CENTRO PAULA SOUZA

FACULDADE DE TECNOLOGIA DE MOGI-MIRIM

Diego Victor Gonzaga

Kauê Rocha

Tomas Coradi Lino

Vinícius de Moraes da Silva Pinto

**TO INFINTY AND BEYOND**

JOGO PARA A PLATAFORMA *WEB*, DESENVOLVIDO COM *HTML*5 *CANVAS* E *JAVASCRIPT*

MOGI MIRIM

2018

Diego Victor Gonzaga

Kauê Rocha

Tomas Coradi Lino

Vinícius de Moraes da Silva Pinto

**TO INFINTY AND BEYOND**

JOGO PARA A PLATAFORMA *WEB*, DESENVOLVIDO COM *HTML*5 *CANVAS* E *JAVASCRIPT*

Trabalho de Graduação apresentado ao Curso de Tecnologia em Análise e Desenvolvimento de Sistemas da Faculdade de Tecnologia de Mogi Mirim como pré-requisito para a obtenção do Título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

**Orientador: Thales de Tarsis Cezare**

MOGI MIRIM

2018

Eu **Thales de Tarsis Cezare** orientador deste trabalho de graduação e dos seguintes alunos **Diego Victor Gonzaga**, **Kauê Rocha**, **Tomas Coradi Lino** e **Vinícius de Moraes da Silva Pinto**, na data de 05 de junho de 2018, autorizo-os a apresentação deste trabalho.



RESUMO

Os jogos digitais estão cada vez melhores, mas para que ocorram melhorias, as tecnologias disponíveis precisam evoluir ou novas devem surgir. Recentemente a tecnologia *HTML*5 *Canvas* surgiu para trazer algumas facilidades e novas funcionalidades aos navegadores mais modernos o que possibilitou a criação de novos jogos com gráficos muito melhores, mais leves e menos complexos, tudo isso sem a necessidade de instalar ou utilizar programas terceiros.

Ao decorrer deste trabalho serão apresentados discussões e resultados de todas as fases do processo de desenvolvimento de um jogo, intitulado de *To Infinity and Beyond*, desde a fase de entrevista e planejamento até a fase de programação e implementação, dando ênfase em algumas etapas, como a de desenvolvimento da interface e do motor, ao final estarão anexados alguns documentos técnicos produzidos durante essas etapas, inclusive o código fonte do jogo.

E como decorrência de todo esse trabalho, surgiu o resultado final, foi produzido um jogo, acima citado, utilizando as novas tecnologias disponíveis.

ABSTRACT

Digital games are getting better, but for improvements to take place, the available technologies need to evolve or new ones must emerge. Recently *HTML*5 *Canvas* technology came to bring some features and new features to the most modern browsers, which made it possible to create new games with much better graphics, lighter and less complex, without the need to install or use third-party programs.

Throughout this paper, discussions and results will be presented for all phases of the game development process, titled To Infinity and Beyond, from the interview and planning phase to the programming and implementation phase, with emphasis on some stages, such as the development of interface and engine, at the end will be attached some technical documents produced during these steps, including the source code of the game.

And as a result of all this work, the final result appeared, a game was produced, mentioned above, using the new technologies available.

INDICE DE FIGURAS

[Figura 2.1 – Console Mattel Intellivision 9](#_Toc515819794)

[Figura 2.2 – Jogos mais baixados 2014 11](#_Toc515819795)

[Figura 2.3 – Wolfenstein 3D (1992) 12](#_Toc515819796)

[Figura 2.4 – Camadas da Engenharia de Sotware 14](#_Toc515819797)

[Figura 2.5 – Fases do método em cascata 15](#_Toc515819798)

[Figura 2.6 – O processo da Extreme Programming (XP) 18](#_Toc515819799)

[Figura 2.7 – Exemplo de caso de uso 24](#_Toc515819800)

[Figura 2.8 – Exemplo de Diagrama de Classe 25](#_Toc515819801)

[Figura 2.9 – Exemplo de Diagrama de Atividades 26](#_Toc515819802)

[Figura 2.10 – Função de um programa 29](#_Toc515819803)

[Figura 2.11 – Exemplo de documento *HTML* 31](#_Toc515819804)

[Figura 2.12 – Exemplo de código *Canvas* e sua renderização 32](#_Toc515819805)

[Figura 2.13 – Exemplo de código *JavaScript* 35](#_Toc515819806)

[Figura 2.14 – Jogo criado com Cocos2d-JS 36](#_Toc515819807)

[Figura 2.15 – Jogo desenvolvido com Stage.js 37](#_Toc515819808)

[Figura 4.1 – Protótipo de baixa fidelidade da tela de continuação 43](#_Toc515819809)

[Figura 4.2 – Protótipo de baixa fidelidade da tela de fim de jogo 44](#_Toc515819810)

[Figura 4.3 – Protótipo de baixa fidelidade das telas de pausa, inicial e carregamento 45](#_Toc515819811)

[Figura 4.4 – Protótipos de alta fidelidade das telas de pausa, continuação, fim de jogo e carregamento 45](#_Toc515819812)

[Figura 4.5 – Protótipo de alta fidelidade da tela inicial 46](#_Toc515819813)

[Figura 4.6 – Interface de usuário destacada em vermelho 56](#_Toc515819814)

[Figura 4.7 – Spritesheet do jogo 70](#_Toc515819815)

SUMARIO

[RESUMO 4](#_Toc518681264)

[ABSTRACT 5](#_Toc518681265)

[INDICE DE FIGURAS 6](#_Toc518681266)

[SUMARIO 7](#_Toc518681267)

[1. INTRODUÇÃO 7](#_Toc518681268)

[2. REVISÃO BIBLIOGRAFICA 8](#_Toc518681269)

[2.1. JOGOS E SUAS PLATAFORMAS 8](#_Toc518681272)

[2.1.1. CONSOLE 9](#_Toc518681273)

[2.1.2. DISPOSITIVOS MOVEIS 10](#_Toc518681274)

[2.1.3. COMPUTADORES 11](#_Toc518681275)

[2.1.4. *WEB* 12](#_Toc518681276)

[2.2. ENGENHARIA DE SOFWARE 13](#_Toc518681277)

[2.2.1. METODOLOGIAS TRADICIONAIS 14](#_Toc518681278)

[2.2.2. METODOLOGIAS ÁGEIS 17](#_Toc518681279)

[2.2.3. LEVANTAMENTO DE REQUISITOS 20](#_Toc518681280)

[2.2.4. UML 22](#_Toc518681281)

[2.3. PROJETO DE INTERFACE 26](#_Toc518681282)

[2.4. DESENVOLVIMENTO DE *SOFTWARE* 28](#_Toc518681283)

[2.4.1. PROGRAMAÇÃO ORIENTADA A OBJETOS (POO) 29](#_Toc518681284)

[2.4.2. PROGRAMAÇÃO BASEADA EM PROTÓTIPOS 29](#_Toc518681285)

[2.4.3. INTERATIVIDADE NA *WEB* 30](#_Toc518681286)

[2.4.4. TECNOLOGIAS 30](#_Toc518681287)

[2.4.5. ANATOMIA DE UM VIDEOGAME 32](#_Toc518681288)

[2.4.6. TÉCNICAS 33](#_Toc518681289)

[2.4.7. FERRAMENTAS DE DESENVOLVIMENTO 34](#_Toc518681290)

[3. METODOLOGIA 38](#_Toc518681291)

[3.1. INSTRUMENTO DE COLETA DE DADOS 38](#_Toc518681293)

[3.2. CRONOGRAMA 38](#_Toc518681294)

[3.3. ATIVIDADES 39](#_Toc518681295)

[4. DESENVOLVIMENTO 40](#_Toc518681296)

[4.1. ENTREVISTA 40](#_Toc518681298)

[4.2. LEVANTAMENTO DE REQUISITOS 42](#_Toc518681299)

[4.3. DESENVOLVIMENTO DA INTERFACE 42](#_Toc518681300)

[4.4. DESENVOLVIMENTO DO MOTOR DA APLICAÇÃO 46](#_Toc518681301)

[4.4.1. LAÇO PRINCIPAL 47](#_Toc518681302)

[4.4.2. RECURSOS Externos 49](#_Toc518681303)

[4.4.3. CONTROLES 50](#_Toc518681304)

[4.4.4. JOGO 52](#_Toc518681305)

[4.4.5. JOGADOR E NAVE 54](#_Toc518681306)

[4.4.6. INIMIGOS 59](#_Toc518681307)

[4.4.7. COLISÕES E EXPLOSÕES 64](#_Toc518681308)

[4.4.8. ITENS 67](#_Toc518681309)

[4.5. IMPLEMENTAÇÃO DAS TELAS 70](#_Toc518681310)

[5. RESULTADOS E DISCUSSÕES 73](#_Toc518681311)

[6. BIBLIOGRAFIA 75](#_Toc518681312)

[7. APÊNDICE A 79](#_Toc518681313)

# INTRODUÇÃO

Segundo notícia do G1 (G1 RIO, 2015), o Brasil é o quarto maior consumidor de jogos digitais do mundo, os empregos no setor cresceram 13 vezes mais que o mercado de trabalho em geral, mas vem sofrendo com a falta de mão de obra qualificada. A consequência disso é o crescimento das ferramentas e tecnologias de apoio para esta atividade, que estão se tornando mais poderosas e eficientes, e cada vez mais simples, ao ponto de não ser necessário ter conhecimento de nenhuma linguagem de programação, tornando tudo muito intuitivo. Alguns exemplos abordados ainda neste trabalho são, o *Cocos2d-JS*, o *Stage.js* e o *Isogenic Engine*.

Com o crescimento constante da tecnologia, hoje é possível através de um celular, ter acesso à Internet em qualquer lugar e a qualquer hora. Tal facilidade implica diretamente no mercado de jogos digitais, pois a alta taxa de pessoas com acesso à rede mundial de computadores, que segundo o IBGE (G1 RIO, 2016) é de mais de 50% dos domicílios brasileiros desde 2014, gerou um maior número de interessados neste segmento, tornando-o cada vez mais forte e atraindo mais empresas do ramo a investir no Brasil, o que possibilita um crescimento muito grande para aqueles que trabalham neste mercado.

As aplicações construídas para a plataforma *web* estão em maior evidência, prova disso é o fato de a rede pública de computadores tem milhares de acessos diariamente, exemplo disso é a informação compartilhada pelo Olhar Digital (REDAÇÃO OLHAR DIGITAL, 2016), empresa jornalística do ramo de informática, que destaca o site *Giphy* por ter alcançado o incrível número 100 milhões de usuários diários.

Há constantemente novas tecnologias surgindo para auxiliar no desenvolvimento dos jogos digitais, tecnologias essas que são emergentes, como por exemplo, o *HTML5* e o *Canvas* que são bem recentes, mas já são muito empregadas nos jogos, como, o *Sand Trap*[[1]](#footnote-1) e *CUBE[[2]](#footnote-2)*.

Uma vez que o acesso à Internet está cada vez mais fácil, a demanda de jogos para esta plataforma aumenta consequentemente, dados que comprovam isso são os apresentados pelo Estadão (RONCOLATO, 2011), que previa que o mercado de games sócias, jogos de *Facebook*, por exemplo, até 2015 ia quintuplicar chegando a impressionante marca de cinco bilhões de dólares. E com esse grande crescimento uma nova necessidade surge, que é a agilização do processo de construção de aplicações para esse mercado, para tentar suprir essa grande demanda procura-se ferramental especializado, que poderiam ser usadas para potencializar, facilitar e acelerar o desenvolvimento de aplicações neste segmento. Por ser um mercado novo, será que as ferramentas de modelagem de *software* convencionais estão preparadas? Este tipo de aplicação não exige um tipo de ferramenta mais especializada?

Levando em conta esses pontos, neste mundo completamente novo, tanto para os desenvolvedores quanto para as aplicações de apoio ao desenvolvimento, a viabilidade da aplicação se torna possível?

Este trabalho de graduação tem como objetivo, mostrar que mesmo com pouco ferramental especializado para o desenvolvimento neste ambiente, já é possível criar um jogo com recursos de qualidade, com boa usabilidade e capaz de concorrer com outros jogos da mesma área. Propõe-se um jogo simples o suficiente para funcionar até mesmo em um computador com baixo poder computacional, que pode atingir pessoas de todas as faixas de etárias. A ideia principal do jogo é propor partidas rápidas, e que os jogadores em um intervalo de tempo curto consigam se divertir sem compromisso.

# REVISÃO BIBLIOGRAFICA



## JOGOS E SUAS PLATAFORMAS

Segundo Arruda (2014, p. 3):

A concepção de jogos digitais envolve um entendimento mais amplo do que apenas saber o que são videogames. O termo videogame historicamente esteve limitado aos jogos de console e às máquinas de fliperama. Já as tecnologias digitais são baseadas na microinformática, o que engloba jogos para computadores, consoles, fliperamas, smart*Phone*s, *tablets* e qualquer equipamento que venha a existir.

Com o passar dos anos, cada vez mais os jogos vêm ganhando espaço, em 2013 no Brasil, conforme apresentado por Arruda (2014, p. 4), uma pesquisa da *Newzoo* revelou que mais de 35 milhões de brasileiros jogam jogos digitais, a pesquisa não aborda outras plataformas, como celulares e tablets, mas apresenta uma pequena ideia da grandeza que este mercado possui.

As diversas plataformas apresentam suas peculiaridades, porém de acordo com o ponto de vista do jogador vantagens podem virar desvantagens e vice-versa, afinal a escolha da plataforma tem muita influência dos fatores que o jogador mais leva em consideração, como, conforto, qualidade dos gráficos, desempenho, portabilidade, custos, etc. A seguir uma breve contextualização de cada uma delas.

### CONSOLE

De acordo com o artigo publicado por Clua e Bittencourt (2005, p. 1319)os consoles ou videogames são jogos eletrônicos desenvolvidos para uso doméstico, que basicamente funcionam acoplados a aparelhos de televisão.

Em outro artigo, Pinto, Coronel e Bresolin (2013, p. 46), complementam que um console tem por objetivo “executar *software*s de jogos de vídeo e é caracterizado como um computador otimizado nos seus componentes internos e externos”, ou seja, foram projetados com *hardware* dedicado para executar jogos, de diversas naturezas.

Os primeiros consoles executavam jogos muito simples, monocromáticos, com simples formas geométricas e sem som, mas isso mudou no decorrer dos anos, pois avanços de *hardware* permitiram aos consoles evoluírem a ponto de hoje executarem jogos 3D coloridos, multijogadores, com efeitos sonoros, músicas, acesso à Internet, e muitas outras novas funcionalidades.

Figura 2.1 – Console *Mattel Intellivision*



Fonte: GONZALES CLUA e BITTENCOURT (2005, p. 1320)

O maior benefício apresentado pelos consoles é sua alta performance, uma vez que todo seu *hardware* é dedicado para executar jogos seu desempenho realizando esta atividade é surpreendentemente superior a demais plataforma, entretanto, como todo seu poder de processamento está direcionado para esta tarefa os consoles acabam por ter um conjunto limitado de funcionalidades se comparado a um computador comum. E em geral são relativamente caros se levando em conta esse limitado número de tarefas que podem realizar.

### DISPOSITIVOS MOVEIS

Segundo o artigo de Lemes, Tomaselli e Camarotti (2012, p. 2), publicado na *SBGames*, os dispositivos moveis são telefones celulares, tablets, e aparelhos pequenos e leves com capacidade de reproduzir jogos.

Em uma matéria da revista Veja (REDAÇÃO VEJA, 2016), a empresa de tecnologia Cisco projeta que o número de celulares no mudo chegará a 5,4 bilhões em 2020, e essa grande massa está a procura sempre de algo a mais, Lecheta (2013, p. 21) afirma, “usuários comuns estão procurando cada vez mais celulares com diversos recursos, como câmeras, músicas, *Bluetooth*, ótima interface visual, jogos, GPS, acesso à internet e e-mail”.

Os números apresentados pela Cisco (REDAÇÃO VEJA, 2016) deixam claro o poder de popularização desta plataforma, devido a esses números absurdos, e como citado no artigo da *SBGames* (LEMES, TOMASELLI e , 2012), os dispositivos moveis são uma plataforma portátil de grande alcance, mesmo em países de baixa renda, mostrando tanto potencial era de se esperar que os jogos também embarcassem nesse mercado.

