um conjunto de *assets,* que são arquivos de imagens ou áudio que o usuário configura e que podem ser acessados a qualquer momento. O motor de jogos também possui uma séries de *systems,* para criar integrações (feitas por eventos) entre o motor de jogos e dispositivos de entrada de dados (teclado, mouse, Kinect e *joysticks*), para realizar o desenho das cenas e atualizações dos objetos.

Uma das partes implementadas na arquitetura do motor foram os componentes. Estes componentes podem ser associados aos objetos do jogo, ao jogo, a uma cena ou a uma camada. Os componentes podem ser vistos como comportamentos, que ao serem associados a um objeto do jogo, adicionam comportamentos a este objeto, que podem ser removidos em tempo de execução. Os componentes são a maneira do usuário adicionar extensões aos seus objetos. Por exemplo, se o usuário deseja que seu personagem pule, basta criar um componente que encapsule esta lógica e adicionar uma instância deste componente ao seu personagem. Este componente pode ser removido a qualquer momento do objeto, removendo assim o comportamento de pular. Os componentes podem ser associados a vários objetos. Assim, a lógica de pular poderia ser reaproveitada caso outro objeto da cena também necessite de um comportamento similar.

3.3.2.2 Estrutura do motor de jogos

O motor e o editor de jogos foram criados a fim de facilitar o desenvolvimento de jogos em HTML5, abstraindo as partes comuns no desenvolvimento de um jogo e permitindo que desenvolvedores que não possuam um grande conhecimento na área de jogos consigam desenvolver aplicações de forma mais simplificada.

A estrutura básica de um jogo no motor consiste de um jogo (`Game`) com uma cena (`Scene`) que contenha ao menos uma camada (`Layer`). Para dar início ao jogo deve-se chamar o método `init` do objeto `Game`. Este método irá carregar a cena e iniciar o *loop* principal do jogo (Figura 20). O método `init` e o *loop* podem ser visto no Quadro 2.

Quadro 2 – Código fonte de `Game`

```
1  var Game = new function(){
2
3      /**
4       * Método que inicializa o jogo.
5       *
6       * @author Marcos Harbs
7       * @method init
8       * @param {Canvas} canvas
9       * @param {Scene} scene
10      */
11     this.init = function(canvas, scene){
12        this.canvas = canvas;
13         this.context = canvas.getContext("2d");
14         this.camera = new Camera().initialize(canvas.width/2, canvas.height/2);
15         this.loadGame();
16         this.setScene(scene);
17         this.lastUpdateTime = 0;
18         this.frameRate = 60;
19         this.startGameLoop();
20         this.running = true;
21         this.paused = false;
22     }
23
24     …
25
26     /**
27      * Loop principal do jogo.
28      *
29      * @author Marcos Harbs
30      * @method gameLoop
31      */
32     this.gameLoop = function(){
33         var deltaTime = (Date.now() - this.lastUpdateTime) / 1000;
34         if(!Game.paused){
35             this.updateGame(deltaTime);
36             this.stepGame();
37             this.renderGame();
38         }
39         this.lastUpdateTime = Date.now();
40     }
41     …
42 }
```

Uma cena pode conter várias camadas definidas pelo objeto `Layer` que contém uma lista de `GameObject`. Os `GameObject`, por sua vez, representam os objetos criados pelo usuário. O objeto `Layer` também possui um atributo `world` do tipo `b2World`. O `b2World` faz parte da biblioteca Box2DJS e serve para definir um mundo particular onde serão aplicadas as simulações físicas. Como cada camada possui seu próprio mundo, então as simulações físicas

de uma camada não influenciam em outras. O objeto `Layer` possui métodos para adicionar e remover objetos do jogo, assim como também buscar objetos que estão em um determinado espaço do mundo. Estes métodos podem ser vistos no Quadro 3.

Quadro 3 – Código fonte de `Layer`

```
1   function Layer(){}
2
3   /**
4    * Faz uma query no mundo da Box2D e retorna os
5    * objetos selecionados.
6    *
7    * @author Marcos Harbs
8    * @method queryGameObjects
9    * @param {Float} x
10   * @param {Float} y
11   * @param {Float} w
12   * @param {Float} h
13   * @param {Float} max
14   * @return {Array} gameObjects
15   */
16   Layer.prototype.queryGameObjects = function(x, y, w, h, max){
17       var aabb = new b2AABB();
18       var shapes = [];
19       var gameObjects = [];
20       aabb.minVertex.Set(x-w/2, y-h/2);
21       aabb.maxVertex.Set(x+w/2, y+h/2);
22       this.world.Query(aabb, shapes, max);
23       if(shapes.length > 0){
24           for(var i=0; i<shapes.length; i++){
25               if(shapes[i].TestPoint(new b2Vec2(x, y))){
26                    gameObjects[gameObjects.length] = shapes[i].m_body.m_userData;
27               }
28           }
29       }
30       return gameObjects;
31   }
32
33  /**
34   * Adicona um objeto na camada.
35   *
36   * @author Marcos Harbs
37   * @method addGameObject
38   * @param {GameObject} gameObject
39   */
40   Layer.prototype.addGameObject = function(gameObject){
41       this.listGameObjects = ArrayUtils.addElement(this.listGameObjects,
42                                                    gameObject);
43       gameObject.layer = this;
44       if(Game.running){
45          gameObject.recreateBody = true;
46       }
47   }
48
49   /**
50    * Remove um objeto da camada.
51    *
52    * @author Marcos Harbs
53    * @method removeGameObject
54    * @param {GameObject} gameObject
55    */
56    Layer.prototype.removeGameObject = function(gameObject){
57        this.listGameObjects = ArrayUtils.removeElement(this.listGameObjects,
58                                                        gameObject);
59    }
```

Todo objeto representado no jogo que possuir um corpo e uma forma, deve ser um `GameObject`. Este objeto abstrai o que existe de comum entre todos os objetos do mundo. Por exemplo, armazena a localização do objeto no mundo, os componentes associados a este objeto e também possui um atributo `body` do tipo `b2Body`. O `b2Body` pertence à biblioteca

Box2DJS, e representa um corpo físico que será associado ao mundo físico da camada a qual o `GameObject` está inserido. Para criação deste corpo todos os objetos que representem um `GameObject` (herdam da classe `GameObject`) devem sobrescrever o método `createBodyShape`. O motor já possui três `GameObject` criados, que são: `BoxObject`, `CircleObject` e `PolygonObject`. Estes objetos implementam o método `createBodyShape` para criar `GameObject` que representem respectivamente retângulos, círculos e polígonos. A implementação do corpo físico de cada um pode ser vista no Quadro 4. O objeto `GameObject` também possui métodos para realizar a movimentação do objeto do jogo no mundo e para configurar uma velocidade de deslocamento.

Quadro 4 – Implementações do método `createBodyShape`

```
1  //Implementação da classe BoxObject.
2  BoxObject.prototype.createBodyShape = function(){
3            var shape = new b2BoxDef();
4            var xb = this.getWidth();
5            var yb = this.getHeight();
6            var scale = ComponentUtils.getComponent(this, "SCALE_COMPONENT");
7            if(scale){
8                      xb *= Math.abs(scale.scalePoint.x);
9                      yb *= Math.abs(scale.scalePoint.y);
10           }
11           shape.extents.Set(xb/2, yb/2);
12           return shape;
13 }
14
15 //Implementação da classe CircleObject.
16 CircleObject.prototype.createBodyShape = function(){
17           var shape = new b2CircleDef();
18           var rb = this.radius;
19           var scale = ComponentUtils.getComponent(this, "SCALE_COMPONENT");
20           if(scale){
21                     rb = this.radius * Math.abs(scale.scalePoint.x);
22           }
23           shape.radius = rb;
24           return shape;
25 }
26
27 //Implementação da classe PolygonObject.
28 PolygonObject.prototype.createBodyShape = function(){
29           var shape = new b2PolyDef();
30           shape.vertexCount = this.points.length;
31           var scale = ComponentUtils.getComponent(this, "SCALE_COMPONENT");
32           for(var i=0; i<shape.vertexCount; i++){
33                     var point = this.points[i];
34                     if(scale){
35                               shape.vertices[i].Set(point.x *
36                                                   Math.abs(scale.scalePoint.x),
37                                                   point.y *
38                                                   Math.abs(scale.scalePoint.y));
39                     }else{
40                               shape.vertices[i].Set(point.x, point.y);
41                     }
42           }
43           return shape;
44 }
```

Em um jogo os objetos possuem comportamentos como pular, andar, atirar, entre outros. Estes comportamentos são representados no motor através de componentes. Um componente é um objeto que herda de `Component`. Os componentes podem ser adicionados ou removidos a qualquer momento pelo usuário. Para realizar as operações de adição ou remoção

de componentes a um objeto deve-se usar os métodos utilitários do objeto `ComponentUtils`. Os componentes funcionam através de eventos. Eles possuem vários métodos que são chamados em diversas fases pelo motor e estes métodos, quando chamados, atuam sobre o objeto o qual o componente está associado. Um componente pode ser associado a um `Game`, `GameObject`, `Scene` ou `Layer`. No Quadro 5 pode-se conferir os eventos que podem ser implementados por um `Component`. O motor já possui vários componentes criados que abstraem comportamentos de física de corpos rígidos, desenho de objetos, transformações e animações (seção 3.2.2.1).

Quadro 5 – Eventos do objeto `Component`

```
1  // Callback chamado quando algum componente enviar uma mensagem para este.
2  Component.prototype.onReceiveMessage = function(message, extras){}
3
4  // Callback chamado quando o usuário apertar uma tecla.
5  Component.prototype.onKeyDown = function(keyCode){}
6
7  //Callback chamado quando o usuário soltar uma tecla.
8  Component.prototype.onKeyUp = function(keyCode){}
9
10 //Callback chamado quando o usuário clicar com o mouse.
11 Component.prototype.onClick = function(x, y, wich){}
12
13 //Callback chamado quando o usuário pressionar o mouse.
14 Component.prototype.onMouseDown = function(x, y, wich){}
15
16 //Callback chamado quando o usuário soltar o mouse.
17 Component.prototype.onMouseUp = function(x, y, wich){}
18
19 //Callback chamado quando o usuário mover o mouse.
20 Component.prototype.onMouseMove = function(x, y){}
21
22 //Callback chamado antes do objeto ser renderizado.
23 Component.prototype.onBeforeRender = function(context){}
24
25 //Callback chamado quando o objeto for renderizado.
26 Component.prototype.onRender = function(context){}
27
28 //Callback chamado quando o objeto for atualizado.
29 Component.prototype.onUpdate = function(delta){}
30
31 //Callback chamado quando o objeto colidir com outro objeto.
32 Component.prototype.onCollide = function(otherGameObject){}
33
34 //Callback chamado quando o component é carregado.
35 Component.prototype.onLoad = function(){}
36
37 //Callback chamado quando o component é destruído.
38 Component.prototype.onDestroy = function(){}
```

Um `Game` deve ter sempre associado a ele um objeto `Camera`. Uma câmera representa a maneira com que o jogador está visualizando o jogo. O objeto `Camera` tem associado a ele três componentes: `TranslateComponent`, `ScaleComponent` e `RotateComponent`, usados para realizar respectivamente a translação, escala e rotação da câmera. Os atributos destes componentes podem ser modificados para mudar o jeito de visualizar a cena atual sem modificar a posição dos objetos no mundo. Porém, ao aplicar uma transformação na câmera do jogo, a posição do mouse em cima do *canvas* não representa a mesma posição do objeto no mundo. Isto se mostrou um problema para fazer a seleção de objetos através do mouse. Para

resolver este problema foram aplicadas todas as transformações feitas na câmera também nas coordenadas do mouse, porém de maneira inversa. Para realizar estas transformações criou-se um método chamado `getNormalizedCoordinate` no objeto `MouseSystem`, como pode ser visto no Quadro 6. Para a câmera se tornar ativa ela deve ser configurada no atributo `camera` do objeto `Game`.