A portabilidade dos dispositivos moveis é a sua maior vantagem e ao mesmo tempo seu calcanhar de Aquiles, afinal suas dimensões reduzidas implicam que os componentes que o constituem sejam menores, que por sua vez possuem recursos reduzidos, consequentemente a experiência de usuário pode ser prejudicada em diversos fatores, como, qualidade dos gráficos, áudios, desempenho, etc.

Outro ponto a se considerar é a jogabilidade, que devido ao pequeno conjunto de botões, controles e o tamanho da tela pode diminuir ou dificultar as ações durante o jogo, outro velho inimigo dos dispositivos moveis sempre foi a bateria, pois jogos em geral consomem mais energia, afinal utilizam muitos recursos do aparelho para identificar as interações dos jogadores.

Mas independente de todos esses fatores, o número de *downloads* de jogos para dispositivos moveis vêm crescendo cada vez mais, isso fica claro na matéria apresentada pelo G1 (GOMES, 2014), que lista os cinco jogos mais baixados no *Google Play*, loja de aplicativos do *Google*. Em primeiro e segundo lugar estão Pou e *Subway Surfers* respectivamente, mesmo sem números oficiais publicados é possível imaginar a popularidade atingida por esses aplicativos tendo em vista o grande sucesso do jogo Meu Tom Falante, terceiro colocado, que a *Outfit7*, desenvolvedora do jogo, afirma ter cerca de 230 milhões de *downloads*, seguido por Meu Malvado Favorito em quarto, e Perguntados em quinto.

Figura 2.2 – Jogos mais baixados 2014



Fonte: (GOMES, 2014)

### COMPUTADORES

No passado, assim como outras plataformas, os computadores possuíam limitações de *hardware* que reduziam sua capacidade de processamento e consequentemente prejudicava a qualidade dos jogos. Com os avanços tecnológicos durante vários anos a eficiência dos componentes que constituíam os computadores se desenvolveu muito, o que implicou no surgimento de novas tecnologias relacionadas ao universo de desenvolvimento de jogos, como o 3D.

Clua e Bittencourt (2005, p. 1326) em seu artigo destacam “Com as novas tecnologias de criação de universos tridimensionais e a redução dos preços de processadores mais potentes, os jogos para computador tornaram-se altamente imersivos” e hoje, existem até computadores com *hardware* dedicado para execução de jogos, construídos com componentes de alto desempenho e configurações específicas para tal atividade.

Figura 2.3 – *Wolfenstein* 3D (1992)



Fonte: (GONZALES CLUA e BITTENCOURT, 2005, p. 1326)

Os computadores têm uma grande vantagem em relação às demais plataformas, um computador sendo *gamer* ou não, absolutamente não será utilizado apenas para jogar, mas também para outras inúmeras finalidades, ou seja, além da possibilidade de jogar os usuários podem também emprega-lo em outras atividades, como, para trabalhar, estudar, ouvir músicas, assistir a filmes, etc.

### *WEB*

Antes de qualquer coisa é preciso lembrar que a Internet nem sempre foi o que ela é hoje. A sua primeira versão, a *Web* 1.0, segundo Silveira Ferreira (2010, p. 13, 19), os sites eram estáticos e possuíam baixa ou nenhuma interatividade, isto é, apresentavam muitas informações aos visitantes e estes não podiam modifica-las ou contribuir de alguma forma. Na *Web* 2.0 “ao contrário da *web* 1.0, uma informação pode ser atualizada por inúmeras pessoas, permitindo que os usuários sejam muito mais do que meros espectadores e façam parte do espetáculo” (SILVEIRA FERREIRA, 2010, p. 21).

A nova possibilidade de interação do usuário trouxe muitas possibilidades e inovações, alguns exemplos apresentados por Silveira Fereira (2010, p. 21) são, *Flickr*, *Facebook*, *Twitter* e *MySpace*, e, claro, os jogos podem entrar nessa lista.

A *Web* moderna se tornou r*API*damente uma plataforma viável não só para a criação de jogos de alta qualidade, mas, mais importante que isso, tornou-se um ótimo meio de distribuição desses jogos.

De acordo com uma página sobre desenvolvimento de jogos *Web* da *Mozilla* *Developers Network*, MDN (2017):

A gama de jogos que podem ser criados está a par com desktops e sistemas operacionais nativos. Com as tecnologias da *Web* moderna e um navegador recente, é perfeitamente possível fazer jogos impressionantes, de alto nível. Nada de jogos de cartas simples ou jogos sociais multijogadores de antigamente feitos usando *Flash*, mas sim jogos de ação em 3D, RPGs, e muito mais. Graças a grandes melhorias de desempenho na tecnologia do compilador *JavaScript*, e novas *APIs*, é possível construir jogos que rodam no navegador, ou em dispositivos com suporte a *HTML*5 como aqueles baseados no Firefox OS, sem fazer concessões.

As vantagens de desenvolver aplicações *web* são muitas, por exemplo, o alcance da Internet é enorme, está em toda parte, jogos construídos com *HTML*5 funcionam em smart*Phone*s, tablets, PCs e *Smart* TVs, o marketing e a popularização são maiores, a aplicação não fica limitada a ser publicada em somente uma loja de aplicativos, ao contrário, ela pode ser anunciada e publicada em toda a rede como também em outras formas de mídia, aproveitando a facilidade de compartilhamento que a Internet proporciona, e assim alcançando novos horizontes.

Uma notícia do G1 (2011) pode exemplificar o sucesso dos jogos desta plataforma, em 2011, o jogo *CityVille* tinha uma média de 75 milhões de usuários ativos por mês, enquanto o *The Sims* Social possuía uma média de 40 milhões, Empires & Allies e Zynga Poker da famosa empresa de jogos Zynga, contavam com 39 e 36 milhões respectivamente, jogos estes disponíveis no *Facebook*. Apenas estes quatro jogos somavam quase 200 milhões de usuários e isso no ano de 2011 e em 2013 o *Facebook* (G1, 2013) registrou mais de 250 milhões usuários mensais jogando na rede, e com certeza o número continuará a crescer.

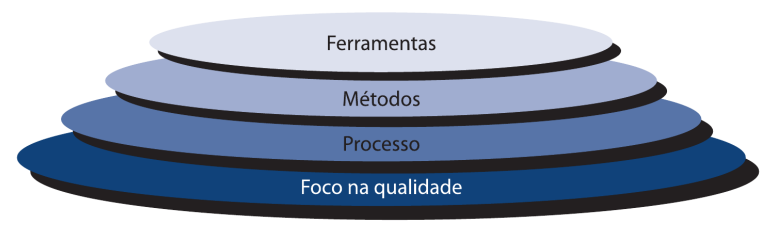
## ENGENHARIA DE SOFWARE

Pressman (2009, p. 39) e Hirama (2011, p. 7) lembram que Fritz Bauer propôs a definição, “Engenharia de *Software* é o estabelecimento e o emprego de sólidos princípios de engenharia de modo a obter *software* de maneira econômica, que seja confiável e funcione de forma eficiente em maquinas reais”, e logo Pressman (2009) apontou que essa definição “diz pouco sobre os aspectos técnicos da qualidade de *software*; ela não trata da necessidade de satisfação do cliente ou da entrega do produto dentro do prazo”, então para melhor definir a engenharia de *software* ele cita que a IEEE, Instituto de Engenheiros Eletricistas e Eletrônicos, desenvolveu uma definição mais abrangente ao afirmar que:

Engenharia de *software*: (1) A aplicação de uma abordagem sistemática, disciplinada e quantificável no desenvolvimento, na operação e na manutenção de *software*, isto é, a aplicação de engenharia ao *software*. (2) O estudo de abordagens como definido em (1)

Porém a abordagem varia muito de equipe para equipe, o importante é disciplina, mas sem perder muita adaptabilidade e agilidade. A Engenharia de *Software* é uma tecnologia dividida em camadas, conforme visto na figura 3.1.

Figura 2.4 – Camadas da Engenharia de *Software*



Fonte: (PRESSMAN, 2009)

A base da engenharia de *software* é a camada de processos. O processo de engenharia de *software* é a liga que mantém as camadas de tecnologia coesas e possibilita o desenvolvimento de *software* de forma racional e dentro do prazo. (PRESSMAN, 2009, p. 40)

Hirama (2011, p. 9) diz que a:

Camada de métodos provê as abordagens e as atividades necessárias para a construção de um *software*. Os métodos abrangem um conjunto amplo de tarefas que incluem analise de requisitos, projeto, implementação, teste e manutenção.

Sobre a camada de ferramentas Pressman (2009) afirma que as ferramentas de engenharia de *software* devem fornecer suporte automatizado ou semi-automatizado para processos e métodos, de forma que quando integradas uma criei informação e que esta possa ser usada por outra.

Hirama (2011, p. 8) comenta que a “camada foco em qualidade, dá-se ênfase à preocupação de qualquer disciplina de engenharia que é a qualidade”.

Pode-se concluir então que a engenharia de *software* é uma ferramenta, uma técnica, que tem o objetivo de guiar, criar uma estrutura para o desenvolvimento de *software* de maneira a obter maior valor e qualidade possível ao processo e ao resultado final da atividade de desenvolvimento do produto.

### METODOLOGIAS TRADICIONAIS

As metodologias tradicionais são aquelas que foram as primeiras a serem pensadas para organizar os processos do desenvolvimento de *software* e foram muito utilizadas no passado, elas possuíam um custo muito alto de mudança, pois o acesso aos computadores era limitado.

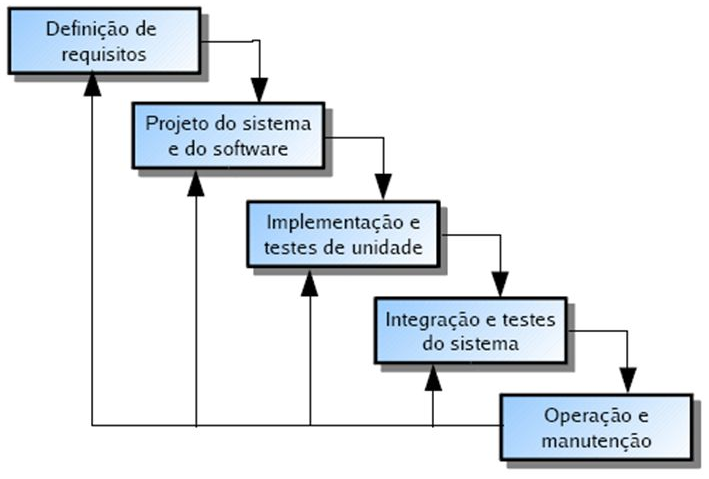
A história tem demonstrado que esses modelos tradicionais proporcionaram uma considerável contribuição quanto à estrutura utilizável no trabalho de engenharia de *software* e forneceram um roteiro razoavelmente eficaz para as equipes de *software*. Entretanto, o trabalho de engenharia de *software* e o seu produto permanecem “à beira do caos”. (PRESSMAN, 2009, p. 58, 59)

Sommerville (2003, p. 36) diz que “Esses modelos genéricos não são descrições definitivas de processos de *software*. Em vez disso, são abstrações úteis, que podem ser utilizadas para explicar diferentes abordagens do desenvolvimento de *software*. ”

#### MÉTODO EM CASCATA

O modelo em cascata, conhecido também como ciclo de vida de *software* foi o primeiro modelo a ser desenvolvido e, basicamente, consistia em implementar engenharia em desenvolvimento de *software*, ou seja, desenvolver o *software* em fases, como mostrado na figura 2.5.

Figura 2.5 – Fases do método em cascata



Fonte: (SOMMERVILLE, 2003)

Segundo Sommerville (2003, p. 38):

Devido aos custos de produção e aprovação de documento, as iterações são onerosas e envolvem um “retrabalho” significativo. Portanto, após um número pequeno de iterações, é normal suspender partes do desenvolvimento, como a especificação, e continuar com os estágios posteriores de desenvolvimento.

Posteriormente afirma:

Hoje em dia, o trabalho de *software* tem um ritmo acelerado e está sujeito a uma cadeia de mudanças intermináveis (em características, funções e conteúdo de informações). O modelo cascata é frequentemente inapropriado para tal trabalho. (SOMMERVILLE, 2003, p. 61)

#### MÉTODO DE DESENVOLVIMENTO EVOLUCIONÁRIO

O método de desenvolvimento evolucionário consiste em primeiramente, entregar um *software* básico ao usuário e, com base nas necessidades do cliente, ir evoluindo o *software*, tendo assim, várias versões.

Segundo Sommerville (2003, p. 39) “A abordagem evolucionária do desenvolvimento de *software*, muitas vezes, é mais eficaz do que a abordagem em cascata, no sentido de produzir sistemas que atendam às necessidades imediatas dos clientes”.

Esse método é muito interessante quando usado em sistemas pequenos ou até em sistemas médios, porém quando se pensa em implementar o método evolucionário em sistemas grandes, temos problemas, pois os sistemas são mal estruturados devido às mudanças e para definir o progresso no desenvolvimento de um *software* com este método é difícil.

Pressman (2009, p. 64) diz que “O segredo é definir as regras do jogo logo no início; isso significa que todos os envolvidos devem concordar que o protótipo é construído para servir como um mecanismo para a definição de requisitos”.

#### MÉTODO INCREMENTAL

No método incremental, após definir as funcionalidades do sistema, o cliente deve dizer quais funcionalidades são mais importantes ou menos importantes, já na parte do desenvolvimento, é desenvolvido e entregue primeiro as funcionalidades mais importantes e as outras são incrementadas depois, fazendo assim com que o cliente tenha acesso às funcionalidades que ele mais necessita em um tempo mais curto e após isso o sistema vai evoluindo e suprindo todas as necessidades.

Sommerville (2003, p. 44) diz que “Uma vez identificados os incrementos, os requisitos para as funções a serem entregues no primeiro incremento são definidos em detalhes, e esse incremento é desenvolvido [...]”.

O desenvolvimento incremental é particularmente útil nos casos em que não há pessoal disponível para uma completa implementação na época de vencimento do prazo estabelecido para o projeto. (PRESSMAN, 2009, p. 62)

### METODOLOGIAS ÁGEIS

Métodos ágeis são caracterizados por serem mais focados no trabalho cooperativo do que na documentação escrita em si, ou seja, são métodos mais práticos, reduzindo assim o tempo do desenvolvimento do *software*.

Segundo Pressman (2009, p. 82):

Na economia moderna é frequentemente difícil ou impossível prever como um sistema computacional (por exemplo, uma aplicação baseada na *Web*) irá evoluir com o tempo. As condições de mercado mudam r*API*damente, as necessidades dos usuários finais se alteram e novas ameaças competitivas emergem sem aviso. Em muitas situações, não se conseguirá definir completamente requisitos antes que se inicie o projeto. É preciso ser ágil o suficiente para dar uma resposta ao ambiente de fluido negócios.

Esses princípios foram reunidos por um grupo de metodologistas conhecidos em um documento chamado de **Manifesto Ágil**. Esse documento proclama os seguintes valores:

* Indivíduos e interações, em relação a processos e ferramentas.
* *Software* funcional, em relação a documentação abrangente
* Colaboração com o cliente, em relação a negociações contratuais.
* Resposta à mudança, em relação a seguir planos.

(FILHO, 2009, p. 102)

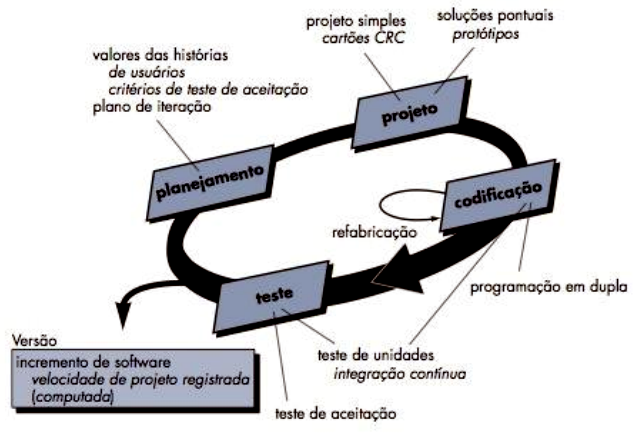
#### EXTREME PROGRAMMING (XP)

O método XP é o mais utilizado dos métodos ágeis e funciona em volta de cinco valores – comunicação, simplicidade, *feedback*, coragem e respeito.