Quadro 6 – Normalização de coordenada do *mouse*

```
1  /**
2  * Método que normaliza a coordenada com base nas
3  * transformações da câmera setada no jogo.
4  *
5  * @author Marcos Harbs
6  * @method getNormalizedCoordinate
7  * @static
8  * @param {Float} x
9  * @param {Float} y
10 * @return {Point2D} normalizedCoordinate
11 */
12 this.getNormalizedCoordinate = function(x, y){
13     //aplica o deslocamento inverso
14     x -= -Game.camera.centerPoint.x+(Game.canvas.width/2);
15     y -= -Game.camera.centerPoint.y+(Game.canvas.height/2);
16 
17     //aplica a escala inversa
18     x -= Game.camera.centerPoint.x;
19     x /= Game.camera.scale.scalePoint.x;
20     x += Game.camera.centerPoint.x;
21 
22     y -= Game.camera.centerPoint.y;
23     y /= Game.camera.scale.scalePoint.y;
24     y += Game.camera.centerPoint.y;
25 
26     //aplica a translação inversa
27     x -= Game.camera.translate.translatePoint.x;
28     y -= Game.camera.translate.translatePoint.y;
29 
30     //aplica a rotação inversa
31     var sine = Math.sin(-Game.camera.rotate.angle);
32     var cosine = Math.cos(-Game.camera.rotate.angle);
33     x -= Game.camera.centerPoint.x;
34     y -= Game.camera.centerPoint.y;
35     var xn = x * cosine - y * sine;
36     var yn = x * sine + y * cosine;
37     x = xn + Game.camera.centerPoint.x;
38     y = yn + Game.camera.centerPoint.y;
39 
40     //Retorna a coordenada normalizada
41     return new Point2D().initialize(x, y);
42 }
```

Um jogo também deve permitir que o jogador interaja com os objetos do jogo através dos dispositivos de entrada. Para abstrair esta parte de integração entre o motor e os dispositivos criou-se uma série de classes `system` (`MouseSystem`, `KeySystem`, `KinectSystem` e `GamepadSystem`). Porém existem duas classes `system` que não fazem comunicação com dispositivos externos. Uma delas é a `LogicSystem`. Esta classe guarda as informações de colisões obtidas da Box2DJS e propaga estas colisões para serem tratadas pelos componentes. A outra classe é a `RenderSystem`, que realiza o desenho de todos os objetos e propaga os eventos de `onBeforeRender` e `onRender` para todos os componentes. A classe `RenderSystem` também gerencia e cuida para que as transformações de um objeto na hora de desenhar não

impactem em outro objeto. Isto é feito através dos métodos `save` e `restore` do `context`. O método `save` é chamado sempre antes de chamar o evento de `onRender` dos componentes de um objeto. O método `restore` é chamado ao final após todos os componentes do objeto terem sido processados fazendo o contexto do *canvas* voltar para o estado que estava no momento em que foi chamado o método `save`. Isto isola o desenho de cada objeto do jogo fazendo com que as transformações aplicadas no *canvas* dentro deste bloco não sejam aplicadas nos outros objetos do jogo.

3.3.3 O editor de jogos

O editor de jogos consiste de uma ferramenta web desenvolvida para facilitar a criação de jogos utilizando o motor criado. O editor necessita do motor porém o inverso não é verdade. É possível usar o motor criado sem a utilização do editor web.

Para criação do editor foi utilizada a tecnologia JSF e também o padrão MVC. As classes do pacote `managedbean` correspondem às classes da camada *View*. Estas classes atendem as requisições feitas pelo usuário nas telas web da ferramenta. As classes do pacote `controller` correspondem às classes da camada *Controller*. Estas classes possuem a lógica da ferramenta. São nelas que são implementadas todas as funcionalidades do editor. As classes do pacote `model` correspondem às classes da camada *Model*. Estas classes são *Plain Old Java Objects* (POJO), classes simples que possuem apenas atributos e métodos *get* e *set*. Elas são usadas para representar entidades do sistema.

O editor utiliza o motor criado. Ele apenas fornece uma *interface* gráfica para facilitar o uso do motor. Sendo assim ele suporta todas as funcionalidade citadas anteriormente do motor de jogos. Duas funcionalidades que o editor possui a mais é a de salvar e carregar um projeto. Para isso ele utiliza um arquivo no fomato JSON. Um exemplo deste aquivo pode ser visto no Quadro 7. Cada bloco delimitado por abre e fecha chaves do arquivo representa um nó da árvore de projetos do editor. Os atributos `id` e `idParent` são usados para identificar o nó e o nó pai. Dessa maniera o editor consegue recriar a hierarquia da árvore na importação do arquivo.

Quadro 7 – Exemplo de arquivo gerado pelo editor

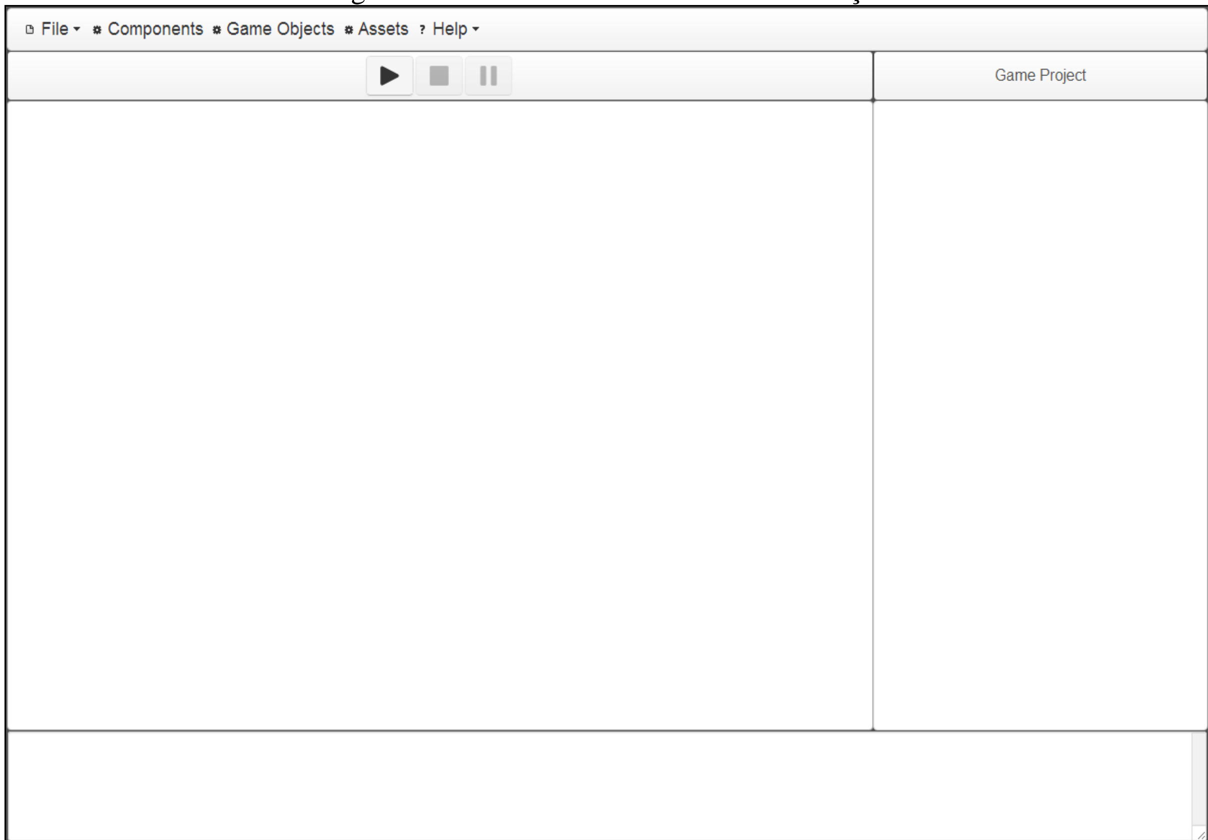
```
1  [
2    {
3      "id": "3d4027b5-b9a0-4bd0-bc6d-6737595b142c",
4      "type": "GAME",
5      "displayNodeName": "Game",
6      "jsVarName": "Game",
7      "jsInitialFunction": "Game.init(document.getElementById(\"gameCanvas\"), scene);",
8      "jsCode": "Game.camera.centerPoint.x \u003d 500; Game.camera.centerPoint.y \u003d 500;"
9    },
10   {
11     "id": "3b696434-86ef-4b54-9637-3028e0d0bc9e",
12     "idParent": "3d4027b5-b9a0-4bd0-bc6d-6737595b142c",
13     "type": "SCENE",
14     "displayNodeName": "Scene",
15     "jsVarName": "scene",
16     "jsInitialFunction": "new Scene().initialize(-1000, -1000, 1000, 1000);",
17     "jsCode": ""
18   },
19   {
20     "id": "df2d4c32-cf78-43bb-bfa2-76be6c032284",
21     "idParent": "3b696434-86ef-4b54-9637-3028e0d0bc9e",
22     "type": "LAYER",
23     "displayNodeName": "Layer",
24     "jsVarName": "layerBack",
25     "jsInitialFunction": "new Layer().initialize();",
26     "jsCode": ""
27   },
28   {
29     "id": "b72c260a-b1a1-4547-9a7f-f007edee5d33",
30     "idParent": "df2d4c32-cf78-43bb-bfa2-76be6c032284",
31     "type": "GAME_OBJECT",
32     "displayNodeName": "BoxObject",
33     "jsVarName": "background",
34     "jsInitialFunction": "new BoxObject().initialize(500, 500, 1280, 720, \"black\",
35 \"black\");",
36     "jsCode": "ComponentUtils.removeComponent(background,
37 ComponentUtils.getComponent(background, \"BOX_RENDER_COMPONENT\"));"
38   },
39   {
40     "id": "baf4cfff-149f-4482-b36b-1380b47f49a7",
41     "idParent": "b72c260a-b1a1-4547-9a7f-f007edee5d33",
42     "type": "COMPONENT",
43     "displayNodeName": "ImageRenderComponent",
44     "jsVarName": "backgroundImage",
45     "jsInitialFunction": "new
46 ImageRenderComponent().initialize(AssetStore.getAsset(\"space_background\").getAssetInstance(),
47 false, \"HORIZONTAL\");",
48     "jsCode": ""
49   }
50 ]
```

3.3.4 Operacionalidade da implementação

Esta seção apresenta um manual de uso do motor e do editor de jogos desenvolvido. Essas instruções permitem que o desenvolvedor inicie uma aplicação com os recursos disponíveis no motor de jogos. Para ilustrar as funcionalidades, foi desenvolvido um protótipo de um jogo Tangram.