Segundo Pressman (2009, p. 88):

A Extreme *Programming* (programação extrema) emprega uma abordagem orientada a objetos (Apêndice 2) como seu paradigma de desenvolvimento preferido e envolve um conjunto de regras e práticas constantes no contexto de quatro atividades metodológicas: planejamento, projeto, codificação e testes.

Figura 2.6 – O processo da Extreme *Programming* (XP)



Fonte: (PRESSMAN, 2009)

Wilson de Pádua Paula Filho (2009, p. 105) diz que “Provavelmente, o maior problema com muitas tentativas de aplicação do XP é o fato de que a **metodologia não é aplicada de forma completa e consistente**”.

O grupo utilizou uma forma modificada do método XP, pois com isso teríamos entregas constantes de partes operacionais, fazendo com que pudéssemos avaliar e ter um *feedback* de cada parte separadamente conforme o desenvolvimento, além do processo de desenvolvimento ser mais rápido.

#### SCRUM

O método *Scrum* segue também o Manifesto Ágil e tem algumas características estruturais, como: requisitos, análise, projeto, evolução e entrega. O método consiste em dividir o projeto em partes, chamadas *sprints* (o número de *sprints* em cada atividade varia dependendo de tamanho e complexidade) e todo dia ter as *daily scrums*, que são reuniões curtas para que toda a equipe esteja por dentro do que está sendo feito diariamente no projeto, na qual são feitas três perguntas-chave:

* O que você realizou desde a última reunião de equipe?
* Quais obstáculos estão encontrando?
* O que planeja realizar até a próxima reunião da equipe?

Segundo Pressman (2009, p. 96) “Um líder da equipe, chamado *Scrum* *master*, conduz a reunião e avalia as respostas de cada integrante. A reunião *Scrum*, realizada diariamente, ajuda a equipe a revelar problemas potenciais o mais cedo possível”.

Ao final do *sprint* deverá haver a *sprint review* meeting, para avaliar o produto do trabalho, e, eventualmente, a *sprint retrospective*, para avaliar os processos de trabalho. Assim, se aprovado, o produto (parcial ou final) poderá ser entregue ao cliente. (WAZLAWICK, 2013)

#### MÉTODO DE DESENVOLVIMENTO DE SISTEMAS DINÂMICOS (DSDM)

O método DSDM consiste em trabalhar com a filosofia de que 80% de um *software* pode ser entregue em 20% do tempo que levaria para entregar 100% e utiliza prototipagem incremental, ou seja, faz-se vários protótipos, cada um sendo uma versão melhorada do anterior, esses 20% que faltam pode ser incrementado depois, quando os requisitos estiverem melhores definidos ou quando forem solicitadas alterações.

Segundo Pressman (2009, p. 97) O DSDM “[...] é uma abordagem para construir e manter sistemas que atendem restrições de prazo apertado através do uso da prototipagem incremental em um ambiente controlado”.

O DSDM Consortium é um grupo de empresas do mundo todo que mantém esse método e eles definiram um método ágil chamado ciclo de vida DSDM, onde há três ciclos iterativos: Iteração de modelos funcionais – produzir protótipos incrementais que demonstram funcionalidades – Iteração de projeto e desenvolvimento – revisita protótipos para validar a qualidade – Implementação – É quando se coloca a última versão do *software* no ambiente de trabalho (O incremento não pode estar completo e alterações ainda podem ser solicitadas na hora da implementação).

“O DSDM não é recomendável para projetos nos quais a segurança é um fator crítico, pois a necessidade de testes exaustivos desse tipo de sistema entra em conflito com os objetivos de custo e prazo do DSDM. ” (2013, p. 53)

### LEVANTAMENTO DE REQUISITOS

Levantamento de requisitos é um processo no desenvolvimento de *software* e a primeira atividade técnica. Uma das atividades mais importantes, é de vital importância para um bom entendimento sobre o *software* desejado pelo cliente. Ele deve definir os serviços e atividades que o sistema deve ou não realizar, como interface e restrições do sistema.

É metodologia de desenvolvimento e manutenção de sistemas modulares com as seguintes características: processo (roteiro) dinâmico, integrado e inteligente de soluções tecnológicas. (REZENDE, 2005, p. 90)

O mapeamento dos processos ajuda o levantamento de requisitos para melhoria dos resultados, estabelecendo o que se deve fazer.

Conforme proposto por Somerville (2003, p. 83), um requisito é tratado como funcional quando descreve um serviço ou função que o sistema deve realizar. Paralelamente pode haver requisitos não-funcionais, que são restrições impostas, tanto ao sistema quanto ao desenvolvedor. Esse levantamento deve ser feito com o intuito de satisfazer as necessidades do cliente e funcionalidades propostas. Não se deve esquecer do especialista do domínio, profissional que trabalha na área em que o *software* será aplicado. Esse profissional possui experiência sobre os assuntos do ramo de mercado, ele não deve se apegar apenas a opinião da diretoria, mas também a dos usuários, mas é necessário tomar cuidado com a utilização dos profissionais. Sempre lembrando que essas práticas se bem-feitas, ajudam na garantia de qualidade se os processos não forem negligenciados.

#### ENTREVISTAS

A técnica da Entrevista também conhecida como Reunião é uma das mais conhecidas e mais simples de se utilizar no processo de levantamento de requisitos, justamente por ser fácil e simples de ser realizada essa técnica é comumente utilizada ao primeiro contato com o cliente e geralmente produz bons resultados nessa fase onde o principal objetivo é a obtenção de dados para assim entender as necessidades a serem vistas e desse modo trazer as possíveis soluções que se adequem bem ao desejo e satisfação do cliente. Antes de tudo é necessário que o entrevistador já tenha uma pauta para a sua entrevista, desse modo não indo totalmente a parte do assunto e dessa maneira não a entrevista tão longa e nem dispersa, podendo ser eficaz e com objetivo.

As seguintes diretrizes podem ser de grande auxilio na direção de entrevistas bem-sucedidas com o usuário: desenvolver um plano geral de entrevistas, certificar-se da autorização para falar com os usuários, planejar a entrevista para fazer uso eficiente do tempo, utilizar ferramentas automatizadas que sejam adequadas, tentar descobrir que informação o usuário está mais interessado e usar um estilo adequado ao entrevistar. (MORAES, 2009)

O planejamento de uma entrevista pode se dizer que é o pré-requisito da mesma, sendo assim nessa etapa do planejamento devem ser coletados e estudados todos os dados pertinentes a discussão dessa forma deixando os assuntos tratados na hora da execução mais contextualizados, fazendo assim a produtividade ser maior.

Cabe ao analista ao fim da reunião realizar uma validação de todo aquele documento que foi feito, verificando se o conteúdo abordado vai atender as reais necessidades do cliente, se o cliente mudou de ideia em algum aspecto e se o mesmo está ciente e bem entendido sobre as questões abordadas e tratadas na entrevista.

A atitude do analista em relação à entrevista é determinar seu fracasso ou sucesso. Uma entrevista não é uma competição, deve-se evitar o uso excessivo de termos técnicos e não conduzir a entrevista em uma tentativa de persuasão. O modo como o analista fala não deve ser muito alto, nem muito baixo, tampouco indiretamente, ou seja, utilizar os termos: ele disse isso ou aquilo na reunião para o outro entrevistado. (MORAES, 2009)

O fator que decide o fracasso ou sucesso de uma entrevista é o comprometimento e maneira de como o entrevistador irá conduzi-la, é importante que o ao executar tenha a consciência de que o objetivo ali é ajudar o cliente e o atender da melhor maneira possível, soluções devem ser apresentadas e a opinião do entrevistado sempre deve ser escutada pois o perito no assunto é o próprio entrevistado e sua credibilidade jamais deve ser questionada, desse modo um bom entrevistador é aquele que saber usar as duas etapas da entrevista a de ouvir as necessidades e ideias e desse modo poder transmitir ao cliente algumas ideias e opiniões que agreguem e melhorem o todo.

Pode ser concluído que a entrevista mesmo com o passar do tempo nunca perde seu valor e poder no quesito e sempre é bem-vinda, principalmente por estar sem dirigida diretamente com o cliente que é o responsável por todo o processo existir e desse modo poder satisfazê-lo da melhor maneira.

#### QUESTIONÁRIO

Também muito utilizada e conhecida, o Questionário é uma técnica de levantamento de requisitos mais usados quando a informação a ser extraída é de um grande número de pessoas e os assuntos abordados são os mesmos, desta forma simplificando a tarefa e fazendo-a ser mais eficiente.

Deve ser desenvolvido um controle que identifique todas as pessoas que receberão os questionários. A distribuição deve ocorrer junto com instruções detalhadas sobre como preenchê-lo e ser indicado claramente o prazo para devolução do questionário. (MORAES, 2009)

Nessa técnica diferente da entrevista, temos as questões sendo dirigidas por escrito ao grupo de participantes e o objetivo final é poder ser realizada uma análise sobre as diferentes opiniões e pontos de vista sobre o mesmo assunto, podendo ao juntar todas essas opiniões em uma forma só e realizando um produto que agrade de maneira geral.

Na fase de preparação do questionário deve ser indicado o tipo de informação que se deseja obter. Assim que os requisitos forem definidos o analista deve elaborar o questionário com questões de forma simples, clara e concisa, deixar espaço suficiente para as repostas que forem descritivas e agrupar as questões de tópicos específicos em um conjunto com um título especial (MORAES, 2009)

Como visto o questionário é uma boa técnica justamente por atingir um grande número de pessoas e dessa forma trazendo uma visão mais ampla, porém o analista deve ter cuidado pois ao mesmo tempo essa quantidade o de opiniões podem se dispersar justamente por tratarem muitas vezes de pontos de vista diferente e interpretações diferentes para cada entrevistado, cabe ao analista juntar todos os dados coletos ao final e assim realizar o melhor padrão conforme as ideias vistas.

### UML

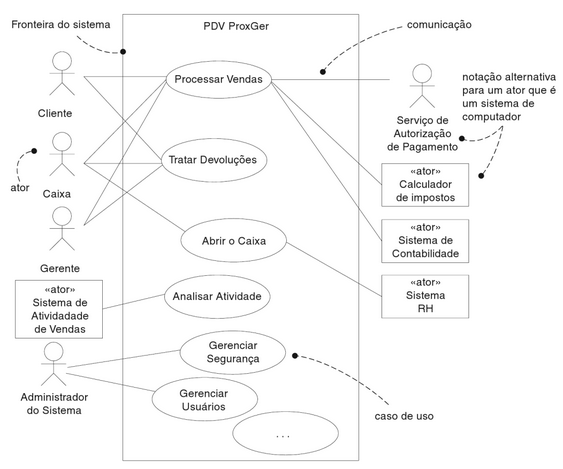
Larman (2007, p. 39) define UML, *Unified Modeling Language* traduzido como Linguagem de Modelagem Unificada, como uma notação diagramática padrão, usada para desenhar ou apresentar figuras relacionadas a *software*, Booch, Rumbaugh e Jacobson (2005, p. 14) definem como uma linguagem para elaboração da estrutura de projetos de *software* e diz que sempre será necessário utilizar vários modelos, referindo se aos diagramas, pois estes conectados tornam possível a compreensão do sistema. Unindo estas duas visões é possível concluir que UML é uma linguagem usada para representar graficamente os comportamentos, sequência de passos dos processos, como os usuários vão utilizar o sistema ou produto, e que para isso são usados diagramas esquemáticos, que serão descritos nos tópicos a seguir.

#### DIAGRAMA DE CASO DE USO

Segundo Pressman (2009, p. 137) “um caso de uso conta uma história estilizada sobre como o usuário final (desempenhando um de uma série de papeis possíveis) interage com o sistema sob um conjunto de circunstancias especificas” e Larman (2007, p. 20) complementa “um caso de uso é utilizado para estruturar o comportamento de itens em um modelo”.

O objetivo deste diagrama é representar graficamente o *software* de um ponto de vista diferente, utilizando de atores, que representam as diferentes pessoas ou dispositivos que usam o produto no contexto que será descrito, Pressman (2009, p. 137) define um ator como “qualquer coisa que se comunica com o sistema ou o produto e que é externa ao sistema em si”, esses atores podem interagir com o produto, executar tarefas, e essas ações que são representadas por elipses com uma curta e clara mensagem interna, que expõe a ideia principal da ação em questão e assim essa relação demonstra que o ator pode executar determinada ação.

Figura 2.7 – Exemplo de caso de uso



Fonte: (LARMAN, 2007)

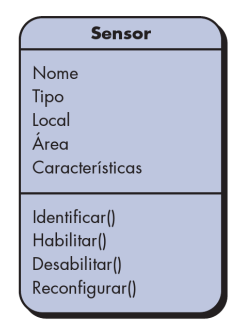
#### DIAGRAMA DE CLASSES

Pressman (2009, p. 166) afirma que a modelagem baseada em classes:

Representa objetos que o sistema irá manipular, as operações (também denominadas métodos ou serviços) que serão aplicadas aos objetos para efetuar a manipulação, os relacionamentos (alguns hierárquicos) entre os objetos e as colaborações que ocorrem entre as classes definidas.

O diagrama de classes visa representar graficamente esses objetos e suas operações, relacionamentos, hierárquicos ou não, e colaborações. Larman (2007, p. 18) descreve a representação gráfica de uma classe, “as classes são representadas por retângulos, geralmente incluindo seu nome, atributos e operações”.

Figura 2.8 – Exemplo de Diagrama de Classe



Fonte: (PRESSMAN, 2009)

Na figura 2.8, existe uma classe Sensor, cujo os atributos, também chamados de propriedades, são Nome, Tipo, Local, Área e Características, e seus métodos, ou operações, Identificar, Habilitar, Desabilitar e Reconfigurar que acessam os atributos da classe se necessário para efetuar seus devidos propósitos.

#### DIAGRAMA DE ATIVIDADES

O diagrama de atividades é descrito de forma parecida por Larman (2007, p. 483), Booch, Rumbaugh e Jacobson (2005, p. 28), que dizem que este diagrama tem por objetivo representar o comportamento de um sistema descrevendo a sequência de ações em um processo, são parecidos com fluxogramas, pois mostram o fluxo entre as ações em uma atividade, estes podendo ser paralelos ou simultâneos.

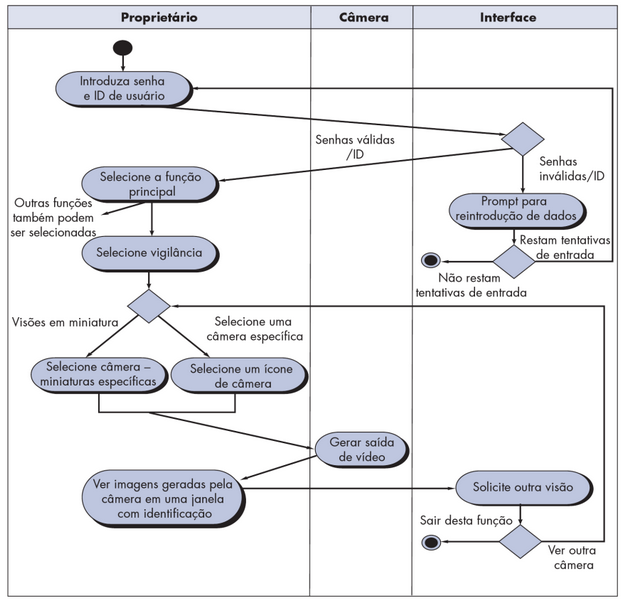
Pressman (2009, p. 161) acrescenta que este representa o “fluxo de interação em um cenário especifico” e o descreve:

Usa retângulos com cantos arredondados para representar determinada função do sistema, setas para representar o fluxo através do sistema, losangos de decisão com ramificação (cada seta saindo do losango é identificada) e as linhas horizontais cheias indicam as atividades paralelas que estão ocorrendo.

Porem graficamente é tão simples que acaba sendo muito intuitivo seu entendimento.

Para este trabalho foi optado por esquematizar o diagrama de atividades ao invés do diagrama de sequência, pois ambos têm objetivos muito semelhantes.