A primeira coisa a ser feita é acessar a ferramenta web de edição do jogo. Ao acessá-la o usuário irá visualizar uma tela como na Figura 22.

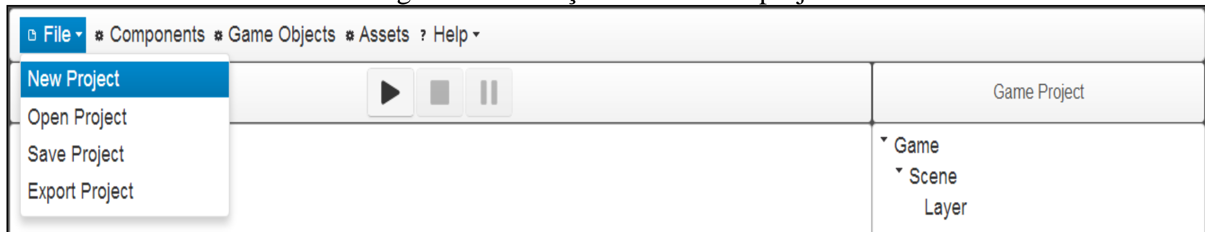
Figura 22 – Tela inicial da ferramenta de edição



A tela inicial da ferramenta de edição é dividida em uma área para menu com ações que o usuário pode realizar como: criar novo projeto, abrir projeto, salvar projeto, exportar projeto, gerenciar os componentes, gerenciar os objetos do jogo, gerenciar os *assets* e acessar a documentação. Abaixo existe uma área onde o jogo será executado e com três botões para o usuários poder executar, pausar ou parar o jogo. A área a direita é reservada para exibir uma estrutura em forma de árvore que representa o jogo sendo desenvolvido. A área abaixo é uma área de mensagens onde serão exibidos erros e sucessos de execuções para o usuário.

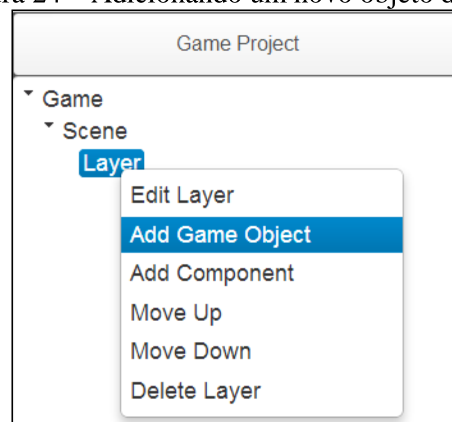
O segundo passo no processo de criação do jogo Tangram consiste em criar um projeto. Para isso o usuário precisa clicar na opção do menu `New Project` e então a ferramenta irá criar a estrutura básica para um novo jogo como pode ser visto na Figura 23.

Figura 23 – Criação de um novo projeto



O terceiro passo consiste na criação das peças do Tangram. Ele é formado por sete peças, então na ferramenta deve-se adicionar sete `GameObject` do tipo `PolygonObject`. A Figura 24 ilustra o procedimento para adicionar um novo objeto do jogo a uma camada.

Figura 24 – Adicionando um novo objeto do jogo



O sistema irá apresentar a tela (Figura 25) para criação de uma instância de `GameObject` no jogo. O usuário deve selecionar o `GameObject` pressionando o botão `Select` da opção `PolygonObject`. Desta forma o sistema irá preencher a função de inicialização cadastrada para este tipo de objeto do jogo. O usuário deve substituir os parâmetros na função de inicialização e preencher o campo `Variable Name` e pressionar o botão `Save`. Desse modo o sistema irá criar um novo nó na árvore de projetos representando a nova instância recém criada. A Figura 25 ilustra o resultado deste procedimento.

Figura 25 – Criação de uma instância de um objeto do jogo

Game Object Instance ×

Game Objects

Name	
PolygonObject.js	Select
SpaceShipObject.js	Select

Variable Name *

poligono1

Initial Function *

```
new PolygonObject().initialize(430, 200,
                               [new Point2D().initialize(-50, 100),
                                new Point2D().initialize(-50, -100),
                                new Point2D().initialize(50, 0)],
                               "purple", "black");
```

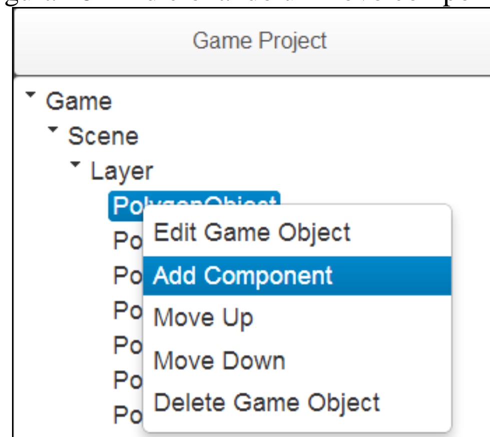
Code

Save

O processo descrito acima deve ser repetido sete vezes para criar as sete peças do Tangram, lembrando que os valores dos vértices e da posição das peças devem ser alterados para formar as sete peças.

O quarto passo consiste em fazer com que as peças sofram ação da física. Para isso, deve-se adicionar a cada peça criada anteriormente um componente chamado `RigidBodyComponent`. A Figura 26 mostra como adicionar um componente a um objeto.

Figura 26 – Adicionando um novo componente



O sistema irá apresentar a tela (Figura 27) para criação de uma instância de `Component` no jogo. O usuário deve selecionar na tabela o botão `Select` de `RigidBodyComponent`, informar o campo `Variable Name`, substituir os parâmetros na função de inicialização no campo `Initial Function` e pressionar o botão `Save`. O sistema então irá criar um novo nó na árvore representando esta nova instância do componente adicionado. Agora a peça irá responder aos eventos de física e será tratada como um corpo rígido pelo motor do jogo. Este procedimento pode ser visualizado na Figura 27 e deve ser repetido para cada peça do Tangram criada.

Figura 27 – Criação de uma nova instância de um componente

Component Instance ×

Components

Name	
RigidBodyComponent.js	Select
MoveCameraComponent.js	Select

Variable Name *

fisicaPoligono1

Initial Function *

new RigidBodyComponent().initialize(0,1,false,false,0.2);

Code

Save

O quinto passo é criar um componente que se registre nos eventos de mouse para permitir selecionar e movimentar as peças. Para criar um novo componente deve-se acessar a opção do menu `Components`. O sistema irá abrir uma tela com a listagem de todos os componentes existentes, então deve-se pressionar o botão `New`. O sistema abrirá a tela de criação de componente. Deve-se informar no campo `Name` o nome do componente que será `MoveTangramComponent`. No campo `Init Function` deve-se informar a função de inicialização e no campo `Code` deve-se informar o código fonte deste componente. O código fonte completo pode ser visto no Quadro 8. A Figura 28 mostra a tela de criação do componente.

Figura 28 – Tela de criação de um componente

Edit Component ✖

Name *

MoveTangramComponent

Init Function *

```
new MoveTangramComponent().initialize();
```

Code

```
function MoveTangramComponent(){}
MoveTangramComponent.prototype = new Component();
MoveTangramComponent.prototype.selectedPiece = null;
MoveTangramComponent.prototype.startX = 0;
MoveTangramComponent.prototype.startY = 0;
MoveTangramComponent.prototype.onMouseDown = function(x, y, wich){
  var point = MouseSystem.getNormalizedCoordinate(x, y);
  x = point.x;
  y = point.y;

  this.startX = x;
  this.startY = y;

  var selected = layer.queryGameObjects(x, y, 2, 2, 20);

  this.selectedPiece = selected[0] || null;

    if(!this.selectedPiece == null &&
      ComponentUtils.getComponent(this.selectedPiece, "RIGID_BODY_COMPONENT").density == 0){
        this.selectedPiece = null;
  }
 }
 MoveTangramComponent.prototype.onMouseUp = function(x, y, wich){
  this.startX = 0;
  this.startY = 0;
```

Save

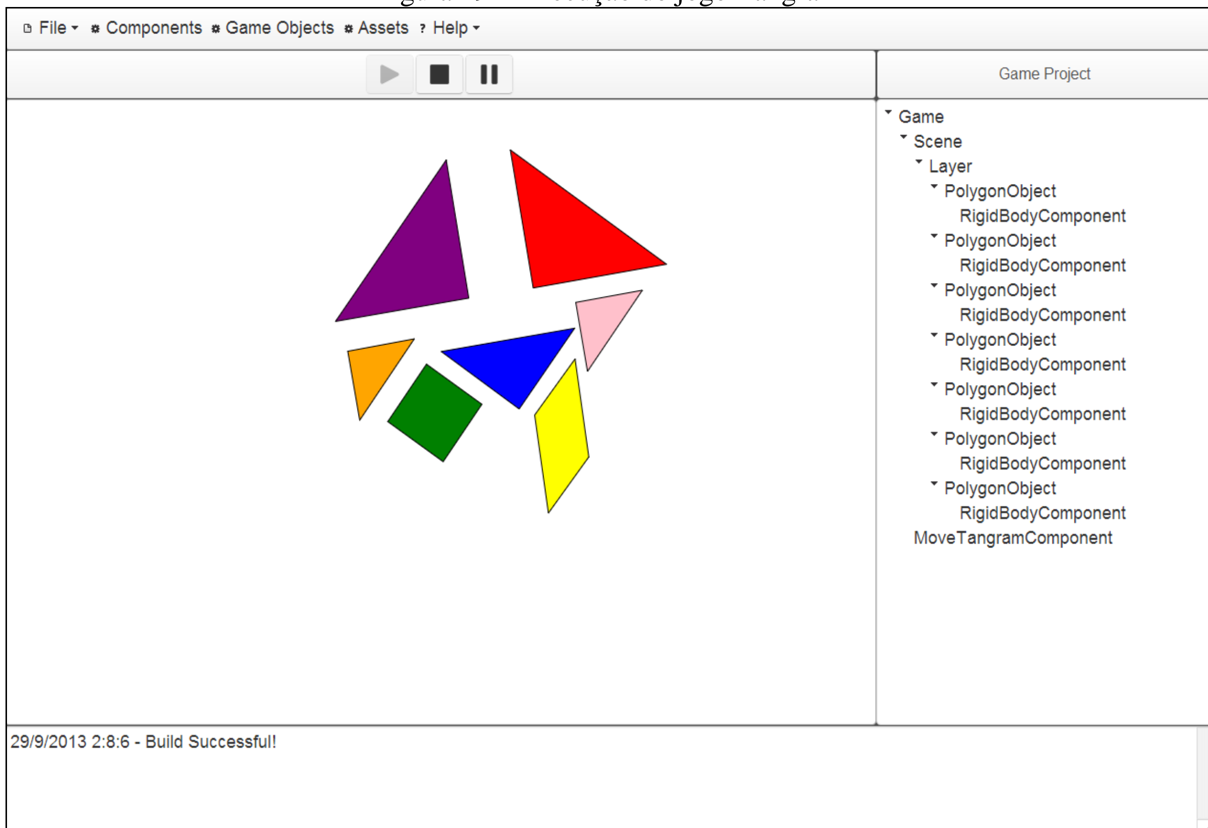
Quadro 8 – Código fonte de `MoveTangramComponent`

```
function MoveTangramComponent(){}
MoveTangramComponent.prototype = new Component();
MoveTangramComponent.prototype.selectedPiece = null;
MoveTangramComponent.prototype.startX = 0;
MoveTangramComponent.prototype.startY = 0;
MoveTangramComponent.prototype.onMouseDown = function(x, y, wich){
   var point = MouseSystem.getNormalizedCoordinate(x, y);
   x = point.x;
   y = point.y;

   this.startX = x;
   this.startY = y;

   var selected = layer.queryGameObjects(x, y, 2, 2, 20);

   this.selectedPiece = selected[0] || null;
 }
 MoveTangramComponent.prototype.onMouseUp = function(x, y, wich){
   this.startX = 0;
   this.startY = 0;
   this.selectedPiece = null;
 }
 MoveTangramComponent.prototype.onMouseMove = function(x, y, wich){
   if(this.selectedPiece != null){
     var point = MouseSystem.getNormalizedCoordinate(x, y);
     x = point.x;
     y = point.y;

     //movimentação
     var dx = (x - this.startX);
     var dy = (y - this.startY);

     this.selectedPiece.addMove(dx, dy);

     this.startX = x;
     this.startY = y;
   }
 }
 MoveTangramComponent.prototype.onKeyDown = function(keyCode){
   if(keyCode == 109){
     if(this.selectedPiece != null){
       var rotate = ComponentUtils.getComponent(this.selectedPiece, "ROTATE_COMPONENT");
       var angle = rotate.getAngle() + (1/60);
       rotate.setRotate(angle);
     }
   }else if(keyCode == 107){
     if(this.selectedPiece != null){
       var rotate = ComponentUtils.getComponent(this.selectedPiece, "ROTATE_COMPONENT");
       var angle = rotate.getAngle() - (1/60);
       rotate.setRotate(angle);
     }
   }
 }
 MoveTangramComponent.prototype.getSystems = function(){
   var systems = new Array();
   systems = ArrayUtils.addElement(systems, KeySystem.getTag());
   systems = ArrayUtils.addElement(systems, MouseSystem.getTag());
   return systems;
 }
 MoveTangramComponent.prototype.getTag = function(){
   return "MOVE_TANGRAM_COMPONENT";
 }
```