Figura 2.9 – Exemplo de Diagrama de Atividades



Fonte: (PRESSMAN, 2009)

## PROJETO DE INTERFACE

O principal objetivo de projeto de interface é sensibilizar os engenheiros de *software* sobre algumas questões básicas importantes para o projeto de interface com o usuário. O projeto centrado no usuário é uma abordagem de projeto de interface onde a análise das atividades do usuário é primordial para o sucesso do projeto como um todo, projeto de interface sempre envolve o desenvolvimento de protótipos de interfaces.

O termo interface é aplicado normalmente àquilo que interliga dois sistemas. Tradicionalmente, considera-se que uma interface homem-máquina é a parte de um artefato que permite a um usuário controlar e avaliar o funcionamento deste artefato através de dispositivos sensíveis às suas ações e capazes de estimular sua percepção. No processo de interação usuário-sistema a interface é o combinado de *software* e *hardware* necessário para viabilizar e facilitar os processos de comunicação entre o usuário e a aplicação.

A interface entre usuários e sistemas computacionais diferencia-se das interfaces de máquinas convencionais por exigir dos usuários um maior esforço cognitivo em atividades de interpretação e expressão das informações que o sistema processa. (SOUZA, LEITE, *et al.*, 1999, p. 3)

Muitas vezes, o designer deve identificar quais destes fatores têm prioridade sobre quais outros, uma vez que dificilmente se consegue alcançar todos de forma equivalente. As decisões do projetista determinam a forma de interação entre usuários e sistemas. Frequentemente designers definem a facilidade de uso como sendo o aspecto de usabilidade prioritário e, por vezes, acabam desenvolvendo sistemas em que os usuários não cometem erros, mas também não têm muita opção de ação ou decisão.

A meta do projeto de interface é definir um conjunto de objetos e ações de interface que permitam ao usuário realizar todas as tarefas estabelecidas, de um modo que satisfaça todas as metas de usabilidade definidas para o sistema. (PRESSMAN, 2009, p. 239)

Por mais que o levantamento de requisitos por si só levante uma infinidade de informações que deverão estar dispostas na interface, é durante a interação real do usuário com o sistema que os detalhes realmente são percebidos. Para projetar a experiência perfeita, a utilização de protótipos se faz a forma mais eficaz de simular essa interação, devendo ser feita ainda no período de concepção do sistema, evitando gastos dispendiosos de tempo e monetários.

Os protótipos de baixa fidelidade, também chamados de rascunhos ou sketches, são concebidos ainda na fase inicial, durante a concepção do sistema.

Desenhados geralmente à mão utilizando lápis, borracha e papel, essas representações são feitas de maneira rápida e superficial, apenas margeando a ideia do projeto e definindo superficialmente sua interação com o usuário, não se preocupando ainda com elementos de layout, cores, disposições, etc.

Os *mockups* ou protótipos funcionais constituem a representação mais próxima do sistema a ser desenvolvido. Em alguns casos, é possível simular o fluxo completo das funcionalidades, permitindo a interação do usuário como se fosse o produto final.

## DESENVOLVIMENTO DE *SOFTWARE*

Para entender o que é desenvolvimento de *software* é preciso saber o que é um *software*, Okuyama, Milleto e Nicolao (2014, p. 29) definem *software* como um programa que tem a finalidade de automatizar um processo ou resolver algum problema, Milleto e Bertagnolli (2014, p. 4) apresentam alguns exemplos, como, “um sistema que controla o metrô de uma cidade, um jogo de computador, um sistema de folha de pagamento de uma empresa”, entre outros.

Pressman e Maxim (2016, p. 4) utiliza de uma definição mais formal para explicar o que é *software*:

*Software* consiste em: (1) instruções (programas de computador) que, quando executadas, fornecem características, funções e desempenhos desejados; (2) estruturas de dados que possibilitam aos programas manipular informações adequadamente; e (3) informação descrita, tanto na forma impressa quanto virtual, descrevendo a operação e o uso do programa.

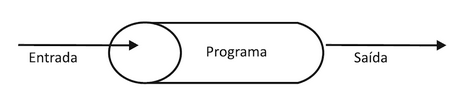
Milleto e Bertagnolli (2014, p. 4) afirmam que para “o desenvolvimento dessas aplicações, requer técnicas, métodos e mecanismos específicos para que elas funcionem conforme o esperado”. Essas técnicas, métodos e mecanismos são parte do processo de desenvolvimento de *software*, Milleto e Bertagnolli (2014, p. 21) afirmam que o uso de um processo é opcional, segundo Pressman e Maxim (2016, p. 17) o processo de desenvolvimento de *software* é “um conjunto de atividades, ações e tarefas realizadas na criação de um artefato”, mas lembrando de que um processo não é uma prescrição rígida de como desenvolver um *software*.

Portanto, desenvolvimento de *software* é, conforme citado por Milleto e Bertagnolli (2014, p. 21), um conjunto de atividades sistematizadas, que devem ser executadas em uma sequência pré-estabelecida de modo a agregar o máximo de valor possível a equipe envolvida no processo e ao produto que será produzindo com base na qualidade.

### PROGRAMAÇÃO ORIENTADA A OBJETOS (POO)

Silva Filho (2011, p. 1) define programação como uma atividade que tem por objetivo o “desenvolvimento de programas que implementam determinadas funcionalidades” e capazes de produzir uma saída, ou seja, algum resultado.

Figura 2.10 – Função de um programa



Fonte: (SILVA FILHO, 2011, p. 2)

Para Santos (2013) “Programação Orientada a Objetos, ou abreviadamente, POO é um paradigma de programação de computadores onde se usam classes e objetos” para “representar e processar dados usando programas de computadores”.

Silva Filho (2011, p. 11) explica, “uma classe serve como um padrão, modelo ou *template*”, ela “especifica quais dados e quais funções serão incluídos nos objetos daquela classe”, e lembra que definir uma classe não cria objetos, afinal, a classe é na realidade uma abstração para criar modelos baseados no mundo real.

A POO pode ser vista como o projeto de *software* utilizando uma coleção de objetos em cooperação, em oposição a uma vista tradicional, em que um programa pode ser visto como uma série de funções, ou simplesmente como uma lista de instruções para o computador. Em POO, cada objeto é capaz de receber mensagens, processar dados e enviar mensagens para outros objetos. Cada objeto pode ser visto como uma pequena máquina independente, com um papel ou responsabilidade distinta.

A POO se destina a promover uma maior flexibilidade e facilidade de manutenção na aplicação, e é muito popular em engenharia de *software*s de grande escala. Em virtude de sua forte ênfase na modularidade, código orientado a objetos destina-se a ser mais simples de desenvolver e mais fácil de entender mais tarde, prestando-se a uma análise mais direta, codificação e compreensão de situações e procedimentos mais complexos do que nos métodos de programação menos modulares. (TRASVIÑA, 2005)

### PROGRAMAÇÃO BASEADA EM PROTÓTIPOS

Segundo a Trasviña (2005), “Programação baseada em protótipos é um estilo de programação orientada a objetos na qual não existe a presença de classes” e lembra que “Este modelo também é conhecido como sem classes, orientado a protótipo, ou programação baseada em exemplares”. A relação de herança desse paradigma é realizada através do processo de decorar ou expandir objetos que servem como protótipo, ou seja, um objeto base, o protótipo, decora outro objeto, que permite que esse outro objeto utilize de seus recursos e funcionalidades.

O exemplo de uma linguagem baseada em protótipo citada por Trasviña (2005) é a linguagem de programação Self e logo afirma que este estilo de programação tem se tornado mais popular recentemente e vem sendo adotado por outros linguagens como *JavaScript, NewtonScript*, Cecil, entre outras.

### INTERATIVIDADE NA *WEB*

Nieradka (2014) lembra que o desenvolvimento *web* era puramente estático, mas que com o surgimento do *JavaScript*, ou JS, as páginas *web* passaram a ser interativas, possibilitando validação de formulários, caixas de diálogo, e muitos outros recursos.

Remoaldo (2008, p. 31) complementa, a interatividade é disponibilizada pela tecnologia Flash também, mas a diferença principal é que o *JavaScript* é integrado ao navegador e não precisa de um plug-in terceiro como o Flash que necessita do Adobe Flash Player.

Mas como é feita a interação do usuário com as páginas *web*? A interação é dada por meio dos dispositivos periféricos conectados aos computadores, smart*Phone*s e tablets, por exemplo, teclado, mouse, sensores de toque, *touchpad*, microfone, entre outros, sendo os mais comuns, teclado e mouse.

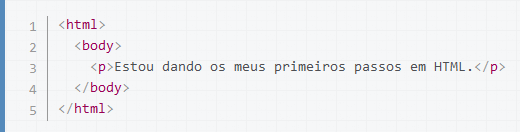
### TECNOLOGIAS

Existem muitas tecnologias disponiveis para uso no desenvolvimento de aplicações *web*, para diversas finalidades, algumas com desempenho melhor que outras em determinadas situações. A seguir serão listadas as tecnologias utilizadas neste trabalho.

#### *HTML*5

Para Milleto e Bertagnolli (2014, p. 62) o *HTML*, ou *Hypertext Markup Language* traduzida como Linguagem de Marcação de Hipertexto, é “uma linguagem de marcação utilizada para criar páginas acessadas a partir de um navegador”. MDN (HTML) explica, o *HTML* é o bloco de construção mais básico de uma página *web*, é usado para criar e representar visualmente uma página. Ele determina o conteúdo de uma página, mas não sua funcionalidade.

Figura 2.11 – Exemplo de documento *HTML*



Fonte: MDN (Introdução ao HTML)

A última versão estável do *HTML*, o *HTML*5 leva o *HTML* de uma marcação simples de estruturação de um documento para uma completa plataforma de desenvolvimento de aplicativos. Entre outras características, o *HTML*5 inclui novos elementos e *APIs*, Application Programming Interface e trazudido como Interface de Programação de Aplicativos, *JavaScript* para melhorar o armazenamento, multimídia e acesso ao *hardware*. (MDN *HTML*5)

#### *Canvas*

Para Cássio (2014, p. 1) *Canvas* é “uma área retangular em uma página *web* onde podemos criar desenhos programaticamente, usando *JavaScript*”, a figura 2.12 mostra bem isso. Essa área desenhável pode ser usada, por exemplo, para desenhar gráficos, fazer composição de fotos, criar animações ou até mesmo fazer processamento de vídeo em tempo real.

Figura 2.12 – Exemplo de código *Canvas* e sua renderização



Fonte: MDN (Canvas API)

Segudo a documentação da MDN (Canvas API) o *Canvas* foi originalmente criado pela Apple para seus widgets do Dashboard, mas desde então tem sido adotado pelos principais navegadores e agora faz parte do *HTML*5.

### ANATOMIA DE UM VIDEOGAME

A documentação apresentada pela MDN (Anatomia de um vídeo game) explica, o objetivo de cada jogo é apresentar ao usuário, uma situação, indentificar sua entrada, interpretar esses sinais em ações, e calcular uma nova situação resultante dos referidos atos. Os jogos estão constantemente em um ciclo através destes estágios, varias e varias vezes, até que alguma condição final ocorra, como, ganhar ou perder. Não surpreendentemente, este padrão corresponde a como o nucleo de jogo está programado.

Mais adiante explica, alguns jogos conduzem este ciclo através da entrada do usuário, por exemplo, um jogo dos sete erros. Este jogo apresenta duas imagens, aceita cliques ou toques na tela, interpreta a entrada como um sucesso, fracasso, pausa, interação de menus, etc. Finalmente, ele calcula uma cena atualizada resultante dessa entrada. O ciclo do jogo é avançado pela entrada do usuário e espera até que a proxima seja fornecida. Esta é uma abordagem baseada em turnos, que não exige uma atualização constante a cada segundo, somente quando o jogador reage.

Outros jogos exigem controle sobre a menor fatia de tempo possivel. Os mesmos princípios se aplicam como anteriormente citado com uma leve diferença: cada quadro de animação progride o ciclo e qualquer alteração na entrada do usuário é capturada no primeiro quadro disponível. Este modelo é implementado em algo chamado de laço principal. Se os ciclos do jogo forem baseados no tempo, então o jogo irá aderir a este modelo.

### TÉCNICAS

Existe uma vasta quantidade de técnicas que podem ser encontradas pela Internet, porem algumas são importantes demais para serem deixadas de lado, são essenciais para qualquer jogo digital em desenvolvimento. A seguir serão listadas as principais e mais conhecidas segundo a MDN (Técnicas para desenvolvimento de jogos).

#### ANIMAÇÕES EFICIENTES PARA JOGOS *WEB*

De acordo com a MDN (Efficient animation for web games) essa técnica aborda meios de criar animações eficientes para jogos *web*, com uma maior ênfase em dispositivos com poder de processamento inferior, como telefones celulares.

Algumas dessas técnicas podem ajudar a aplicação a consumir menos bateria e recursos de dispositivos de baixo desempenho, e assim suportando uma quantidade maior de dispositivos.

#### INICIALIZAÇÃO OTIMIZADA

Sobre esta tecnica a MDN (Optimizing startup performance) lembra que um aspecto muitas vezes esquecido no desenvolvimento de aplicações, mesmo entre aqueles com foco na otimização do desempenho é o desempenho de inicialização. Quanto tempo um aplicativo necessita para a inicialização? Parece que ele trava o dispositivo ou o navegador do usuário enquanto o aplicativo carrega?

Independentemente da plataforma, é sempre uma boa ideia iniciar o mais rápido possível e esta técnica oferece dicas e sugestões para garantir que a inicialização de uma aplicação ocorra r*API*damente, sem travamentos e erros inesperados tanto ao escrever um novo aplicativo quanto ao portar para outra plataforma.

#### ÁUDIO PARA JOGOS *WEB*

Destacada pela MDN (Audio for Web games), a técnica de Áudio é uma parte importante de qualquer jogo, ele adiciona *feedback* e atmosfera. O áudio baseado em tecnologias *web* está amadurecendo rápido, mas ainda há muitas diferenças de navegador para navegador. Muitas vezes é necessário decidir quais áudios são essenciais para uma boa experiência de jogo e quais são agradáveis de ter, mas não essenciais, e elaborar uma estratégia nesse sentido.

De longe, as plataformas mais difíceis para fornecer suporte de áudio na *web* são as plataformas móveis. Infelizmente, estas são também as plataformas que as pessoas em sua maioria usam para jogar, e essa técnica tenta minimizar ao máximo estes problemas.

#### DETECÇÃO DE COLISÃO 2D

Esta técnica consiste em utilizar algoritmos para detectar colisões 2D, independente do tipo de formas que podem colidir, por exemplo, entre retangulos, entre um retangulo e um círculo, entre cirulos, e ainda colisões entre poligonos irregulares.

Esta tecnica apresenta os metodos mais comuns e de alto desempenho utilizados para proporcionar a detecção de colisão dos diversos tipos de formas.

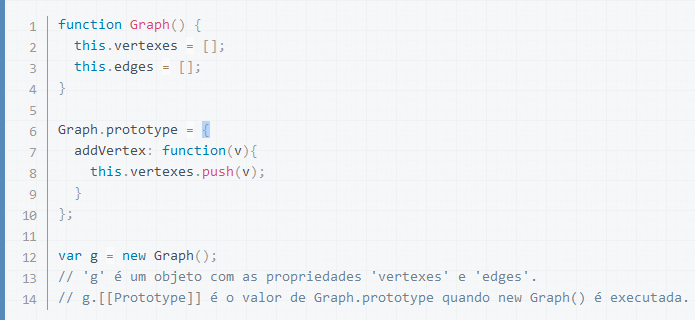
### FERRAMENTAS DE DESENVOLVIMENTO

Uma vez que a Internet evoluiu ao longo dos anos, as tecnologias por ela utilizadas também evoluíram, e isso permitiu que o número de ferramentas para o desenvolvimento de aplicações *web* acompanhasse esse crescimento, a seguir são apresentadas algumas ferramentas disponíveis, escolhidas pela MDN (Ferramentas para o desenvolvimento de jogos) para facilitar a criação de experiências de jogos na *web*, tais como frameworks e linguagens de programação *web*.

#### *JavaScript*

Criado em 1995 por Brendan Eich, Remoaldo (2008, p. 35) explica, a linguagem *JavaScript* “é uma linguagem de *scripting* interpretada que está disponível de forma gratuita na maioria dos browsers modernos”. A MDN (Introdução) descreve o *JavaScript* como uma linguagem orientada a objetos, multiplataforma, pequena e leve, onde, dentro de um ambiente de *host*, por exemplo, um navegador *web*, pode ser ligado aos objetos deste ambiente para prover um controle programático sobre eles.

Figura 2.13 – Exemplo de código *JavaScript*



Fonte: MDN (Herança e cadeia de protótipos (prototype chain))

Segundo documentação da MDN (Introdução), núcleo do *JavaScript* pode ser estendido para uma variedade de propósitos, complementando assim a linguagem:

O lado cliente do JavaScript estende-se do núcleo da linguagem, fornecendo objetos para controlar um navegador *web*. Por exemplo, as extensões do lado do cliente permitem que uma aplicação coloque elementos em um formulário *HTML* e responda a eventos do usuário, como cliques do mouse, entrada de formulário e de navegação da página.

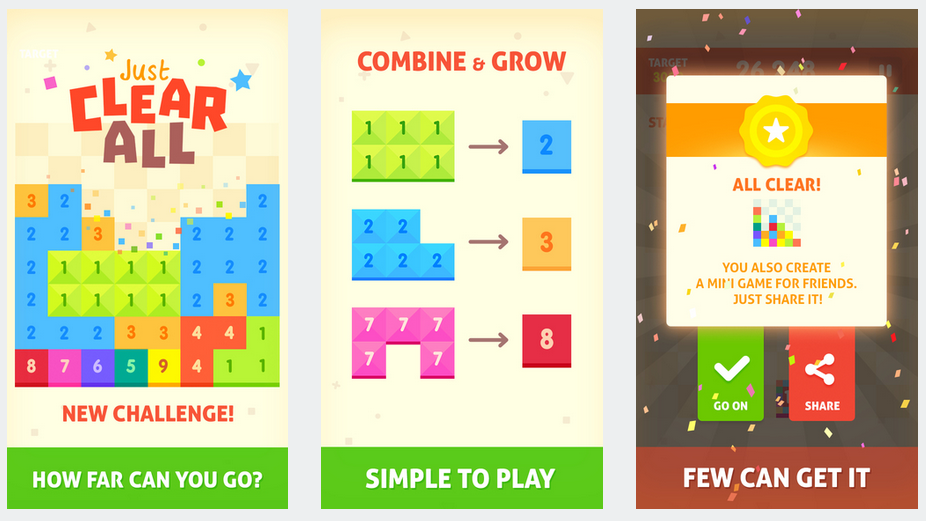
O lado do servidor do JavaScript estende-se do núcleo da linguagem, fornecendo objetos relevantes à execução do *JavaScript* em um servidor. Por exemplo, as extensões do lado do servidor permitem que uma aplicação se comunica com um banco de dados, garantindo a continuidade de informações de uma chamada para a outra da aplicação, ou executar manipulações de arquivos em um servidor.

O *JavaScript* foi escolhido para este trabalho porque não exige nenhuma extensão ou biblioteca terceira para o desenvolvimento de aplicações *web*. Como está presente em todos os navegadores modernos, por exemplo, Mozilla Firefox, Google Chrome, Opera e Microsoft Edge, e por ser uma linguagem multiplataforma, permite um alcance maior de usuários, e permiti ao programador maior liberdade ao estruturar e codificar a aplicação, ao contrário de um framework, que pode restringir o programador em diversos pontos.

#### Cocos2d-JS

Cocos2d-JS[[3]](#footnote-3) é um motor de código aberto para criar jogos *web* e nativos. Tem uma alta performance, é amigável com o usuário e suporta o desenvolvimento multiplataforma. As plataformas suportadas incluem *web*, *Android*, *iOS*, *Windows* *Phone* 8, *Mac*, *Windows*, etc. O *Cocos2d-JS* faz a programação de jogos 2D mais fácil e rápida. Ele simplifica os componentes principais da programação de jogos 2D sendo fácil de aprender e possui uma de fácil uso.

Figura 2.14 – Jogo criado com Cocos2d-JS



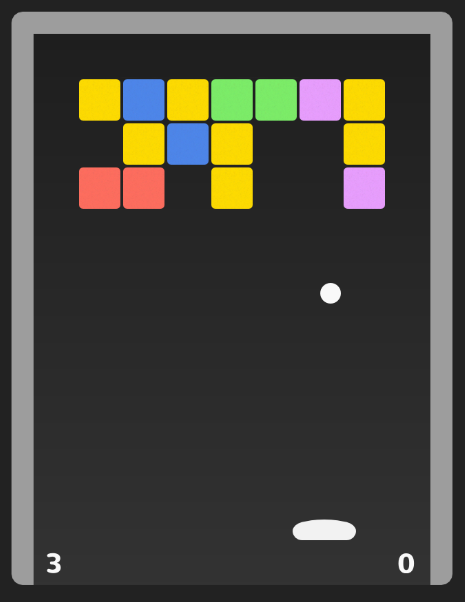
Fonte: http://cocos2d-x.org/games/3036

*Cocos2d-JS* é a versão *JavaScript* do *Cocos2d-x*, que também é um motor de jogos, de código aberto. Ele permite aos desenvolvedores explorar o seu conhecimento em C++, Lua e *JavaScript* com o desenvolvimento em plataforma cruzada para iOS, Android, *Windows* *Phone*, *Mac* OS X, *Windows* *Desktop* e Linux, poupando tempo, esforço e custo.

#### Stage.js

*Stage.js*[[4]](#footnote-4) é uma biblioteca 2D *HTML*5 escrita em *JavaScript* para desenvolvimento de jogos multiplataforma, ela é leve, rápida e de código aberto. Internamente gerencia ciclos de renderização e de desenho da aplicação, também processa e distribui para os respectivos responsáveis os eventos de clique e toque.

Figura 2.15 – Jogo desenvolvido com Stage.js



Fonte: http://piqnt.com/stage.js/example/game-breakout/

#### Isogenic Engine

*Isogenic Engine*[[5]](#footnote-5) é um sistema *JavaScript*, executado do lado do cliente e também do servidor, que fornece um grafo de cena, com uma funcionalidade de multijogadores embutida. Simulações em execução no lado do servidor podem ser transmitidas automaticamente para todos os clientes ligados à rede ou seccionado em salas distintas em que apenas os clientes que estão participando recebem as atualizações.

O motor apoia completamente tanto renderização 2D quanto isométrica e pode ser executado com ou sem funcionalidade multijogador. Tem um sistema de construção incluído para analisar automaticamente o código final do jogo, que detecta as seções da *API* do *Isogenic* que tenham sido usadas e minifica o jogo e o Isogenic, para obter o menor tamanho possível do jogo já pronto para distribuição.

# METODOLOGIA

Prodanov e Freitas (2013, p. 14) descrevem metodologia como “uma disciplina que consiste em estudar, compreender e avaliar os vários métodos disponíveis para a realização de uma pesquisa acadêmica” e afirmam que seu objetivo é avaliar métodos e técnicas de pesquisa que possibilitam a coleta e o processamento de informações para solucionar problemas e/ou questões de investigação.

Silva e Menezes (2005) em um artigo da UFSC concluem que “Pesquisa é um conjunto de ações, propostas para encontrar a solução para um problema, que têm por base procedimentos racionais e sistemáticos” e é realizada “quando se tem um problema e não se têm informações para solucioná-lo”.

Segundo Gil (2002, p. 43-45) existe várias formas de classificar as pesquisas. Do ponto de vista dos procedimentos técnicos:

* Pesquisa Bibliográfica, que é elaborada com base em material já existente, como livros, artigos científicos, e disponibilizados na internet, o caso deste trabalho.
* Pesquisa Documental, desenvolvida a partir de materiais que não receberam tratamento analítico.
* Estudo de caso, quando é necessário o estudo profundo de maneira que se permita o seu amplo e detalhado conhecimento.

E muitas outras como, *Expost-Facto*, Experimental, Participante, etc.



## INSTRUMENTO DE COLETA DE DADOS

Para a coleta das informações necessárias para o levantamento de requisitos foi aplicada a técnica de entrevista, os resultados obtidos podem ser visualizados na seção 4.1 deste trabalho.

## CRONOGRAMA

A seguir o cronograma com as atividades desenvolvidas durante todo o trabalho.

Tabela 3.1 – Cronograma das atividades desenvolvidas

|  |  |  |
| --- | --- | --- |
| **Meses** | **Atividade** | **Estado** |
| Janeiro de 2017 | Entrevista | Concluído antecipadamente |
| Janeiro de 2017 | Levantamento de requisitos | Concluído dentro do prazo estipulado |
| Fevereiro de 2017 | Modelagem estática (Diagrama de classes e Casos de Uso) e dinâmica (Diagrama de Atividades) | Concluído antecipadamente |
| Março de 2017 | Protótipos de tela de baixa e alta fidelidade (Mockups) | Concluído dentro do prazo estipulado |
| Março/Abril/Maio de 2017 | Desenvolvimento do motor da aplicação | Concluído com um pouco de atraso |
| Maio/Junho de 2017 | Implementação das telas | Concluído dentro do prazo estipulado |
| Julho/Agosto de 2017 | Teste de unidade e integração | Concluído dentro do prazo estipulado |
| Setembro de 2017 a Junho de 2018 | Documentação do trabalho | Concluído com muito atraso |

Fonte: Autores deste trabalho

## ATIVIDADES

Neste tópico será apresentado, de forma resumida, o que aconteceu em cada passo do desenvolvimento do trabalho, já os resultados serão apresentados no próximo c*apí*tulo.

* Entrevista: Nesta etapa visava-se recolher o máximo de informações dos interessados no projeto de forma a garantir que todas as expectativas fossem atingidas ao final do trabalho, ou seja, que nenhum detalhe fosse deixado de lado.
* Levantamento de requisitos: Com base nos dados coletados na fase anterior foram criadas as fichas de requisitos, presentes no Apêndice A, que descrevem melhor cada uma das necessidades levantadas pelos interessados.
* Modelagem estática e dinâmica: Foram criados diagramas para representar os objetos do sistema e as relações entre eles, a sequência de passos executada durante os processos do jogo e as suas interações com os jogadores.
* Protótipos de tela de baixa e alta fidelidade: Tinha a finalidade de criar protótipo de telas do jogo, como tela inicial, de fim de jogo, a própria tela onde o jogo iria ocorrer, entre outras, onde o protótipo de baixa fidelidade era apenas um simples esboço que serviria de base para as telas de alta finalidade.
* Desenvolvimento do motor da aplicação: Etapa de pura programação, basicamente todos os dados criados até então se tornaram código e deram vida ao jogo.
* Implementação das telas: Com base nos protótipos de tela anteriormente elaborados foram codificadas as telas, ou seja, foram construídas programaticamente.
* Documentação do trabalho: Elaboração deste documento.

# DESENVOLVIMENTO

Nesta etapa são apresentados os resultados obtidos durante o desenvolvimento deste trabalho.



## ENTREVISTA

A ideia do jogo *To Infinity and Beyond*, foi criada visando o mercado atual dos jogos digitais, pois eles estão cada vez mais fortes, criativos e bem desenhados, a ideia principal é unir recursos sonoros e gráficos de qualidade com estilo de jogo *arcade*, dos fliperamas, onde qualquer um pode jogar, em pouco tempo, ou seja, partidas rápidas e sem compromisso.

**Como o jogo deve ser basicamente?**

Ser de rolagem vertical, ao estilo do antigo jogo *Space Invaders*.

**O que se espera relacionado a arte do jogo?**

Que tenha recursos sonoros e gráficos de qualidade e podem ser terceirizados.

**Como deve ser o personagem do jogador?**

O personagem do jogador deve ser uma nave.

**Quais são os comandos que o jogador pode dar?**

Disparar lasers utilizando a tecla Z, se movimentar usando as setas do teclado, em duas velocidades, normal e lenta combinando as setas com a tecla *Shift* para a velocidade lenta, e em qualquer direção sem sair da tela do jogo.

**O disparo da nave pode ser melhorado?**

O disparo da nave pode ser melhorado coletando um tipo especifico de item.

**Como deve ser a tela inicial do jogo?**

A tela inicial deve conter três opções, *Arcade*, iniciar uma partida sem objetivos, *Story*, inicia uma história, um conjunto de fases com objetivos específicos, e tutorial, explicando como jogar.

**Como devem ser os inimigos?**

Inimigos, quantidades e tipos ficam a cargo dos desenvolvedores desde que estejam relacionados ao tema espaço.

**O que acontece quando se atinge um inimigo?**

Ao atingir um inimigo, devem surgir itens em seu lugar.

**O que os itens devem fazer?**

Devem existir três tipos de itens que valem ponto, estes podem valer 50, 150 ou 300 pontos, item multiplicador que deve incrementar o multiplicador do jogador, e o item de vida extra que incrementa o contador de vidas e este deve ser muito raro.

**O que deve conter na tela do jogo?**

Contador de vidas do jogador, contador de multiplicadores e contador de pontos.

**Como funcionam as vidas do jogador?**

As vidas do jogador inicialmente devem ser três, e a cada vez que a nave do jogador for atingida deve ser decrementada uma vida.

**Como funciona o contador de multiplicadores?**

Ele irá multiplicar os pontos obtidos a partir de coletas de pontos ou derrotando inimigos, deve funcionar da seguinte forma, toda vez que a nave coletar um ponto ou atingir um inimigo o valor equivalente em pontos do inimigo ou ponto deve ser multiplicado pelo multiplicador e somado a pontuação, por exemplo, o jogador atinge um inimigo de 50 pontos, e tem um multiplicador de cinco, o valor de 250, ou seja, 50 vezes 5, pontos será somado a pontuação.

**Como funciona o contador de pontos?**

Ele deve armazenar a pontuação obtida do jogador.

**O que deve acontecer quando o jogador perde uma vida?**

Ter uma tela de continuar, que deve ser exibida toda vez que a nave do jogador for atingida e o jogador ainda tiver vidas restantes, deve ter duas opções, sim, continuar jogando, e não, fim de jogo.

**O que deve acontecer quando o jogador perde todas as vidas?**

Ter uma tela de fim de jogo, quando o jogador não tiver mais vidas restantes e for atingido ou se na tela de continuar o jogador decidir não continuar jogando, a tela de fim de jogo deve ser exibida, está deve conter as seguintes opções: Novo Jogo, que deve iniciar uma nova partida, e voltar a Tela Inicial que volta para a tela de início.

**O jogo deve ter uma tela de pausa?**

Sim e deve ser exibida ou escondida quando o jogador pressionar a tecla P.

**O que deve acontecer visualmente quando os inimigos forem atingidos?**

Deve ter animações para quando os inimigos forem atingidos pelos disparos os e itens sendo lançados na tela

**O jogo deve conter fases?**

O jogo deve conter no mínimo 3 fases, os desafios, sequencias e história fica a cargo dos desenvolvedores.

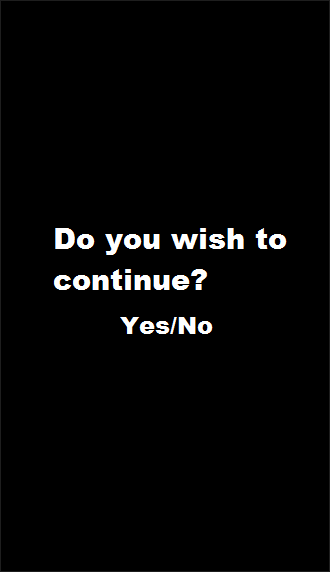
## LEVANTAMENTO DE REQUISITOS

Como resultados desta atividade, utilizando-se das informações levantadas na etapa de entrevista, foram elaboradas as fichas de requisitos que estão anexadas a este trabalho na forma de um apêndice, vide apêndice A.

## DESENVOLVIMENTO DA INTERFACE

Com base na entrevista e requisitos levantados na fase inicial do trabalho foram elaborados os protótipos de baixa e alta fidelidade. Todas as telas do jogo são muito simples, portanto, produzi-las não foi uma tarefa tão complicada, principalmente as telas de baixa fidelidade.