Depois de criado o componente, deve-se adicionar uma instância dele da mesma maneira que foi adicionado o `RigidBodyComponent` a um objeto do jogo, porém ao invés de adicionar a um objeto do jogo, este componente deve ser adicionado diretamente ao nó da árvore denominado de `Game`.

O último passo é rodar o jogo. Para isso basta pressionar o botão de `Play` do editor e o jogo será carregado e desenhado na tela como pode ser visto na Figura 29.

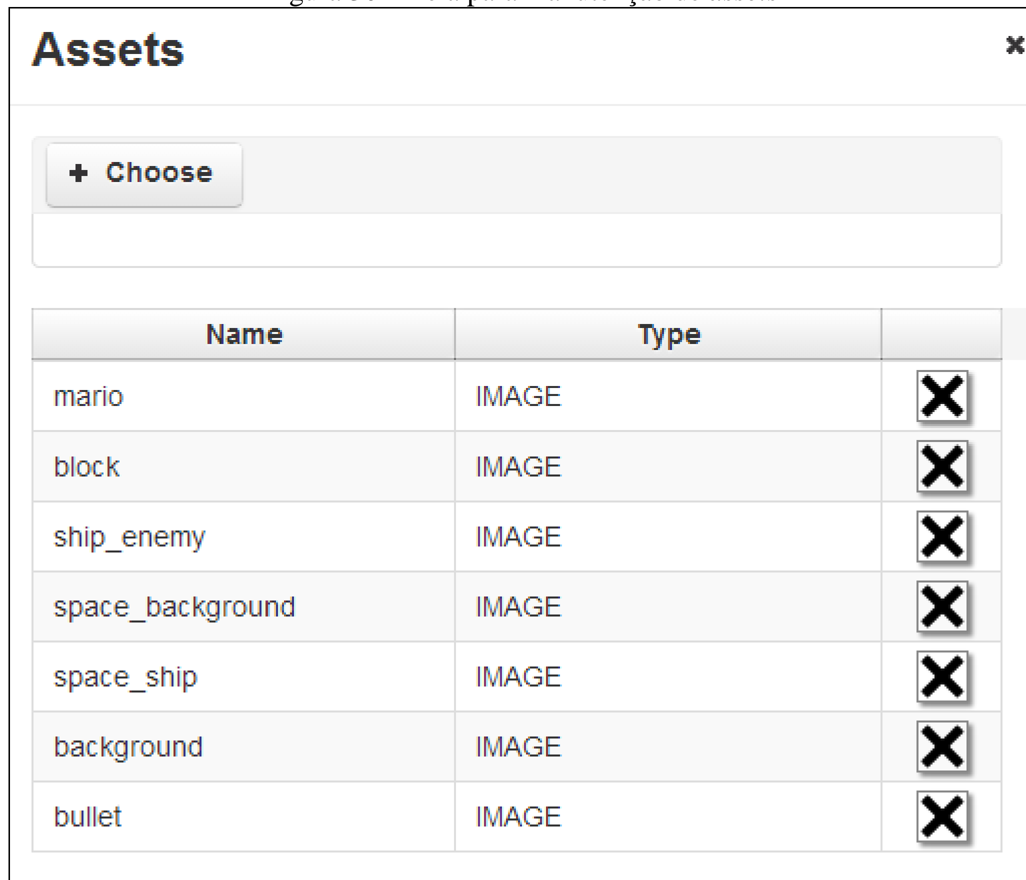
Figura 29 – Execução do jogo Tangram



O editor ainda possui outras funcionalidades que podem ser acessadas a partir do menu `File` como pode ser visto na Figura 23. Quando o usuário aciona a opção `Save Project` o editor irá gerar um arquivo JSON com todos os nós da árvore de projeto e disponibilizar este arquivo para o usuário. Quando o usuário aciona a opção `Open Project` o editor irá permitir que o usuário informe um arquivo de formato JSON que contenham um projeto salvo anteriormente e irá recriar com base neste arquivo a árvore de projeto. Quando o usuário acionar a opção `Export Project` o editor irá gerar toda a estrutura necessária para que o jogo funcione sem a necessidade do editor, empacotar esta estrutura em um arquivo compactado e disponibilizar este arquivo para o usuário.

O editor permite que o usuário adicione, consulte e exclua *assets*. Este gerenciamento é feito na tela de manutenção de *assets* que pode ser visto na Figura 30.

Figura 30 – Tela para manutenção de *assets*



O editor também permite que o usuário acesse a documentação do motor de jogos. Para isto é necessário acionar a opção `Documentation` dentro do menu `About`. Será aberta uma nova aba no navegador do usuário com uma página HTML contendo a listagem de todas as classes do motor de jogos, podendo visualizar a documentação criada para os métodos e atributos de cada uma (Figura 31).

Figura 31 – Tela de documentação do motor

3.4 RESULTADOS E DISCUSSÃO

O presente trabalho teve como objetivo desenvolver um motor de jogos em HTML5 e Javascript, com uma arquitetura orientada a componentes que facilitasse a reusabilidade de código e permitisse um baixo acoplamento. Também foi desenvolvido um editor de jogo. Este editor é uma ferramenta web desenvolvida em Java que utiliza o motor criado e oferece uma interface mais simples para se criar um jogo.

O modelo de desenvolvimento seguido na implementação se assemelhou ao processo incremental da engenharia de software: cada objetivo representava um marco a ser atingido, recebendo prioridade os que eram dependência de outros objetivos. Primeiramente foram desenvolvidas as classes utilitárias. Após isso foi desenvolvido o núcleo da arquitetura orientada a componente. Então foi desenvolvido a câmera, acesso a recursos como imagens, os objetos do jogo e a classe para gerenciar a execução do jogo. Após implementada e testada a parte do motor, foi desenvolvido o editor seguindo o mesmo processo incremental.

Houve modificações nos objetivos iniciais. Primeiramente o editor seria uma aplicação web que rodaria localmente sem necessidade de um servidor. Mas identificou-se um problema de permissão para salvar arquivos no cliente. Então o editor foi criado urtilizando JSF e

rodando em um servidor JBoss o que permitiu a criação de novas funcionalidades, como: salvar *scripts* de novos componentes, exportar o jogo, salvar e recuperar o projeto atual. Inicialmente não estava previsto o desenvolvimento da parte de física do motor. Essa parte se mostrou muito complexa e custosa de se implementar e por isso foi integrado ao motor de jogos o motor de física Box2D. Esta decisão permitiu que o motor contemplasse várias funcionalidades na parte de física de corpos rígidos.

Outra funcionalidade implementada além das previstas foi a comunicação do Kinect com o motor de jogos. Esta comunicação utilizou a biblioteca Zigfu e adicionou ao motor uma nova maneira do usuário interagir com o jogo. Além da integração do Kinect também foi implementada a utilização de *joysticks*, utilizando a Gamepad API.

Um dos requisitos que no início da implementação não estava-se conseguindo cumprir era o de desenhar uma cena com cinquenta objetos mantendo a taxa de quadros por segundo acima de trinta. Porém após serem realizados testes de performance foi possível identificar um gargalo de processamento, e, com alguns ajustes (descritos na próxima seção), atingiu-se este objetivo.