Figura 4.1 – Protótipo de baixa fidelidade da tela de continuação



Fonte: Autores deste trabalho

A figura acima apresenta o protótipo da tela de continuação, que pergunta ao jogador se ele deseja continuar, ela será exibida sempre que o jogador perder um crédito a não ser que ele já tenha perdido todos, nesse caso será exibida a tela de fim de jogo, vide figura 4.2. Na tela de continuação o jogador pode escolher apenas entre duas opções, *yes,* sim e *no*, não.

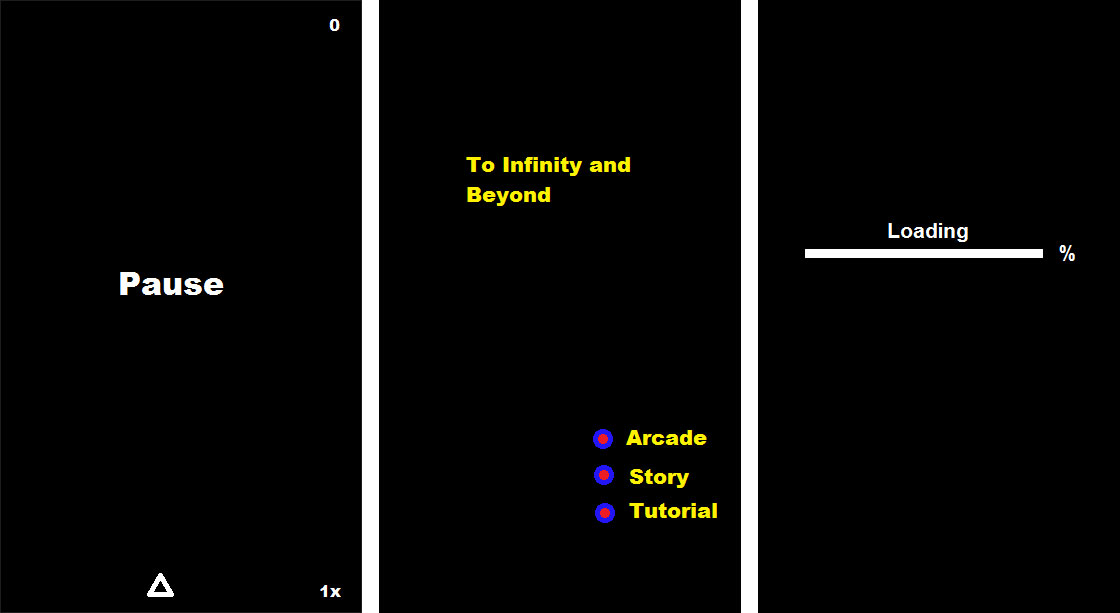
Figura 4.2 – Protótipo de baixa fidelidade da tela de fim de jogo



Fonte: Autores deste trabalho

As telas pausa, inicial e carregamento são bem semelhantes, possuem apenas pequenos trechos de textos com poucas ou até mesmo sem opções:

Figura 4.3 – Protótipo de baixa fidelidade das telas de pausa, inicial e carregamento



Fonte: Autores deste trabalho

Os protótipos de baixa fidelidade foram todos desenvolvidos no *Microsoft Paint* enquanto os protótipos de alta fidelidade foram desenvolvidos no *Adobe Photoshop CS6*.

As diferenças entre os protótipos de baixa e alta fidelidade das telas de pausa, continuação, fim de jogo e carregamento, resume se a definição de uma nova fonte para os textos e do fundo agora possui algumas estrelas, tentado dessa forma reproduzir o espaço.

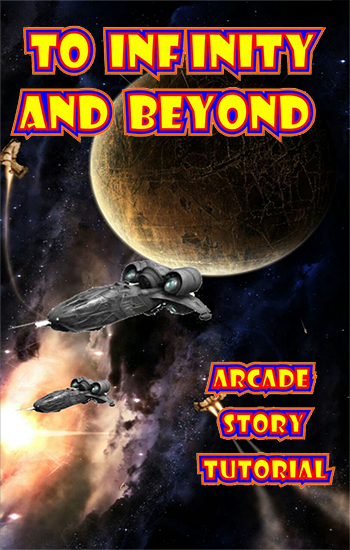
Figura 4.4 – Protótipos de alta fidelidade das telas de pausa, continuação, fim de jogo e carregamento



Fonte: Autores deste trabalho

E por fim o protótipo de alta fidelidade da tela inicial, que ganhou uma arte de fundo e fontes diferentes.

Figura 4.5 – Protótipo de alta fidelidade da tela inicial



Fonte: Autores deste trabalho

## DESENVOLVIMENTO DO MOTOR DA APLICAÇÃO

O motor da aplicação foi desenvolvido, conforme citado anteriormente, utilizando a linguagem programação *web*, *JavaScript*, combinada com o paradigma de Orientação a Objetos.

Nenhuma ferramenta especifica foi utilizada, apenas um editor de texto moderno para acelerar e simplificar o processo de codificação, foi escolhido o *Sublime Text 3*[[6]](#footnote-6).

O código fonte completo possui 1656 linhas, por ser muito extenso foi anexado a este trabalho na forma de um apêndice, apêndice C.

### LAÇO PRINCIPAL

Uma boa pratica, ou padrão de projeto, da linguagem de programação *JavaScript* é a de envolver o código de uma aplicação por uma função imediata, uma função imediata é uma função que é executada logo que sua declaração termina, assim como no trecho de código a seguir:

(function () {

var MG = {};

[...]

}());

Ainda neste trecho é possível perceber também que uma única variável foi definida, MG, é outro padrão de projeto da linguagem muito comum que sugere definir apenas uma variável para toda a aplicação, evitando poluir o escopo e simplificando sua exportação para outras aplicações.

O jogo utiliza o modelo de ciclo anteriormente citado, ciclo com base em fatias de tempo, por isso possui um laço principal:

MG.main = function () {

}

O laço principal é encarregado de algumas atividades, como, verificar e atualizar o estado atual do jogo e redesenhar a tela resultante de acordo com o respectivo estado:

MG.main = function () {

var status = MG.game.status;

requestAnimationFrame(MG.main);

if (status === 'paused') {

return;

}

MG.game.draw(0).update();

switch (status) {

case 'playing':

[...]

case 'loading':

case 'gameover':

case 'continue':

case 'start':

case 'tutorial':

MG.game.screen[status]();

}

};

No trecho acima, a variável *status* recebe o estado atual do jogo e faz diversas comparações no decorrer do código, verificando se está carregando, na tela inicial, no tutorial, etc. A função *requestAnimationFrame*, é responsável por chamar a função fornecida como parâmetro, informada entre parênteses, no caso, a função *MG.main* que irá chamar a si própria de novo e de novo, transformando assim a *MG.main* em um laço.

As funções *update* e *draw*, atualizar e desenhar respectivamente, são duas funções que estão presentes em muitos dos objetos do jogo, muitas vezes com logicas semelhantes e outras completamente diferentes. No trecho de código apresentado anteriormente, a imagem de fundo do jogo é atualizada e desenhada por estes métodos do objeto *game*.

MG.game.draw(0).update();

O laço principal também verifica se determinadas teclas foram pressionadas e executa alguma lógica para a determinada ação, também faz chamadas a processos terceiros que desenham e atualizam os inimigos, itens, explosões, disparos, a interface do usuário e a nave do jogador:

MG.main = function () {

[...]

switch (status) {

case 'playing':

if (MG.joystick.pressed[Z]) {

MG.ship.shoot.create();

}

if (MG.joystick.pressed[SHIFT] && MG.ship.step !== 1) {

MG.ship.step = 1;

}

else if (!(MG.joystick.pressed[SHIFT]) && MG.ship.step === 1) {

MG.ship.step = 3;

}

MG.ship.update().draw();

MG.ship.shoot.update().draw();

MG.meteor.update().draw();

MG.ufo.update().draw();

MG.ufo.shoot.update().draw();

MG.explosion.update().draw();

MG.item.update().draw();

MG.player.ui();

break;

[...]

O objeto *joystick* é o responsável por toda interação do jogador efetuada através do teclado, os objetos *ship*, *ship.shoot*, *meteor*, *ufo*, *ufo.shoot*, *explosion*, *player* e *item* são responsáveis por desenhar e atualizar os inimigos, itens, disparos, etc. Todos esses objetos serão abordados em breve.

### RECURSOS Externos

Os gráficos e efeitos sonoros são arquivos externos armazenados nas suas respectivas pastas dentro da pasta da aplicação, pasta *app*, e são carregados para dentro dela através de duas funções.

function GameImage (file) {

var o = new Image();

o.src = file;

o.onload = function () {

MG.srcLoaded++;

MG.loaded = (109 \* MG.srcLoaded / MG.srcToLoad) + 1;

};

return o;

};

A variável *o* recebe um novo objeto de tipo imagem e logo em seguida o caminho para o arquivo é definido:

o.src = file;

Então o navegador começa o processo de carregamento da imagem, quando o processo terminar a função *onload* é executada, está por sua vez incrementa a variável *MG.srcLoaded*, que conta o número de arquivos carregados, e a variável *MG.loaded* armazena o valor de pixels que deve ser preenchido da barra de progresso na tela de carregamento.

function GameAudio (file) {

var a = new Audio();

a.src = file;

a.status = setInterval(function () {

if (a.readyState === 4) {

MG.srcLoaded++;

MG.loaded = (109 \* MG.srcLoaded / MG.srcToLoad) + 1;

clearInterval(a.status);

if (MG.srcLoaded === MG.srcToLoad) {

setTimeout(function() {

MG.game.status = 'start';

MG.game.screen.change(2, 'arcade');

MG.sounds[6].started = true;

MG.sounds[6].loop = true;

playSound(6);

}, 1000);

}

}

}, 500);

return a;

}

O carregamento de arquivos de áudio é muito semelhante ao processo de imagens, a diferença aqui é que quando *MG.srcLoaded*, número de arquivos carregados, for igual a *MG.srcToLoad*, total de arquivos a serem carregados, o estado do jogo é alterado para *start*, e a tela inicial é apresentada.

if (MG.srcLoaded === MG.srcToLoad) {

setTimeout(function() {

MG.game.status = 'start';

MG.game.screen.change(2, 'arcade');

[...]

}, 1000);

}

### CONTROLES

O objeto *joystick* foi criado para identificar e manipular todas as interações do jogador através do teclado, ele escuta determinadas teclas, as setas que movimentam a nave nas quatro direções, direita, esquerda, cima e baixo, a tecla Shift para mudar a velocidade da nave, P para pausar o jogo e Z para efetuar disparos.

MG.joystick = {

// shift, setas e tecla Z

listening: [SHIFT, 37, 38, 39, 40, Z],

pressed: [], // teclas pressionadas

[...]

};

Além de escutar as teclas o objeto *joystick* faz chamadas aos métodos pré-definidos ou executa uma ação de logica simples para cada tecla pressionada:

MG.joystick = {

[...]

keyDown: function (e) {

var k = e.which;

if (typeof this.pressed[k] === 'function') {

return this.pressed[k]();

}

else if (this.listening.indexOf(k) !== -1) {

this.pressed[k] = true;

}

},

keyUp: function (e) {

if (this.listening.indexOf(e.which) !== -1) {

this.pressed[e.which] = false;

}

}

[...]

};

O método *keyDown* armazena em uma propriedade do objeto *joystick* as teclas que foram pressionadas ou executa uma função previamente atribuída a tecla em questão. A função *keyUp* simplesmente apaga da propriedade o registro de uma tecla após esta ser liberada, ou seja, quando deixa de ser pressionada. Existe ainda um terceiro método, *keyDownMenu*, responsável por alternar entre as opções dos menus, onde esses existem, por exemplo, na tela inicial, de fim de jogo e de continuar.

### JOGO

O objeto *game* é responsável pela renderização e a lógica das diversas telas que o jogo pode apresentar ao jogador, por exemplo, a tela inicial, continuação, fim de jogo, carregamento, entre outras.

MG.game = {

bg: MG.bg,

ctx: MG.ctx,

status: 'loading',

y: 0, // posição do fundo

screen: {

[...]

},

[...]

};

Este objeto possui uma propriedade, *screen*, que nada mais é que outro objeto que tem o objetivo de reunir os métodos responsáveis por apenas desenhar as diversas telas, por exemplo, a tela de continuação:

screen: {

[...]

continue: function () {

var x = 0, y = 0,

font = '18px Bebas Neue',

half\_width = Math.floor(this.width / 2),

half\_height = Math.floor(this.height / 2);

this.ctx.textAlign = 'left';

this.ctx.fillStyle = 'white';

this.ctx.font = '30px Bebas Neue';

this.ctx.fillText('Do you wish to', half\_width - 72, half\_height);

this.ctx.fillText('Continue?', half\_width - 72, half\_height + 25);

//inverte opção selecionada

if (this.chosen === 'no') {

y = 8;

x = 18;

this.ctx.font = '18px Bebas Neue';

font = '30px Bebas Neue';

}

this.ctx.fillText('Yes', half\_width + 13 + x, half\_height + (80 - y));

this.ctx.font = font;

this.ctx.fillText('/No', half\_width + 50, half\_height + (72 + y));

},

[...]

}

O trecho acima é responsável por escrever a pergunta, *Do you wish to continue?* E as opções disponíveis, *Yes* e *No*. O objeto *game.screen* também possui um método que é capaz de controlar a opção escolhida nas telas de fim de jogo, inicial e de continuação, a opção selecionada fica armazenada na propriedade *chosen* do objeto *game.screen*:

screen: {

[...]

change: function (o, chosen) {

this.options = [

['yes', 'no'], ['retry', 'menu'],

['arcade', 'tutorial']

][o];

this.chosen = chosen;

}

},

Assim como a maioria dos objetos, o objeto *game* possui os métodos *update* e *draw*, que nesta ocasião desenham e atualizam o fundo do jogo, também está presente a lógica das telas do jogo, por exemplo, a lógica da tela de continuação:

MG.game = {

[...]

continue: function () {

if (this.screen.chosen === 'yes') {

this.status = 'playing';

} else {

playSound(3);

this.status = 'gameover';

this.screen.change(1, 'retry');

}

},

[...]

};

Perceba que o nome do método aqui é o mesmo no objeto *game.screen*, porém neste método está presente somente a lógica da tela. No trecho acima, a opção escolhida é identificada e o estado atual do jogo é alterado de acordo com a esta opção.

Os métodos com a lógica para as demais telas possuem uma estrutura bem parecida, por isso não serão apresentados aqui, mas podem ser vistos no apêndice C, onde está presente o código fonte completo.

Por último, o objeto *game* possui a função *reset* que reinicia o jogo.

MG.game = {

[...]

reset: function () {

MG.enemy.clear();

MG.player.reset();

MG.ship.reset();

MG.item.reset();

MG.explosion.reset();

this.screen.change(0, 'yes');

this.status = 'playing';

}

}

Mas na realidade este método faz chamadas para outros métodos *reset* de outros objetos, ou seja, cada objeto é responsável por redefinir os valores padrão de suas propriedades, e por fim o estado atual do jogo e escolha padrão são redefinidos também.

### JOGADOR E NAVE

O objeto *player* possui dados relacionados ao jogador, como, vidas, multiplicador, pontuação e pontuação mínima para liberar um item do tipo vida:

MG.player = {

[...]

credits: 3, // vidas

mult: 1, // multiplicador

score: 0, // pontuação

new\_life: 1000000, // pontuação minima para liberar um credito

[...]

};

Possui alguns métodos, o *reset*, para restaurar os dados do jogador ao estado inicial:

MG.player = {

[...]

reset: function () {

this.credits = 3;

this.mult = 1;

this.score = 0;

},

[...]

};

Um método para desenhar números, que é utilizado em seu outro método, *ui*, que desenha na tela a interface de usuário, ou seja, as vidas, a pontuação e o multiplicador:

MG.player = {

[...]

ui: function() {

var i = 0,

m = this.mult;

// vidas restantes

for (; i < this.credits; i += 1) {

this.ctx.drawImage(

this.spritesheet, 241, 0, 13, 21,

(i \* 12 + 8), 10, 9, 14

);

}

// pontuação

this.numbers(this.score, 10);

// multiplicador

this.numbers(m, this.screen.height – 22);

this.ctx.drawImage(

this.spritesheet, 221, 1, 17, 17,

this.screen.width – (m.toString().length \* 13 + 17),

(this.screen.height – 20.5), 10, 10

);

}

};

Figura 4.6 – Interface de usuário destacada em vermelho



Fonte: Autores deste trabalho

Na imagem a cima é possível ver a interface de usuário destacada em vermelho, onde, os raios representam as vidas do jogador, os números no canto superior direito a pontuação, e no canto inferior direito o multiplicador que deve multiplicar os pontos obtidos, através da derrota de inimigos ou de pontos coletados, pelo valor do multiplicador acumulado e somar a pontuação.