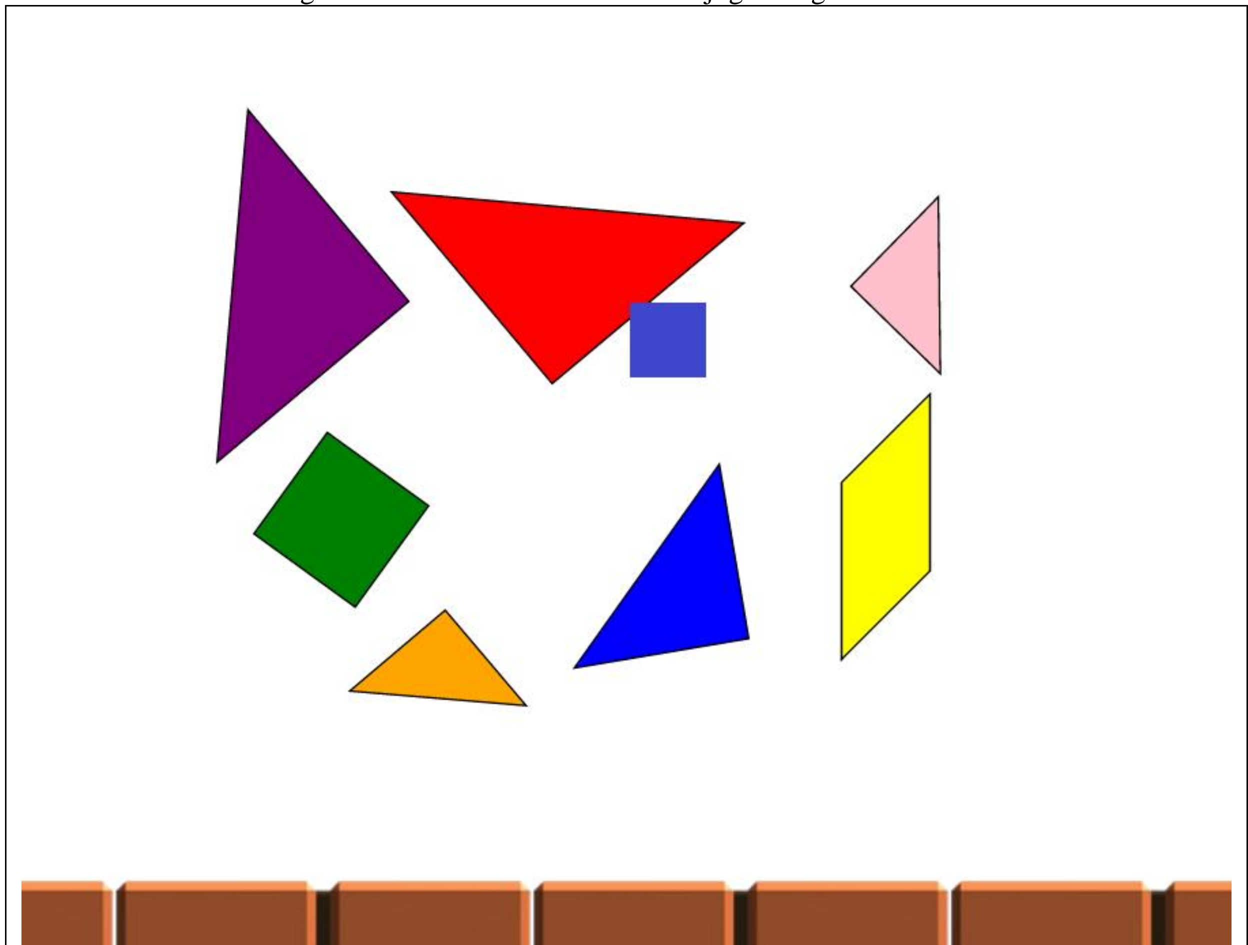
3.4.1 Testes de funcionalidades

Para testar as funcionalidades criadas no motor de jogos foram desenvolvidos dois jogos. Um dos jogos desenvolvido foi o Tangram (Seção 3.4.1.1), outro foi o Space Invaders (Seção 3.4.1.2). A seguir serão detalhados os dois cenários de testes.

3.4.1.1 Tangram

O Tangram foi utilizado como cenário de testes para as funcionalidades básicas do motor como: tratamento de colisão, simulação de gravidade, rotação de objetos, renderização dos objetos da cena, movimentação de câmera e seleção de objetos. Neste cenário também foi testada a integração com o Kinect. O jogador pode selecionar e movimentar as peças, porém, não pode rotaciona-las utilizando o Kinect. A funcionalidade de rotacionar é apenas permitida quando o jogador utilizado o teclado e o *mouse* como meio de interação. A Figura 32 representa o cenário de testes, o quadrado azul nela representa o cursor manipulado pelo Kinect.

Figura 32 – Cenário de testes com o jogo Tangram e Kinect



3.4.1.2 Space Invaders

O cenário de testes Space Invaders consiste de uma nave manipulada pelo jogador, que deve atirar e destruir as naves inimigas e se esquivar dos tiros inimigos. Estes cenário foi criado para testar algumas funcionalidades que não eram possíveis de se testar no cenário do Tangram, como: renderização de imagens, adicionar e remover objetos ao jogo em tempo de execução, renderização de animações e intergração com *joysticks*.

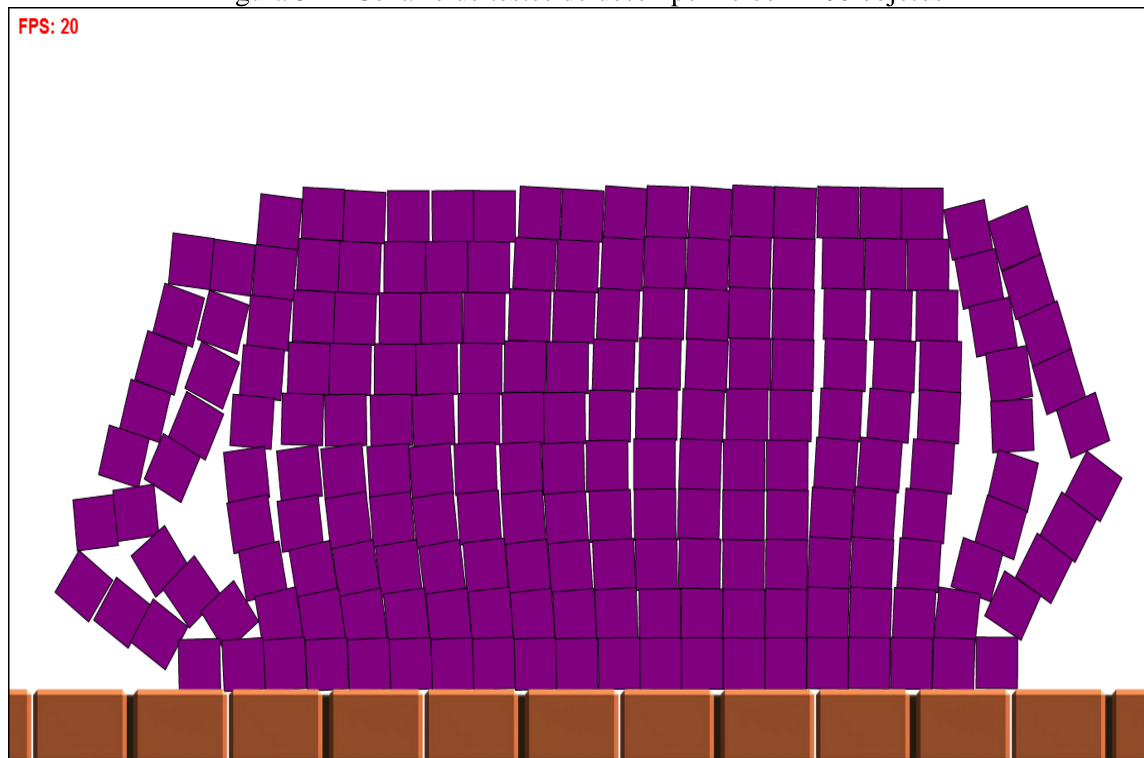
O jogador pode interagir com o jogo de duas maneiras. Através do teclado, ou através do *joystick*. É possível movimentar a nave em qualquer direção com a alavanca do *joystick* e atirar utilizando o botão `A`. Neste teste foi utilizado o *joystick* do XBox 360. Quando o tiro do jogador acerta uma nave inimiga é renderizada uma animação de explosão como pode ser visto na Figura 33.

Figura 33 – Cenário de testes com o jogo Space Invaders

3.4.2 Testes de desempenho

Para execução dos testes de desempenho foi utilizado um cenário (Figura 34) com uma variação no número de objetos de um até duzentos. Estes objetos não estão apenas sendo desenhados. Todos eles são corpos rígidos, que reagem a colisão com outros objetos.

Figura 34 – Cenário de testes de desempenho com 200 objetos



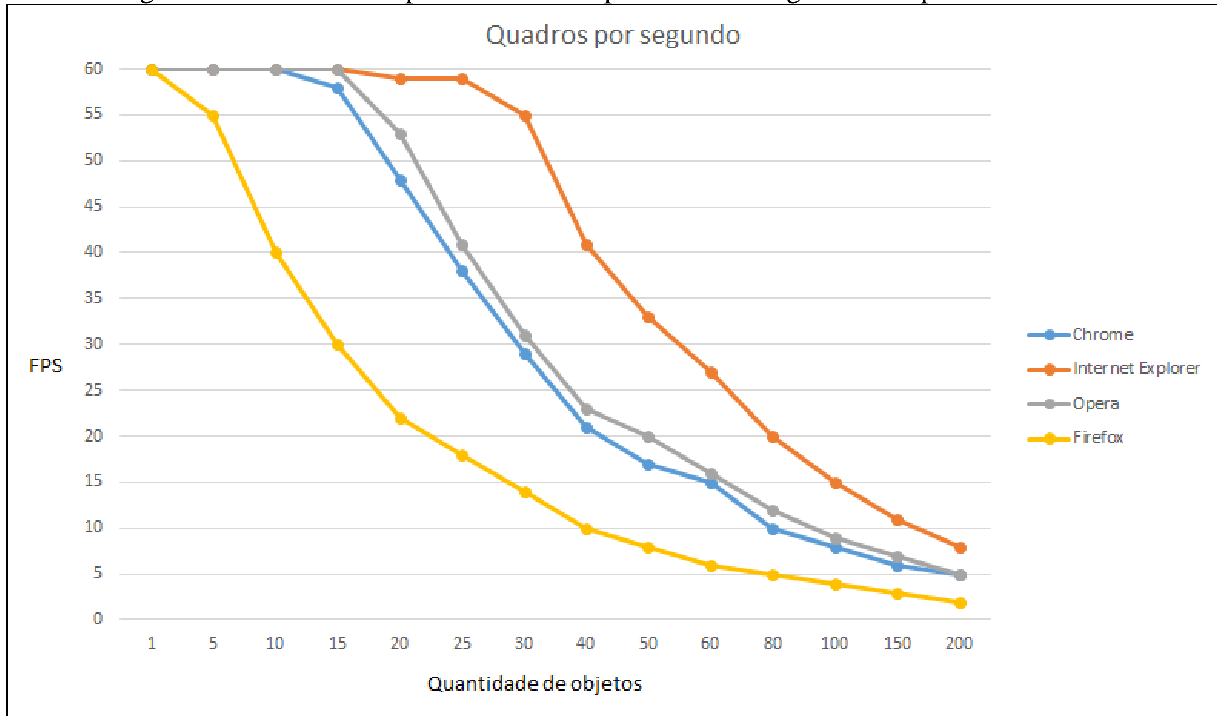
Todos os testes foram aplicados em quatro navegadores (os mais utilizados no mercado): GoogleChrome 32.0.1678.0 dev-m Aura, Internet Explorer 10.0.9200.16721, Opera 17.0 e Mozilla Firefox 25.0. Os dados coletados são a média de quadros por segundo da execução da cena. A Tabela 1 exibe os dados obtidos durante a primeira análise (antes de fazer os ajustes para melhorar o desempenho).

Tabela 1 – Quantidade de quadros por segundo por número de objetos na cena nos principais navegadores na primeira análise

Quantidade de objetos	Google Chrome	Internet Explorer	Opera	Mozilla Firefox
1	60	60	60	60
5	60	60	60	55
10	60	60	60	40
15	58	60	60	30
20	48	59	53	22
25	38	59	41	18
30	29	55	31	14
40	21	41	23	10
50	17	33	20	8
60	15	27	16	6
80	10	20	12	5
100	8	15	9	4
150	6	11	7	3
200	5	8	5	2

Como pode ser visto, até trinta objetos na cena os navegadores mantiveram uma média de trinta ou mais quadros por segundo com exceção do Mozilla Firefox que a partir de quinze objetos já começava a ter uma taxa abaixo de trinta quadros por segundo. A Figura 35 exibe o gráfico comparativo com os resultados obtidos.

Figura 35 – Gráfico comparando o desempenho dos navegadores na primeira análise



A partir dos dados obtidos percebeu-se que eram necessárias melhorias no desempenho, pois não estava-se cumprindo o requisito inicial de cinquenta objetos na cena com uma taxa de quadros por segundo acima de trinta.

Para realizar a análise de gargalos de desempenho utilizou-se a ferramenta Profiler, recurso disponível no Chrome Developer Tools (disponível junto com o Google Chrome). Ela permite analisar quais métodos estão tomando maior tempo de processamento e assim identificar possíveis melhorias como pode ser visto na Figura 36.

Figura 36 – Relatório de execução do *profile* da aplicação

Elements Network Sources Timeline Profiles Resources Audits Console

Profiles

CPU PROFILES

Profile 1

Self	Total	Function	
66.81%	66.81%	contains	ArrayUtils.js:140
9.78%	53.13%	fireCollideListener	LogicSystem.js:64
6.96%	6.96%	(program)	
5.89%	15.82%	fireRenderListener	RenderSystem.js:18
3.49%	18.41%	fireUpdateListener	LogicSystem.js:20
0.98%	0.98%	(garbage collector)	
0.88%	56.48%	stepGame	Game.js:145
0.67%	16.50%	renderGame	Game.js:161
0.34%	2.02%	b2World.Step	b2World.js:299
0.33%	18.74%	updateGame	Game.js:133
0.31%	0.31%	putCollideInfo	LogicSystem.js:94
0.30%	0.30%	(idle)	
0.28%	0.28%	translate	
0.24%	0.25%	b2ContactSolver.initialize	b2ContactSolver.js:26
0.23%	0.23%	fillRect	
0.20%	0.53%	b2Collision.b2CollidePoly	b2Collision.js:372
0.20%	0.20%	strokeRect	
0.20%	0.72%	Object.extend.Evaluate	b2PolyContact.js:79
0.16%	0.92%	b2Island.Solve	b2Island.js:144
0.15%	0.15%	restore	
0.14%	0.27%	Object.extend.Synchronize	b2PolyShape.js:316
0.10%	0.10%	b2Collision.EdgeSeparation	b2Collision.js:105
0.09%	0.09%	b2ContactSolver.SolveVelocityConstraints	b2ContactSolver.js:242
0.09%	0.09%	rotate	
0.08%	0.10%	b2BroadPhase.MoveProxy	b2BroadPhase.js:411
0.08%	0.08%	b2Collision.FindIncidentEdge	b2Collision.js:265
0.07%	0.45%	b2World.Step	b2World.js:299

Ao analisar o relatório gerado pelo Profiler, percebeu-se que um método utilitário que serve para ver se um elemento está contido em um conjunto estava consumindo 66,81% do processamento.

Cada classe *system* já citada anteriormente (ver seção 3.2.2.4) possui uma `String` que as identifica e cada componente da cena possui um método que retorna uma lista contendo quais classes *system* devem processar aquele componente. Isto foi feito para que um componente responsável apenas por desenhar não recebesse eventos de teclado por exemplo. Porém este teste é realizado toda vez que uma classe *system* vai chamar um méotdo *listener* do componente, e esta verificação acabou se tornando muito custosa.

Para resolver este problema foi tirada esta validação, assim um componente de desenho vai receber os eventos de teclado ou mouse, porém o método não estará implementado e nada será executado, o que se mostrou menos custoso do que realizar a validação.

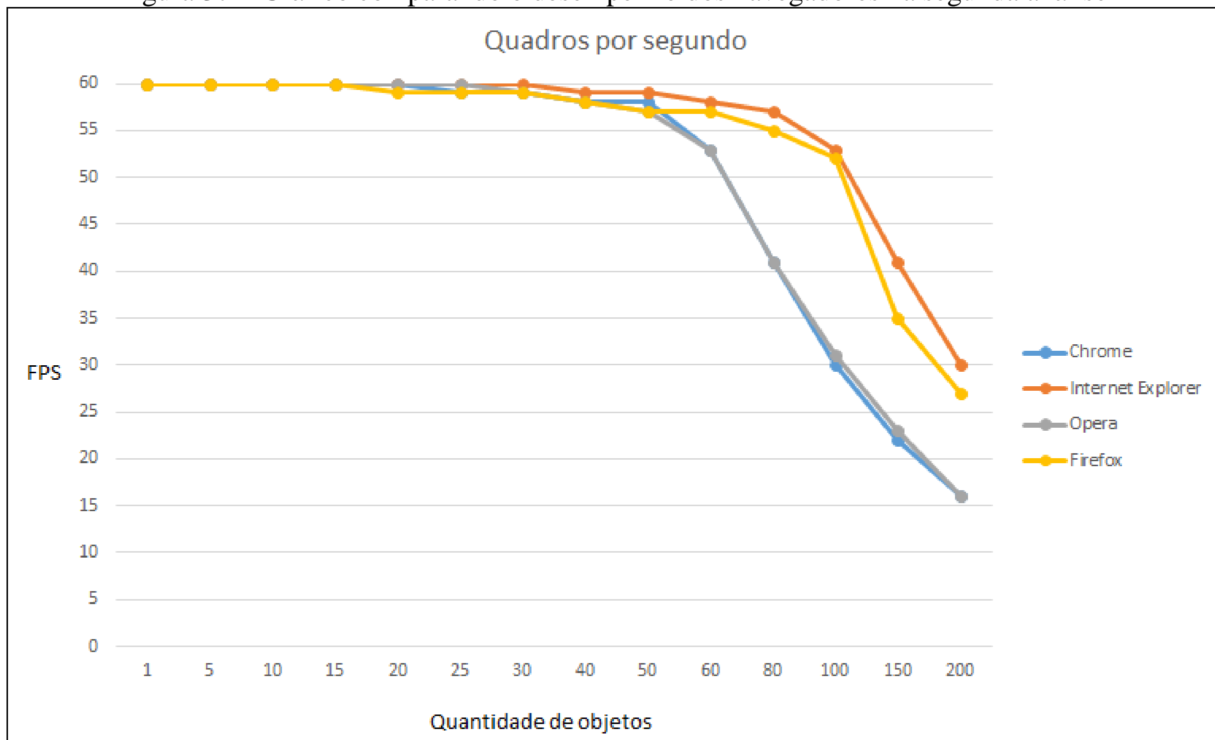
Com esta alteração os testes foram refeitos seguindo o mesmo cenário anterior. Os dados da nova coleta realizada em todos os navegadores pode ser vista na Tabela 2.

Tabela 2 – Quantidade de quadros por segundo por número de objetos na cena nos principais navegadores na segunda análise

Quantidade de objetos	Google Chrome	Internet Explorer	Opera	Mozilla Firefox
1	60	60	60	60
5	60	60	60	60
10	60	60	60	60
15	60	60	60	60
20	60	60	60	59
25	59	60	60	59
30	59	60	59	59
40	58	59	58	58
50	58	59	57	57
60	53	58	53	57
80	41	57	41	55
100	30	53	31	52
150	22	41	23	35
200	16	30	16	27

Ao analisar a segunda tabela é possível perceber que houve um ganho significativo em relação a primeira. O requisito de desempenho não só foi alcançado como superado, conseguindo-se manter a cena em uma média de trinta quadros por segundo (nos navegadores Google Chrome e Opera) com cem objetos ao invés de cinquenta como previa o objetivo inicial. Vale resaltar o desempenho acima da média do Internet Explorer que, mesmo com duzentos objetos na cena, conseguiu manter uma taxa de trinta quadros por segundo, se mostrando superior em desempenho aos outros navegadores analisados. A Figura 37 mostra o gráfico comparativo entre os navegadores.

Figura 37 – Gráfico comparando o desempenho dos navegadores na segunda análise

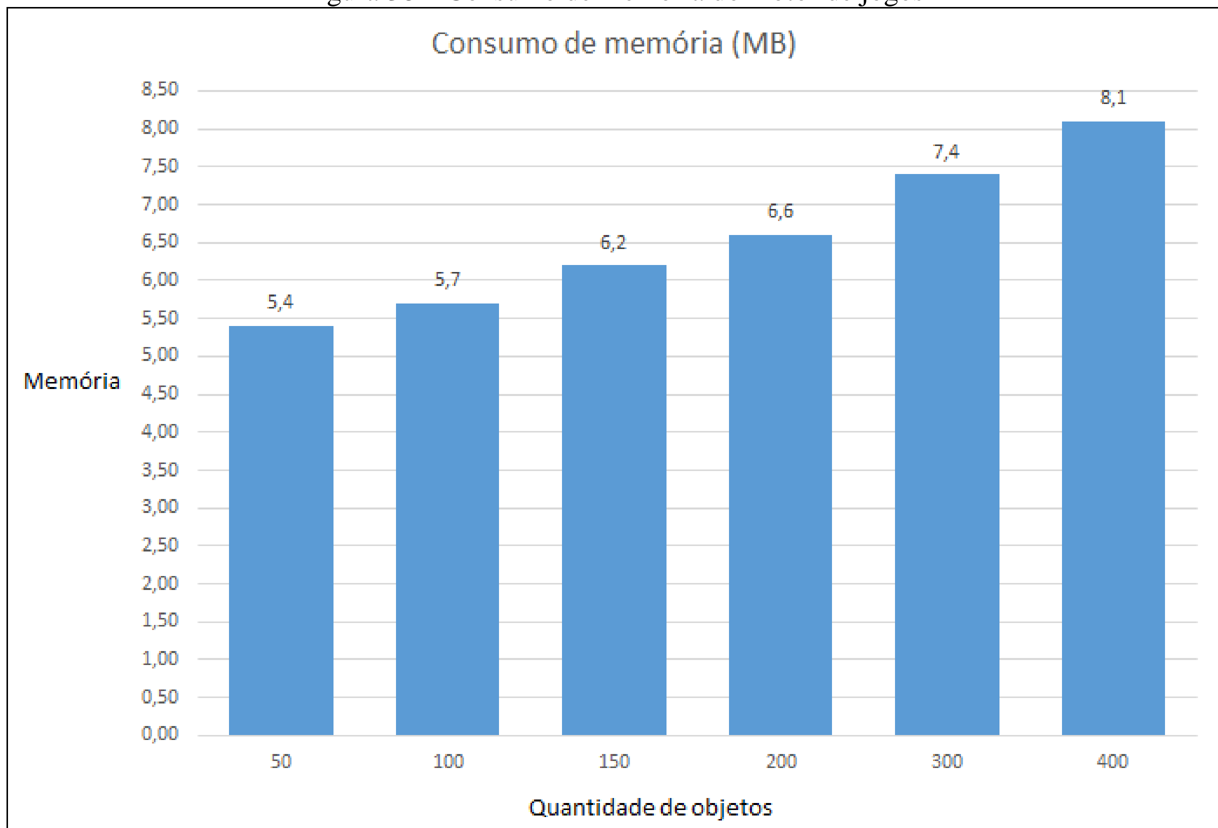


No gráfico fica claro a superioridade de desempenho do Intener Explorer em relação aos outros navegadores. Outra curiosidade é o desempenho muito similar do Opera e do Google Chrome. A explicação para isso é que o navegador Opera utiliza o Webkit e o Google V8 Javascript engine, que foram desenvolvidos pela Google, e é o mesmo motor Javascript utilizado no Google Chrome, justificando o desempenho muito similar entre os dois navegadores.