O objeto responsável pela nave do jogador é o objeto *ship*:

MG.ship = {

[...]

// passo

step: 3,

// posição inicial

x: Math.floor(MG.width / 2) - 11,

y: MG.height - 38,

[...]

};

A propriedade *step* indica a velocidade que a nave se movimenta, neste caso 3px, mas pode chegar a apenas 1px, e *x* e *y* indicam a posição atual da nave na tela.

Também possui os métodos *draw* e *update*, o método *draw* como esperado, desenha a nave, já o método *update* verifica se a nave deve começar ou continuar a se mover. Possui também o método *reset*, que restaura a posição inicial da nave e apaga os disparos que esta possa ter efetuado.

São quatro os métodos responsáveis por mover a nave, *down*, *left*, *right* e *up*:

MG.ship = {

[...]

// movimenta a nave para baixo

down: function () {

this.y += this.step;

if (this.y + this.height > this.screen.height) {

this.y = this.screen.height - this.height;

}

},

// movimenta a nave para a esquerda

left: function() {

this.x -= this.step;

if (this.x < 0) {

this.x = 0;

}

},

// movimenta a nave para a direita

right: function() {

this.x += this.step;

if (this.x + this.width > this.screen.width) {

this.x = this.screen.width - this.width;

}

},

// movimenta a nave para cima

up: function() {

this.y -= this.step;

if (this.y < 0) {

this.y = 0;

}

}

};

Resumidamente, esses métodos incrementam ou decrementam o *step* das propriedades *x* e *y* do objeto *shi*p.

Os disparos da nave são controlados por uma propriedade, na realidade outro objeto, o *shoot*:

MG.ship = {

[...]

shoot: {

[...]

recover: 0, // intervalo entre os disparos

velocity: 6,

s: [],

[...]

},

[...]

};

O objeto *ship.shoot*, possui três propriedade que valem destacar, *recover* que é um tipo de contador regressivo, quando seu valor é zero significa que a nave pode efetuar um disparo, *velocity* indica a velocidade com a qual o disparo se move, e *s* que armazena todos os disparos da nave.

Este objeto também possui os métodos, *update* e *draw*, com uma pequena diferença, nesse caso ao invés de atualizar um objeto, ele atualiza vários, todos os disparos armazenados em *s*, a cada execução deste método a propriedade *recover* é decrementada por um, para que alguns instantes depois a nave possa realizar novamente outro disparo.

E quando um disparo estiver fora dos limites da tela ele é removido de *s*, este comportamento é comum entre vários objetos desta aplicação, pois uma vez fora da tela o objeto não está mais em jogo.

Possui um *reset,* que apaga todos os disparos da propriedade *s*, e um método capaz de criar novos disparos, o *create*:

shoot: {

[...]

// cria e adiciona um tiro ao vetor de tiros da nave

create: function () {

if (this.recover === 0) {

this.s.push({

height: this.height,

width: this.width,

// posição x da nave mais a metade de sua largura

// menos a metade da largura do tiro

x: this.uber.x + Math.floor(this.uber.width / 2 - this.width / 2),

// posição y da nave menos a altura do tiro e espaço extra

y: this.uber.y - this.height - 3,

});

// intervalo entre os disparos

this.recover = 10;

playSound(0);

}

},

[...]

},

Este método verifica se o *recover* é zero, se sim adiciona um novo disparo em *s* utilizando a posição atual da nave, ou seja, de *ship*, em seguida define *recover* com o valor 10 e toca o efeito sonoro de disparo.

### INIMIGOS

Para reunir todos os inimigos, de todos os tipos, em um só lugar existe um objeto chamado *enemy*, ele é muito simples:

MG.enemy = {

e: [],

// remover inimigos

clear: function () {

var enemy;

for (enemy in this.e) {

this.e[enemy].splice(0, this.e[enemy].length);

}

}

};

A propriedade *e* armazena todos os inimigos e o método *clear* apaga todos eles.

O primeiro tipo de inimigo é o meteoro, objeto *meteor*:

MG.meteor = {

[...]

m: [],

drop: [4, 1, 1, 0], // quantidade de itens dropados

interval: 60,

wait: 0,

score: 25,

types: 4,

};

Acerca das propriedades, *m* armazena todos os meteoros na tela, *interval* representa o intervalo de tempo de aparição dos meteoros, *wait* é um contador regressivo que quando é igual a zero um novo meteoro é criado, *score* é a pontuação mínima recebida pelo jogador quando este destrói um meteoro e *types* indica a quantidade de diferentes tipos meteoros disponível.

A propriedade *drop* indica a quantidade máxima de itens que este tipo de inimigo pode lançar ao ser destruído, a ordem indica o tipo do item, sendo, 50, 150 e 300 pontos respectivamente, o último indica a quantidade de multiplicadores a serem lançados.

O método *draw* segue o padrão, desenha o meteoro, porem o método *update* além de atualizar a posição dos meteoros e remove-los ao sair da tela, durante sua execução faz uma chamada a uma função externa, a qual verifica colisões entre cada um dos meteoros em *m*, a nave e seus disparos e caso haja algum chama uma terceira função para executar algumas tarefas, como, lançar itens, contabilizar pontos, remover o objeto das telas, etc.

Por fim, o método *create*, responsável por criar novos meteoros, que após gerar um tipo aleatório, verifica se *wait* é zero, caso seja, cria e adiciona um novo meteoro em *m*, então define a velocidade e a pontuação desse novo meteoro com base no tipo aleatório anteriormente gerado, em seguida atribui-se o valor de *interval* para *wait*.

MG.meteor = {

[...]

create: function () {

var type = randInt(this.types);

if (this.wait === 0) {

this.m.push({

height: this.height,

width: this.width,

// rotaciona os meteoros

rotate: 0,

type: type,

drop: this.drop,

// velocidade randomica

velocity: randInt(2) + 1 + type % 3,

score: this.score \* (type + 1),

// randomiza a posição horizontal inicial do meteor

x: randInt(this.screen.width - this.width),

y: -this.height

});

this.wait = this.interval;

}

},

[...]

};

O segundo tipo de inimigo é o *ufo* que assim como o meteoro possui as propriedades, *interval*, *wait*, *drop*, *score* e *types*, todos com o mesmo propósito, a diferença está apenas nas propriedades *drop* e *score* que possuem vários valores agora, a quantidade de valores ou de conjunto de valores deve ser igual a quantidade de *types*, mas o intuito destas ainda é mesmo que nos meteoros.

MG.ufo = {

[...]

velocity: 0.5,

interval: 150,

recover: 40,

wait: 0,

drop: [

[5, 3, 2, 1],

[6, 4, 3, 2],

[7, 5, 4, 3],

[3, 3, 5, 3]

],

score: [50, 150, 320, 470],

types: 4,

u: [],

};

Diferente dos meteoros o *ufo* possui uma propriedade *velocity*, que fixa a velocidade de movimentação de todos os *ufos*, e a propriedade *u* que armazena todos os *ufos* da tela.

Assim como a nave, um *ufo* pode disparar, portanto, possui uma propriedade *shoot*, que possui as mesmas propriedades e métodos que o objeto *shoot* da nave, a diferença está no fato de que o *ufo* possui um tipo de disparo diferente para cada categoria de *ufo*, ou seja, se *types* é quatro, logo, quatro tipos de disparo.

Os métodos *draw* e *update* do *ufo.shoot* funcionam objetivamente de forma parecida, porem com diferenças logicas, justamente por causa dos diferentes tipos de disparo do *ufo*, é aqui que também acontece a remoção de disparos que estão fora da tela e a verificação de colisão entre os disparos dos *ufos* e a nave do jogador.

O *create* faz praticamente o mesmo processo que o *ship.shoot.create*, porém não acontece nele a geração do tipo aleatório, mas sim na criação do *ufo*, ou seja, no método *ufo.create*.

O método *draw* do *ufo* é muito semelhante ao do meteoro, mas o método *update* é bem diferente, pois ele precisa remover aqueles *ufos* que sairão da tela de jogo e atualizar a posição de vários *ufos* de acordo com o seu tipo, e isso pode mudar completamente o modo como se move, este método é muito extenso, por isso abaixo é exemplificado apenas um trecho deste:

MG.ufo = {

[...]

update: function () {

switch (u.level) {

case 0:

case 1: // laser teleguiado

direction = 1;

break;

// ufo seguidor

case 2:

ucx = u.x + Math.floor(u.width / 2);

scx = MG.ship.x + Math.floor(MG.ship.width / 2);

u.y += u.velocity;

direction = 0;

if (u.y > MG.ship.y) {

break;

}

else if (scx < ucx) {

direction = -1;

}

else if (scx > ucx) {

direction = 1;

}

break;

case 3:

switch (u.status) {

case 'moving':

direction = 1;

if (u.wait === 0) {

u.status = 'stoped';

u.wait = 200;

}

break;

case 'stoped':

direction = 0;

if (u.wait === 0) {

u.status = 'moving';

u.wait = randInt(350) + 100;

}

break;

}

u.wait -= 1;

}

}

};

Este trecho é o responsável por identificar os diferentes tipos de *ufo* e atualizar sua posição respectivamente, ainda no método *update,* também é feita a chamada para a função de verificação de colisão entre o *ufo* e a nave do jogador.

Por último, o método *create*, responsável por aleatoriamente escolher o tipo de *ufo* e consequentemente o tipo de disparo, mas basicamente tem o mesmo comportamento e sequencia logica que o método *create* do meteoro.

### COLISÕES E EXPLOSÕES

As colisões durante o jogo são identificadas através de uma função independente, que após sua execução retorna um valor que indica se houve ou não uma colisão entre os objetos informados.

function hasCollision (array, target, should\_remove) {

var i = 0, a,

max = array.length;

for (; i < max; i += 1) {

a = array[i];

o = target;

if (a.width > o.width) {

a = target;

o = array[i];

}

if (a.y > o.y && a.y < o.y + o.height ||

a.y + a.height > o.y && a.y + a.height < o.y + o.height) {

if (a.x > o.x && a.x < o.x + o.width ||

a.x + a.width > o.x && a.x + a.width < o.x + o.width) {

if (should\_remove) {

array.splice(i, 1);

}

return true;

}

}

}

}

A função acima espera que, um conjunto de objetos, um alvo e um indicador booleano sejam informados. Cada item do conjunto será testado contra o alvo, ou seja, será verificado se existe uma intersecção entre as suas formas em algum ponto naquele instante, caso sim, significa que houve uma colisão e se o indicador booleano for verdadeiro significa que o item do conjunto que colidiu deve ser apagado.

Independentemente do resultado a função retorna sempre um valor booleano que indicando se houve ou não colisão.

O objeto *explosion* é muito simples, possui uma propriedade para armazenar todas as explosões ocorrendo na tela, *e*, um método *draw* para desenhar e um método *create* para criar uma explosão, que espera sempre as coordenadas de onde criar a explosão.

Já o método *update* atualiza o estado da explosão, criando o efeito visual de uma explosão, fazendo com que cresça até seu limite e depois diminua até desaparecer.

MG.explosion = {

[...]

e: [],

create: function (x, y) {

this.e.push({

height: this.height,

width: this.width,

// estado inicial da animação

status: 'exploding',

wait: 40,

rotate: 0,

x: x,

y: y

});

},

draw: function () {

var i = 0, e,

max = this.e.length,

half\_width, half\_height;

for (; i < max; i += 1) {

e = this.e[i];

half\_width = Math.floor(e.width / 2);

half\_height = Math.floor(e.height / 2);

this.ctx.save();

this.ctx.translate(e.x , e.y);

this.ctx.rotate(e.rotate);

this.ctx.drawImage(

this.spritesheet, 99, 24, 25, 24,

-half\_width, -half\_height, e.width, e.height

);

this.ctx.restore();

}

return this;

},

update: function () {

var i = 0,

max = this.e.length,

//redimensionando explosão

transforming = function (e, increase) {

e.height += increase;

e.width += increase;

};

for (; i < max; i += 1) {

e = this.e[i];

e.rotate += RAD \* 3;

switch (e.status) {

//explosão crescendo

case 'exploding':

if (e.height > 18) {

e.status = 'idle';

} else {

transforming(e, this.velocity);

}

break;

//explosão no *API*ce

case 'idle':

if (e.wait == 0) {

e.status = 'imploding';

} else {

e.wait -= 1;

}

break;

//explosão reduzindo

case 'imploding':

if (e.height < 2) {

this.e.splice(i--, 1);

max--;

} else {

transforming(e, -this.velocity);

}

break;

}

}

return this;

},

reset: function () {

this.e.splice(0, this.e.length);

}

};

Possui ainda um *reset* bem simples, que simplesmente apaga todas as explosões em *e*.

### ITENS

O objeto *item* é responsável por todos os itens lançados pelos inimigos, possui a propriedade *i* que armazena todos os itens que ainda não foram coletados ou não passaram pelos limites da tela.

MG.item = {

[...]

i: [],

};

Possui o método *reset* que apaga todos os itens em *i*. Os métodos *draw* e *update* são semelhantes aos demais já citados, desenham, atualizam, e fazem chamadas ao verificador de colisão, mas neste caso a colisão da nave com o item é vista de forma diferente, a colisão significa que a nave coletou o item. Ao coletar um item um método do próprio objeto é chamado, um respectivo ao tipo de item, caso seja de pontuação o método *score* é invocado:

MG.item = {

[...]

score: function () {

MG.player.score += this.score \* MG.player.mult;

},

[...]

};

Este método apenas multiplica o valor do item pelo multiplicador acumulado e soma a pontuação total. Existe o item multiplicador, invoca a função *mult*:

MG.item = {

[...]

mult: function () {

MG.player.mult += 1;

},

[...]

};

Simplesmente incrementa em um, o multiplicador do jogador. E o item do tipo vida, que incrementa a vida do jogador em um, método *live*:

MG.item = {

[...]

live: function () {

MG.player.credits += 1;

},

[...]

};

A combinação de dois métodos, *drop* e *launch*, executam o lançamento dos itens na tela, *drop* gera uma quantidade aleatória respeitando o limite máximo definido na propriedade *drop* do inimigo derrotado e *launch* cria os itens, um por vez, utilizando o método *create* do objeto.

MG.item = {

[...]

create: function (x, y, effect, type) {

this.i.push({

height: this.height,

width: this.width,

// indica o comportamento do item

effect: effect,

score: [50, 150, 300, 0][type],

status: 'dropping', //estado do item

type: type,

velocity: -1,

wait: 50,

x: x - Math.floor(this.width / 2),

y: y - Math.floor(this.height / 2)

});

},

launch: function (e, effect, type, i, max) {

var x = randInt(2) > 1 ? randInt(11) : (-randInt(7));

this.create(e.x + x, e.y, effect, type);

if (i < max) {

setTimeout(function () {

MG.item.launch(e, effect, type, i + 1, max);

}, 200);

}

},

drop: function (e, type, limit) {

var max = randInt(e.drop[type]);

this.launch(e, 'score', type, 0, max);

if (type < limit) {

setTimeout(function () {

MG.item.drop(e, type + 1, limit);

}, 200);

}

},

[...]

};

O método *create* assim como em outros objetos, cria um novo item e adiciona na propriedade *i* do objeto *item*.

## IMPLEMENTAÇÃO DAS TELAS

A implementação da interface em sua maioria é desenhar pedaços de um *spritesheet*, uma imagem constituída de outras pequenas imagens. Uma vez que o recurso está carregado, basta usar as funcionalidades do *Canvas* para desenhar pedaços selecionados da *spritesheet*.

Figura 4.7 – Spritesheet do jogo



Fonte: Autores deste trabalho

Existem dois objetos responsáveis pelas interfaces do jogo, *player* e *game*.

O objeto *player* possui dois métodos para desenhar a interface com as informações do jogador, o método *number*, que é um auxiliar, usado para desenhar qualquer número na tela, e *ui*, que desenha os raios, que representam o número de vidas restantes, e através de números, a pontuação e o multiplicador acumulado.