Realizou-se um teste também usando o navegador Google Chrome com a mesma cena mostrada anteriormente com duzentos objetos, porém estes objetos não eram corpos rígidos e não geravam colisões entre si. A média de quadros por segundo deste teste foi de 45 quadros por segundo ao invés dos 16 quadros por segundo do teste anterior onde os duzentos objetos eram corpos rígidos e geravam colisões a todo instante entre si.

Também foram realizados testes de consumo de memória utilizando novamente a ferramenta Profiler do Google Chrome. Não houve maiores problemas quanto ao consumo de memória que mostrou crescer de maneira linear em relação ao número de objetos na cena como pode ser visto na Figura 38.

Figura 38 – Consumo de memória do motor de jogos



3.4.3 Comparativo entre o trabalho desenvolvido e os correlatos

Nesta seção são apresentadas e discutidas as principais características deste trabalho, comparando-o com os trabalhos correlatos. Essa comparação pode ser vista no Quadro 9.

Quadro 9 – Características dos trabalhos correlatos e do trabalho desenvolvido

Características	Unity 3D	Impact	Motor desenvolvido
Suporte 2D		X	X
Suporte 3D	X		
Orientada a componentes	X		X
Suporte a física	X	X	X
Editor de cena	X	X	X
Ferramentas de debug	X	X	
Geração para dispositivos móveis	X	X	
Código fonte aberto			X
Sem necessidade de licença			X
Possui integração com Kinect			X
Possui integração com Joystick	X		X

O motor de jogos desenvolvido possui as principais funcionalidades dos trabalhos correlatos. Os dois trabalhos correlatos são aplicações comerciais já com um bom tempo de mercado, principalmente a Unity 3D, que é principalmente focada em jogo de três dimensões e possui muitas funcionalidades além das citadas. O Impact é uma ferramenta comercial porém não tão difundida como a Unity 3D. Ela é focada em jogos de duas dimensões em

HTML5, sendo a ferramenta que mais se assemelha ao trabalho desenvolvido. O motor de jogos desenvolvido atingiu as principais funcionalidades que a Impact possui. Apenas duas funcionalidades não foram contempladas que são: ferramentas para depuração do jogo e geração do jogo preparada para dispositivos móveis. Essas duas funcionalidades não foram originalmente propostas porém a geração para dispositivos móveis como se trata de HTML5 pode ser feita futuramente. Já as ferramentas de depuração cabe uma melhor análise de como elas devem ser desenvolvidas e quais as medidas que devem ser observadas, porém acredita-se não ter nenhum problema estender o código desenvolvido com estas novas funcionalidades.

A grande vantagem que o motor de jogos proporcionou é que ela fornece um meio de se trabalhar em duas dimensões com uma arquitetura orientada a componentes, pois a Unity 3D também possui uma arquitetura orientada a componentes mas focada em três dimensões. Já a Impact é focada em três dimensões mas não possui uma arquitetura orientada a componentes. A arquitetura orientada a componentes inspirada na Unity 3D se mostrou extremanete reutilizável, fornecendo um baixo acoplamento entre os objetos do jogo e permitindo um componente ser reutilizado em vários objetos, bem como em diferentes jogos, diminuindo o custo de desenvolvimento de futuros jogos.

4 CONCLUSÕES

Este trabalho apresentou o desenvolvimento de um motor de jogos utilizando HTML5 e Javascript, um editor web utilizando *Java Server Faces* e duas aplicações de teste (Tangram e Space Invaders).

O HTML5 é uma tecnologia recente e ainda continua em desenvolvimento, porém através deste trabalho ficou comprovado que, com o elemento *canvas* juntamente com a linguagem de programção Javascript, é possível desenvolver jogos e aplicações interativas.

Como nem todos os navegadores utilizam o mesmo motor Javascript e implementam a especificação do HTML5 da mesma maneira, acaba existindo diferença entre navegadores, principalmente no que diz respeito ao desempenho. Diante dos testes de desempenho feitos com a aplicação verificou-se que o navegador com melhor desempenho foi o Internet Explorer.

O desempenho do motor superou o proposto, conseguindo desenhar uma média de cem objetos e mantendo trinta quadros por segundo, lembrando que o esforço não é de apenas desenhar cem objetos, mas também de identificar e resolver as inúmeras colisões que estes objetos geram entre si a cada quadro. O desempenho foi ainda melhor quando os objetos não eram corpos rígidos e portanto não geram colisões, mantendo quarenta e cinco quadros por segundo.

Os objetivos propostos foram alcançados, mas para chegar a nível comercial e competir com os trabalhos correlatos ainda necessita de melhorias. A principal melhoria que deve ser feita para competir com os outros produtos é a de exportar o jogo para dispositivos móveis, já que hoje este mercado é um mercado muito procurado e em expansão. Além dos objetivos propostos ainda foram alcançados outros como a comunicação com o Kinect e *joysticks*. A integração com o Kinect necessita ser melhorada para conseguir uma melhor experiência do usuário com o jogo utilizando este sensor, explorando os recursos deste novo hardware para se ter interações diferentes no motor de jogos desenvolvido.

A arquitetura orientada a componentes do motor proposto se mostrou flexível e reutilizável, atingindo o objetivo proposto de construir um motor com baixo acoplamento entre as classes e permitir uma grande reusabilidade das funcionalidades criadas entre os objetos do jogo.

O editor web criado atingiu os objetivos propostos, porém ainda para utilizá-lo é necessário que o usuário receba algum treinamento e que tenha conhecimentos básico de

programção e Javascript, sendo necessário melhorias de usabilidade para um usuário de outra área (sem conhecimento técnico) usá-lo.

4.1 EXTENSÕES

Durante o desenvolvimento deste trabalho observou-se algumas possibilidades de extensões. São elas:

a) explorar mais os recursos que a Box2DJS oferece como a criação de *joints* entre objetos;
b) explorar mais os recursos da Zigfu para fazer rastreamento do esqueleto do usuário com o Kinect, entre outras coisas;
c) criar ou integrar uma biblioteca de inteligência artificial em Javascript para disponibilizar ao usuário funções como calcular caminhos em uma cena, programar personagens não jogáveis de forma mais fácil entre outras coisas;
d) preparar o motor para desenhar utilizando WebGL para trabalhar com objetos 3D;
e) criar integrações com outros sensores de movimento como o *Leap Motion*;
f) permitir que um objeto do jogo seja formado por um conjunto de objetos filhos e refletir as transformações do pai nos objetos filhos (grafo de cena);
g) criar novos componentes que forneçam comportamentos padrões como: barra de vida, movimentação, placar de pontos, entre outros;
h) preparar o motor para abstrair a criação de jogos *multiplayer* utilizando Web Socket;
i) criar controle de acesso por usuário no editor web;
j) criar controle de acesso aos componentes e objetos do jogo criados no editor e permitir que o usuário publique um componente para que outros usuários possam usá-lo em seus projetos;
k) permitir que vários usuários trabalhem em um mesmo projeto de maneira colaborativa;
l) preparar o editor para exportar o jogo para rodar em dispositivos móveis;
m) experimentos para testes da usabilidade do editor de jogos;
n) melhorar a usabilidade do editor para que usuários sem conhecimento de Javascript consigam usar o editor de maneira mais fácil.

REFERÊNCIAS BIBLIOGRÁFICAS

AEM. **Adopting a component model for game development**. [S.l.], set. 2010. Disponível em: <http://www.anotherearlymorning.com/2010/09/adopting-a-component-model-for-gamedevelopment/>. Acesso em: 14 abr. 2013.

BOX2D. **A 2D physics engine for games.** [S.l.], [2013?]. Disponível em: <http://box2d.org/about/>. Acesso em: 22 out. 2013.

BOX2DJS. **Box2DJS.** [S.l.], 2008. Disponível em: <http://box2d-js.sourceforge.net/>. Acesso em: 22 out. 2013.

DEMOZCENE. **Unity 3D as a demo platform on Mac OS X.** [S.l.], jun. 2010. Disponível em: <http://www.demozcene.wordpress.com/2010/06/27/unity-3d-as-a-demo-platform-onmac-os-x/>. Acesso em: 13 abr. 2013.

ECMA. **TC39 – ECMAScript**. [S.l.], 2011. Disponível em: <http://www.ecmainternational. org/memento/TC39.htm>. Acesso em: 14 abr. 2013.

GAMEPAD. **Gamepad.** [S.l.], 2013a. Disponível em: < https://dvcs.w3.org/hg/gamepad/raw-file/default/gamepad.html>. Acesso em: 13 abr. 2013.

GEARY, David. **Core HTML5 canvas**. Crawfordsville: Prentice Hall, 2012.

GREGORY, Jason. **Game engine architecture**. Wellesley: A K Peters/CRC Press, 2009.

IMPACT. **The awesomest way to create even more awesome HTML5 games.** [S.l.], [2013?]. Disponível em: <http://www.impactjs.com>. Acesso em: 13 abr. 2013.

JBOSS. **JBoss aplication server.** [S.l.], [2013?]. Disponível em: <http://www.jboss.org/overview/>. Acesso em: 22 out. 2013.

LONGHI, Diego. **China on-line:** conectando você com a cultura chinesa. Caxias do Sul, 2004. Disponível em: <http://www.chinaonline.com.br/artes_gerais/tangram/default.asp>. Acesso em: 14 abr. 2013.

MICROSOFT. **Kinect for Windows sensor components and specifications.** [S.l.], 2012. Disponível em: <http://msdn.microsoft.com/en-us/library/jj131033.aspx>. Acesso em: 23 out. 2013.

OSCAR JÚNIOR, Alcântara. **Protótipo de uma linguagem de programação de computadores orientada por formas geométricas, voltada para o ensino de programação**. 2003. 58 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau,

Blumenau.

PEREIRA, Daniel. **Desenvolvimento de um motor de Jogos 3D, utilizando WEBGL**. 2012. 64 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

PRIMEFACES. **Ultimate JSF component suite.** [S.l.], 2011. Disponível em: <http://primefaces.org/>. Acesso em: 22 out. 2013.

SKILLAREA. **Tantalizing Tangrams**. [S.l.], abr. 2013. Disponível em <http://skillarea.com/tantalizing-tangrams/>. Acesso em: 14 abr. 2013.

STEFANOV, Stoyan. **JavaScript patterns**. Sebastopol: O'Reilly Media, 2010.

SUBLIME. **Sublime text.** [S.l.], [2013?]. Disponível em: <http://www.sublimetext.com/>. Acesso em: 22 out. 2013.

UNITY. **Create the games you love with Unity.** [S.l.], [2013?]. Disponível em: <http://www.unity3d.com/unity/>. Acesso em: 13 abr. 2013.

W3SCHOOLS. **HTML5 introduction.** [S.l.], 2013a. Disponível em: <http://www.w3schools.com/html/html5_intro.asp>. Acesso em: 13 abr. 2013.

______. **JavaScript introduction**. [S.l.], 2013b. Disponível em: <http://www.w3schools.com/js/js_intro.asp>. Acesso em: 13 abr. 2013.