MG.player = {

[...]

numbers: function (value, y) {

var v = value.toString(),

i = 0, max = v.length;

for (; i < max; i += 1) {

this.ctx.drawImage(

this.spritesheet, (v[i] \* 22), 1, 20, 20,

this.screen.width - ((max - 1 - i) \* 13 + 20),

y, 13, 13

);

}

},

ui: function() {

var i = 0,

m = this.mult;

for (; i < this.credits; i += 1) {

this.ctx.drawImage(

this.spritesheet, 241, 0, 13, 21,

(i \* 12 + 8), 10, 9, 14

);

}

// pontuação

this.numbers(this.score, 10);

// multiplicador

this.numbers(m, this.screen.height - 22);

this.ctx.drawImage(

this.spritesheet, 221, 1, 17, 17,

this.screen.width - (m.toString().length \* 13 + 17),

(this.screen.height - 20.5), 10, 10

);

}

}

O objeto *game* é o responsável pela grande maioria das telas, e o código encarregado dessas telas está dividido da seguinte forma, a programação que desenha os objetos na tela fica dentro de outro objeto, *screen*, dentro de *game*:

MG.game = {

[...]

screen: {

continue: function () {

[...]

},

gameover: function () {

[...]

},

loading: function () {

[...]

},

paused: function () {

[...]

},

start: function () {

[...]

},

tutorial: function () {

[...]

},

[...]

},

}

Os métodos estão nomeados de acordo com a tela que têm por objetivo desenhar, o código interno dos métodos foi omitido para facilitar o entendimento da estrutura, uma vez que este também é muito simples.

A parte incumbida de executar a lógica por trás das telas, por exemplo, alterar entre as opções possíveis ou determinar quais opções foram selecionadas e qual o próximo passo, são métodos do objeto *game*:

MG.game = {

[...]

continue: function () {

[...]

},

gameover: function () {

[...]

},

start: function () {

[...]

},

pause: function () {

[...]

},

[...]

}

Percebe-se que nem todas as telas possuem um método para manipular lógica, isso porque elas não precisam, são apenas telas onde o jogador apenas observa e aguarda, como as telas de *loading* e tutorial.

O código interno dos métodos foi omitido porque em quase todos os casos o processo é basicamente o mesmo, verificando qual a opção selecionada e redirecionam para outra tela.

# RESULTADOS E DISCUSSÕES

Durante as etapas do desenvolvimento os resultados foram surgindo, como as fichas de requisitos, diversos diagramas, protótipos de tela e o código fonte, todos presentes nos seus respectivos apêndices. Mesmo assim, um ponto muito importante deve ser discutido, que na realidade era uma questão levantada no início deste trabalho, sobre as ferramentas para apoio nas diversas etapas do projeto.

O fato é, existem poucas ferramentas para acelerar e apoiar o processo de criação de jogos *web* que utilizam apenas as novas tecnologias nativas dos navegadores, então questionava-se a viabilidade dessas aplicações.

Contudo, mesmo com as poucas opções de ferramentas especializadas ainda existem as ferramentas convencionais, e a partir dessa outra questão surgiu, justamente acerca dessas ferramentas, até que ponto elas conseguem suprir as necessidades que estas aplicações podem ter?

Levando em conta essas questões a viabilidade da aplicação se tornou um pouco questionável, mas como anteriormente citado, os resultados apresentados provaram que é sim possível desenvolver este tipo de aplicação, mesmo sem ferramental especializado.

Com a grade variedade de jogos disponíveis na rede, para que uma aplicação se destaque é muito importante que ela tenha algo de diferente, e o diferencial desta aplicação em relação às demais são as tecnologias utilizadas no desenvolvimento, afinal foram utilizadas apenas as tecnologias nativas dos navegadores mais modernos, isso significa que mais pessoas terão acesso, ou seja, não serão impedidos de jogar por causa de alguma tecnologia terceira não disponível ou algo do tipo.

Por exemplo, os clássicos jogos disponíveis em redes sociais como o *Facebook*, sua grande maioria é desenvolvida utilizando Flash Player, um reprodutor de multimídia e aplicações amplamente distribuído, é instalado como um pacote extra nos navegadores, um *plugin*, sem esse *plugin* instalado o jogo não pode ser reproduzido, dessa forma prejudicando a acessibilidade da aplicação, afinal alguns dispositivos podem não ter suporte ao *plugin* ou simplesmente não permitir a instalação de *plugins*.

Já em relação a aplicações que também foram construídas com essas mesmas tecnologias *web*, o grande diferencial é o estilo do jogo, de rolagem vertical, ao estilo do épico *Space Invader* que fez muito sucesso no passado, e a qualidade dos recursos gráficos e sonoros, que hoje, sem dúvida são bem melhores. Outras diferenças que podem ser citadas, partidas pequenas e rápidas, nenhuma restrição etária e atualmente existem poucos jogos do mesmo estilo e com as mesmas tecnologias, uma pesquisa rápida apontou isso, apenas uma aplicação, Shootr[[7]](#footnote-7), que inclusive possui o mesmo tema que o jogo desenvolvido aqui, porem o estilo do jogo ainda é diferente.

# BIBLIOGRAFIA

ARRUDA, E. P. **Fundamentos para o Desenvolvimento de Jogos Digitais**. Porto Alegre: Bookman, 2014.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML. Guia do Usuário.** 2ª. ed. Rio de Janeiro: Elsevier, 2005.

*CANVAS* *API*. **MozillaDevelopers Network**. Disponivel em: <https://developer.mozilla.org/en-US/docs/*Web*/*API*/*Canvas*\_*API*>. Acesso em: 17 set. 2016.

CÁSSIO, É. **Desenvolva Jogos com *HTML*5 *Canvas* e *JavaScript***. [S.l.]: Casa do Código, 2014.

FILHO, W. D. P. P. **Engenharia de *Software* Fundamentos, Métodos e Padrões**. 3ª. ed. [S.l.]: LTC, 2009.

G1. Conheça os 10 games sociais mais populares do *Facebook*. **G1**, 2011. Disponivel em: <http://g1.globo.com/tecnologia/noticia/2011/09/conheca-os-10-games-sociais-mais-populares-do-*Facebook*.*HTML*>. Acesso em: 24 maio 2017.

G1. Por mês, mais de 250 milhões de usuários jogam no *Facebook*. **G1**, 2013. Disponivel em: <http://g1.globo.com/tecnologia/noticia/2013/03/por-mes-mais-de-250-milhoes-de-usuarios-jogam-no-*Facebook*.*HTML*>. Acesso em: 24 maio 2017.

G1 RIO. Brasil é o 4º consumidor de games, mas mercado carece de mão de obra. **G1**, 2015. Disponivel em: <http://g1.globo.com/rio-de-janeiro/noticia/2015/11/brasil-e-o-4-consumidor-de-games-mas-mercado-carece-de-mao-de-obra.*HTML*>. Acesso em: 19 fev. 2017.

G1 RIO. Internet chega pela 1ª vez a mais de 50% das casas no Brasil, mostra IBGE. **G1**, 2016. Disponivel em: <http://g1.globo.com/tecnologia/noticia/2016/04/internet-chega-pela-1-vez-mais-de-50-das-casas-no-brasil-mostra-ibge.*HTML*>. Acesso em: 19 fev. 2017.

GIL, A. C. **Como Elaborar Projetos de Pesquisa**. 4ª. ed. São Paulo: Atlas, 2002.

GOMES, H. S. Jogos simples são os mais baixados para celular no Brasil em 2014. **G1**, 2014. Disponivel em: <http://g1.globo.com/tecnologia/games/noticia/2014/12/jogos-simples-sao-os-mais-baixados-para-celular-no-brasil-em-2014-veja.*HTML*>. Acesso em: 02 mar. 2017.

GONZALES CLUA, E. ; BITTENCOURT, J. R. Desenvolvimento de Jogos 3D: Concepção, Design e Programação, São Leopoldo, jul. 2005.

HIRAMA, K. **Engenharia de *Software* Qualidade e Produtividade com Tecnologia**. Rio Janeiro: Elsevier, 2011.

LARMAN, C. **Utilizando UML e padrões. Uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo**. 3ª. ed. São Paulo: Bookman, 2007.

LEMES, O.; TOMASELLI, C.; , S. C. A economia digital e o mercado de jogos moveis. **SBC**, Brasilia, November 2012.

*MAC*HADO PINTO, N. ; CORONEL, D. A.; BRESOLIN, R. P. Análise Comparativa da Evolução das Vendas e do PIB. **Revista de Administração, Contabilidade e Sustentabilidade**, v. 3, n. Especial, p. 44-60, 15 set. 2013.

MDN. Introdução ao desenvolvimento de jogos para a *Web*. **Mozilla Developers Network (MDN)**, 2017. Disponivel em: <https://developer.mozilla.org/pt-BR/docs/Games/Introduction>. Acesso em: 25 fev. 2018.

MDN. Anatomia de um vídeo game. **Mozilla Developers Network**. Disponivel em: <https://developer.mozilla.org/pt-BR/docs/Games/Anatomy>. Acesso em: 17 set. 2016.

MDN. Audio for *Web* games. **Mozilla Developers Network**. Disponivel em: <https://developer.mozilla.org/en-US/docs/Games/Techniques/Audio\_for\_*Web*\_Games>. Acesso em: 22 abr. 2016.

MDN. Efficient animation for *web* games. **Mozilla Developers Network**. Disponivel em: <https://developer.mozilla.org/en-US/docs/Games/Techniques/Efficient\_animation\_for\_*web*\_games>. Acesso em: 22 abr. 2016.

MDN. Ferramentas para o desenvolvimento de jogos. **Mozilla Developers Network**. Disponivel em: <https://developer.mozilla.org/pt-BR/docs/Games/Tools>. Acesso em: 22 abr. 2016.

MDN. Herança e cadeia de protótipos (prototype chain). **Mozilla Develeopers Network**. Disponivel em: <https://developer.mozilla.org/pt-BR/docs/*Web*/*JavaScript*/Guide/Inheritance\_and\_the\_prototype\_chain>. Acesso em: 17 set. 2016.

MDN. *HTML*. **Mozilla Developers Network**. Disponivel em: <https://developer.mozilla.org/en-US/docs/*Web*/*HTML*>. Acesso em: 17 set. 2016.

MDN. *HTML*5. **Mozilla Developers Network**. Disponivel em: <https://developer.mozilla.org/en-US/docs/Glossary/*HTML*5>. Acesso em: 17 set. 2016.

MDN. Introdução. **Mozilla Developers Network**. Disponivel em: <https://developer.mozilla.org/pt-BR/docs/*Web*/*JavaScript*/Guide/Introduction>. Acesso em: 17 set. 2016.

MDN. Introdução ao *HTML*. **Mozilla Developers Network (MDN)**. Disponivel em: <https://developer.mozilla.org/pt-BR/docs/Aprender/*HTML*/Introducao\_ao\_*HTML*>. Acesso em: 17 set. 2016.

MDN. Optimizing startup performance. **Mozilla Developers Network**. Disponivel em: <https://developer.mozilla.org/en-US/Apps/Fundamentals/Performance/Optimizing\_startup\_performance>. Acesso em: 22 abr. 2016.

MDN. Técnicas para desenvolvimento de jogos. **Mozilla Developers Network**. Disponivel em: <https://developer.mozilla.org/pt-BR/docs/Games/Techniques>. Acesso em: 17 set. 2016.

MILETTO, E. M.; BERTAGNOLLI, S. D. C. **Desenvolvimento de *Software* II.Introdução ao Desenvolvimento *Web* com *HTML*, CSS, *JavaScript* e PHP**. Porto Alegre: Bookman, 2014.

NIERADKA, I. P. **Javascriot + CSS + DOM Desenvolvimento Para *Web***. [S.l.]: Novaterra, 2014.

OKUYAMA, F. Y.; MILETTO, E. M.; NICOLAO, M. **Desenvolvimento de *Software* I. Conceitos Basicos**. Porto Alegre: Bookman, 2014.

PRADANOV, C. C.; FREITAS, E. C. **Metodologia do Trabalho Científico:** Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico. 2ª. ed. Novo Hamburgo: Feevale, 2013.

PRESSMAN, R. R. **Engenharia de *Software*. Uma Abordagem Profissional**. 7ª. ed. São Paulo: McGraw Hill Brasil, 2009.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de *Software*. Uma Abordagem Profissional**. 8ª. ed. Porto Alegre: AMGH, 2016.

R.LECHETA, R. **Google Android - 3ª Edição:** Aprenda a criar aplicações para dispositivos móveis com o Android SDK. 3ª. ed. [S.l.]: Novatec, 2013.

REDAÇÃO OLHAR DIGITAL. Giphy atinge mais de 100 milhões de usuários diários. **Olhar Digital**, 2016. Disponivel em: <https://olhardigital.com.br/noticia/giphy-atinge-mais-de-100-milhoes-de-usuarios-diarios/63433>. Acesso em: 19 fev. 2017.

REDAÇÃO VEJA. Em 2020, haverá mais pessoas no mundo com celulares que com eletricidade, água e carros. **Veja**, 2016. Disponivel em: <https://veja.abril.com.br/economia/em-2020-havera-mais-pessoas-no-mundo-com-celulares-que-com-eletricidade-agua-e-carros/>. Acesso em: 01 mar. 2017.

REMOALDO, P. **O Guia Pratico do Dreamweaver CS3 com PHP, JqavaScript e Ajax**. 1ª. ed. [S.l.]: Centro Atlântico, 2008.

REZENDE, D. A. **Engenharia de *Software* e Sistemas da Informação**. 3ª. ed. Rio Janeiro: BRASPORT, 2005.

RONCOLATO, M. Games sociais; um mercado em alta. **Estadão**, 2011. Disponivel em: <https://link.estadao.com.br/noticias/geral,games-sociais-um-mercado-em-alta,10000039840>. Acesso em: 19 fev. 2017.

SANTOS, R. **Introdução à Programação Orientada a Objetos. Java**. 2ª. ed. Rio de Janeiro: Elsevier, 2013.

SILVA FILHO, A. M. **Introdução à Programação Orientada a Objetos com C++**. Rio de Janeiro: Elsevier, 2011.

SILVA, E. L.; MENEZES, E. M. **Metodologia da Pesquisa e Elaboração de Dissertação**. 4ª. ed. Florianópolis: [s.n.], 2005.

SILVEIRA FERREIRA, J. S. **A *WEB* SEMÂNTICA**. UNIVERSIDADE DO SUL DE SANTA CATARINA. Palhoça. 2010.

SOMMERVILLE, I. **Engenharia de *Software***. 6ª. ed. [S.l.]: Addison Wesley Longman Limited , 2003.

TRASVIÑA, F. Introdução ao *JavaScript* Orientado a Objeto. **Mozilla Developers Network (MDN)**, 2005. Disponivel em: <https://developer.mozilla.org/pt-BR/docs/*Web*/*JavaScript*/Introduction\_to\_Object-Oriented\_*JavaScript*>. Acesso em: 16 set. 2016.

WAZLAWICK, R. S. **Engenharia de *Software*. Conceitos e Práticas**. [S.l.]: Elsevier Brasil, 2013.

# APÊNDICE A

Neste anexo estão presentes as fichas de requisitos geradas durante o processo de levantamento de requisitos.

1. Disponível em <<http://gopherwoodstudios.com/sandtrap/sand-trap.htm>> Acesso em: 19/02/2017 [↑](#footnote-ref-1)
2. Disponível em <<http://www.playmapscube.com/>> Acesso em: 19/02/2017 [↑](#footnote-ref-2)
3. Disponível em <http://cocos2d-x.org> Acesso em: 22/04/2016 [↑](#footnote-ref-3)
4. Disponível em <http://piqnt.com/stage.js/> Acesso em: 22/04/2016 [↑](#footnote-ref-4)
5. Disponível em <http://www.isogenicengine.com/> Acesso em: 22/04/2016 [↑](#footnote-ref-5)
6. Disponível em <https://www.sublimetext.com/> Acesso em: 01/06/2018 [↑](#footnote-ref-6)
7. Disponível em <http://shootr.signalr.net/> Acesso em: 04/09/2017 [↑](#footnote-ref-7)