WARD, Jeff. **What is a game engine?** [S.l.], 2008. Disponível em: <http://www.gamecareerguide.com/features/529/what_is_a_game_.php?page=1>. Acesso em: 14 abr. 2013.

ZIGFU. **ZDK for javascript.** [S.l.], 2013. Disponível em: <http://zigfu.com/en/zdk/javascript/>. Acesso em: 22 out. 2013.

APÊNDICE A – Detalhamento dos casos de uso da aplicação

Este apêncide apresenta o detalhamento dos casos de uso da aplicação. Do Quadro 10 ao Quadro 19 são apresentados os casos de uso referentes ao `Usuário do Motor`. Do Quadro 20 ao Quadro 28 são apresentados os casos de uso do `Usuário do Editor`.

Quadro 10 – Caso de uso UC01

UC01 – Criar componente	
Descrição	Permite que o `Usuário do Motor` crie um novo componente dentro do motor de jogos.
Cenário Principal	1. O `Usuário do Motor` cria um objeto que estenda do objeto `Component`. 2. O `Usuário do Motor` implementa os métodos padrões do objeto `Component`.
Pós-Condição	O `Usuário do Motor` possui um componente que pode ser associado aos objetos do jogo.

Quadro 11 – Caso de uso UC02

UC02 – Criar cena	
Descrição	Permite que o `Usuário do Motor` crie uma nova cena dentro do motor de jogos.
Cenário Principal	1. O `Usuário do Motor` cria um novo objeto `Scene`.
Pós-Condição	O `Usuário do Motor` possui uma cena na qual podem ser inseridas camadas e componentes.

Quadro 12 – Caso de uso UC03

UC03 – Criar camada	
Descrição	Permite que o `Usuário do Motor` crie uma nova camada dentro do motor de jogos.
Cenário Principal	1. O `Usuário do Motor` cria um novo objeto `Layer`.
Pós-Condição	O `Usuário do Motor` possui uma camada na qual podem ser adicionados os objetos do jogo.

Quadro 13 – Caso de uso UC04

UC04 – Criar câmera	
Descrição	Permite que o `Usuário do Motor` crie uma nova câmera dentro do motor de jogos.
Cenário Principal	1. O `Usuário do Motor` cria um novo objeto `Camera`.
Pós-Condição	O `Usuário do Motor` possui uma câmera que pode ser adicionada no jogo para mudar a visão da cena.

Quadro 14 – Caso de uso UC05

UC05 – Criar *asset*	
Descrição	Permite que o `Usuário do Motor` crie um novo *asset* dentro do motor de jogos.
Cenário Principal	1. O `Usuário do Motor` cria um novo objeto `Asset`.
Pós-Condição	O `Usuário do Motor` possui um *asset* que pode ser utilizado por algum objeto do jogo.

Quadro 15 – Caso de uso UC06

UC06 – Criar objeto do jogo	
Descrição	Permite que o `Usuário do Motor` crie um novo objeto do jogo dentro do motor de jogos.
Cenário Principal	1. O `Usuário do Motor` cria um objeto que estende de `GameObject`. 2. O `Usuário do Motor` implementa os métodos padrões da classe `GameObject`.
Pós-Condição	O `Usuário do Motor` possui objeto do jogo que pode ser associado a alguma camada, receber componentes e desenhado na cena.

Quadro 16 – Caso de uso UC07

UC07 – Executar jogo	
Descrição	Permite que o `Usuário do Motor` execute o jogo.
Cenário Principal	1. O `Usuário do Motor` chama o método `Game.init()`. 2. O motor começa o *loop* principal do jogo, chamando o UC08, UC09 e UC10.
Pós-Condição	O sistema gera todo o HTML necessário e executa o jogo no navegador do `Usuário do Editor`.

Quadro 17 – Caso de uso UC08

UC08 – Capturar entradas do usuário	
Descrição	Permite que o motor capture e propague as entradas do usuário.
Cenário Principal	1. O motor deve se registrar nos eventos disparados pelos dispositivos: mouse, teclado, Kinect e *joysticks*. 2. O motor deve propagar estes eventos para todos os componentes que se registraram nestes eventos.
Pós-Condição	O `Usuário do Motor` irá receber os eventos disparados pelo `Motor`.

Quadro 18 – Caso de uso UC09

UC09 – Tratar colisões	
Descrição	Permite que o motor identifique colisões entre objetos da cena.
Cenário Principal	1. O motor deve se registrar no evento de colisão propagado pela Box2DJS. 2. O motor deve propagar esta colisão para todos os componentes que se registraram neste tipo de evento.
Pós-Condição	O `Usuário do Motor` irá receber o evento de colisão e o objeto que está colidindo.

Quadro 19 – Caso de uso UC10

UC10 – Desenhar cena	
Descrição	Permite que o motor desenhe a cena atual.
Cenário Principal	1. O motor deve percorrer todos os componentes do tipo `RenderSystem`. 2. O motor chama o método `Component.onRender()` do componente atual.
Pós-Condição	O `Usuário do Motor` irá receber o evento de desenhar e cabe a ele a implementação.

Quadro 20 – Caso de uso UC11

UC11 – Criar novo projeto	
Descrição	Permite que o `Usuário do Editor` crie um novo projeto.
Cenário Principal	1. O `Usuário do Editor` clica na opção do menu `New Project.`
Pós-Condição	O sistema cria a estrutura básica de um novo projeto que pode ser visualizada na árvore do projeto.

Quadro 21 – Caso de uso UC12

UC12 – Carregar projeto salvo	
Descrição	Permite que o `Usuário do Editor` carregue um projeto salvo.
Cenário Principal	1. O `Usuário do Editor` clica na opção do menu `Open Project.` 2. O `Usuário do Editor` escolhe o arquivo que deve ser em formato *JavaScript Object Notation* (JSON).
Pós-Condição	O sistema carrega a estrutura salva no arquivo JSON que pode ser visualizada na árvore do projeto.

Quadro 22 – Caso de uso UC13

UC13 – Salvar projeto	
Descrição	Permite que o `Usuário do Editor` salve um projeto.
Cenário Principal	1. O `Usuário do Editor` clica na opção do menu `Save Project.`
Pós-Condição	O sistema gera um arquivo chamado `game.json` que contém toda a estrutura da árvore de projeto.

Quadro 23 – Caso de uso UC14

UC14 – Exportar projeto	
Descrição	Permite que o `Usuário do Editor` exporte um projeto para executar o jogo fora da ferramenta de edição.
Cenário Principal	1. O `Usuário do Editor` clica na opção do menu `Export Project.`
Pós-Condição	O sistema gera um arquivo chamado `game.zip` que contém todos os arquivos necessários para executar o jogo fora da ferramenta em modo *offline*.

Quadro 24 – Caso de uso UC15

UC15 – Manter componentes	
Descrição	Permite que o `Usuário do Editor` insira, edite ou remova um componente.
Cenário Principal	1. O `Usuário do Editor` clica na opção do menu `Components`. 2. O `Usuário do Editor` clica no botão `New`. 3. O `Usuário do Editor` informa o nome, a função de inicialização e o código fonte do componente. 4. O `Usuário do Editor` clica no botão `Save`.
Fluxo Alternativo 1	5. Após o passo 1 o `Usuário do Editor` clica no botão `Editar` do componente. 6. O `Usuário do Editor` edita as informações do componente. 7. O `Usuário do Editor` clica no botão `Save`. 8. O sistema salva as informações no arquivo `.js` do componente no servidor.
Fluxo Alternativo 2	9. Após o passo 1 o `Usuário do Editor` clica no botão de remover componente. 10. O sistema exclui o arquivo do componente do servidor.
Pós-Condição	O sistema cria um arquivo com extensão `.js` em um diretório do servidor contendo o código fonte informado.

Quadro 25 – Caso de uso UC16

UC16 – Manter objetos do jogo	
Descrição	Permite que o `Usuário do Editor` insira, edite ou remova um objeto do jogo.
Cenário Principal	1. O `Usuário do Editor` clica na opção do menu `Game Objects`. 2. O `Usuário do Editor` clica no botão `New`. 3. O `Usuário do Editor` informa o nome, a função de inicialização e o código fonte do objeto do jogo. 4. O `Usuário do Editor` clica no botão `Save`.
Fluxo Alternativo 1	5. Após o passo 1 o `Usuário do Editor` clica no botão `Editar` do objeto do jogo. 6. O `Usuário do Editor` edita as informações do objeto do jogo. 7. O `Usuário do Editor` clica no botão `Save`. 8. O sistema salva as informações no arquivo `.js` do objeto do jogo no servidor.
Fluxo Alternativo 2	9. Após o passo 1 o `Usuário do Editor` clica no botão de remover objeto do jogo. 10. O sistema exclui o arquivo do objeto do jogo do servidor.
Pós-Condição	O sistema cria um arquivo com extensão `.js` em um diretório do servidor contendo o código fonte informado.

Quadro 26 – Caso de uso UC17

UC17 – Manter *assets*	
Descrição	Permite que o `Usuário do Editor` insira ou remova um *asset*.
Cenário Principal	1. O `Usuário do Editor` clica na opção do menu `Assets`. 2. O `Usuário do Editor` clica no botão `Choose`. 3. O `Usuário do Editor` escolhe um arquivo de imagem ou áudio.
Fluxo Alternativo 1	4. Após o passo 1 o `Usuário do Editor` clica no botão de remover de um *asset*. 5. O sistema exclui o arquivo do diretório do servidor.
Pós-Condição	O sistema salva o arquivo escolhido em um diretório do servidor.

Quadro 27 – Caso de uso UC18

UC18 – Manter estrutura do projeto	
Descrição	Permite que o `Usuário do Editor` adicione instâncias na estrutura de árvore do projeto.
Cenário Principal	1. O `Usuário do Editor` clica com o botão direito e abre o menu contextual. 2. O `Usuário do Editor` clica para adicionar uma nova instância (`Layer`, `Component` ou `GameObject`). 3. O `Usuário do Editor` preenche a informações necessárias. 4. O `Usuário do Editor` clica no botão `Save`.
Fluxo Alternativo 1	5. Após o passo 1, o `Usuário do Editor` clica no botão de editar a instância. 6. O `Usuário do Editor` altera as informações. 7. O sistema altera as informações do nó da árvore.
Fluxo Alternativo 2	8. Após o passo 1, o `Usuário do Editor` clica no botão de remover a instância. 9. O sistema remove o nó da árvore.
Fluxo Alternativo 3	10. Após o passo 1, o `Usuário do Editor` clica no botão de mover para cima. 11. O sistema move o nó da árvore para uma posição acima.
Fluxo Alternativo 4	11. Após o passo 1, o `Usuário do Editor` clica no botão de mover para baixo. 12. O sistema move o nó da árvore para uma posição abaixo.
Pós-Condição	O sistema adiciona um novo nó na árvore, representando uma nova instância de `Layer`, `Component` ou `GameObject` no jogo.

Quadro 28 – Caso de uso UC19

UC19 – Executar jogo	
Descrição	Permite que o `Usuário do Editor` execute o jogo.
Cenário Principal	1. O `Usuário do Editor` clica no botão de executar jogo.
Pós-Condição	O sistema gera todo o HTML necessário e executa o jogo no navegador do `Usuário do Editor`.