FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Jogos Digitais

Alison Fahl Nicolau

MODULARIZAÇÃO DE *GAME ENGINE*

Americana, SP

2016

FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Jogos Digitais

Alison Fahl Nicolau

MODULARIZAÇÃO DE GAME ENGINE

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Jogos Digitais, sob a orientação do Prof. Me. Kléber de Oliveira Andrade

Área de concentração: Engenharia de software.

Americana, SP.

2016

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

N546m NICOLAU, Alison Fahl
Modularização de game engine / Alison Fahl Nicolau. – Americana: 2016.
52f.

Monografia (Curso de Tecnologia em Jogos Digitais). - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza.
Orientador: Prof. Ms. Kleber de Oliveira Andrade

1. Jogos eletrônicos 2. Desenvolvimento de software I. ANDRADE, Kleber de Oliveira II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana.

CDU: 681.6

Alison Fahl Nicolau

MODULARIZAÇÃO DE GAME ENGINE

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Jogos Digitais pelo CEETEPS/Faculdade de Tecnologia – FATEC/ Americana.

Área de concentração: Engenharia de software

Americana, 08 de dezembro de 2016.

Banca Examinadora:

Kléber de Oliveira Andrade (Presidente)

Mestre

Faculdade de Tecnologia – FATEC/ Americana

Francesco Artur Perrotti (Membro)

Mestre

Faculdade de Tecnologia – FATEC/ Americana

Wagner Siqueira Cavalcante (Membro)

Mestre

Faculdade de Tecnologia – FATEC/ Americana

AGRADECIMENTOS

Em primeiro lugar agradeço aos meus familiares por me apoiarem durante o desenvolvimento deste trabalho.

Gostaria também de agradecer ao meu orientador por ser atencioso e estar sempre disposto a me ajudar em todos os detalhes do trabalho.

Por fim, agradeço aos amigos que também foram peça fundamental no meu encorajamento.

DEDICATÓRIA

Aos meus familiares, em especial meus pais e meu irmão, e amigos.

RESUMO

Este trabalho monográfico tem como principal objetivo analisar a viabilidade de um motor de criação de jogos digitais desenvolvido em uma arquitetura modular. Para levantar as conclusões, foram estudadas várias *game engines* já existentes no mercado, com intuito de recolher informações suficientes sobre as mais diversas possibilidades de implementação deste tipo de *software*. A fim de adquirir ainda mais conhecimento sobre o tema, também foram estudados os padrões de arquitetura que a maioria das *game engines* compartilham. De modo a testar a viabilidade da modularização, um protótipo de *game engine* modular foi desenvolvido ultizando a linguagem de programação JAVA, juntamente com o *framework* libGDX e a ferramenta de dependências Maven. Os resultados obtidos com o desenvolvimento do protótipo foram bem satisfatórios e demonstram o quão efetiva esta arquitetura é na criação de jogos digitais, dando espaço para a implementação de inúmeros módulos que podem ser compartilhados entre os usuários do sistema.

Palavras Chave: *Game Engine*; Modular; Jogos Digitais.

ABSTRACT

This dissertation has as the main goal analyse the viability of a game engine developed in a modular architecture. In order to rise any conclusion, it was made a study about a variety of existing game engines in the market, with the intention of collecting enough information about the most diverse range of possibilities in implementing this kind of software. In order to acquire even more knowledge about the theme, it was also studied the architecture patterns that the majority of the game engines share with each other. To test the viability of modularization, a modular game engine prototype was developed using the programming language JAVA, along with the libGDX framework and the Maven dependency tool. The results obtained with the prototype development were very satisfactory, showing the effectiveness of this type of architecture in the creation of computer games, giving up space to the implementation of innumerable modules that can be shared between the system's users.

Keywords: *Game Engine; Modular; Computer Games.*

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

Os motores de criação de jogos (do inglês, *game engine*) têm se mostrado ser de extrema importância na criação de jogos eletrônicos. Devido à grande variedade de jogos sendo produzidos para diferentes plataformas e públicos alvo, é possível encontrar jogos com as mais diversas mecânicas e características, o que acaba fazendo com que a quantidade de ferramentas disponíveis em cada *game engine* tenha se multiplicado de forma notória, para que possa então atender às diferentes exigências dos desenvolvedores. Muitos motores de criação de jogos têm optado por focar apenas em algumas plataformas e gêneros, para que assim possam especializar e aperfeiçoar suas ferramentas para o desenvolvimento das mecânicas mais utilizadas pelos desenvolvedores alvo.

Uma situação muito comum de se encontrar quando se utiliza uma *game engine* é a sobrecarga de ferramentas instaladas, das quais muitas acabam nem sequer sendo utilizadas, devido às diferentes exigências no *design* de cada jogo. Por outro lado, outro problema de comum ocorrência é a falta de suporte para a construção de uma certa mecânica, o que leva à prolongação no tempo de desenvolvimento do *game*, visto que tal ferramenta deverá ser adicionada pelos próprios desenvolvedores do jogo, caso possível ou necessário.

A criação de um motor para jogos que possui em seu conteúdo apenas o essencial para seu funcionamento, proporcionando a toda comunidade desenvolvedora uma maneira de criar, disponibilizar e adicionar qualquer tipo de ferramenta, poderia proporcionar um *software* leve e genérico, que possibilitaria a utilização por qualquer desenvolvedor, independente do objetivo do produto. Esse modelo de software, teria como um forte aliado, toda a comunidade de criadores de conteúdos que podem disponibilizar qualquer ferramenta produzida para o restante do público. A diversidade de opções ferramentas, acabaria beneficiando o usuário na utilização de apenas o que for julgado como o melhor e o necessário na criação de cada projeto específico.

O objetivo específico deste trabalho, consiste no desenvolvimento de um motor para criação de jogos baseado em uma arquitetura modular, que pode auxiliar no desenvolvimento de jogos eletrônicos dos mais diversos tipos, de forma simples e

genérica, diminuindo a sobrecarga de ferramentas e possibilitando aos desenvolvedores criar e compartilhar seus próprios módulos separadamente. Este modelo pode possibilitar um vasto leque de opções de ferramentas com semelhantes funcionalidades, deixando a critério do desenvolvedor qual dessas opções se enquadram melhor no projeto a ser desenvolvido.

Como forma de demonstrar a eficiência deste modelo de *game engine*, foi estudado e desenvolvido um protótipo com auxílio da linguagem de programação Java, utilizando o framework LibGDX para encapsulamento de funções e rotinas multi-plataformas, contendo mecanismos que em sua essência, dão suporte para a criação de inúmeras extensões. De modo a facilitar a união dos mais diferentes módulos, o protótipo de motor de criação de jogos conta com o suporte à dependência de projetos Maven 2. Após a criação de um sistema de gerenciamento de dependências, alguns módulos testes foram criados com o objetivo de colocar em prova o funcionamento real do sistema.

No segundo capítulo é apresentado um estudo sobre motores para criação de jogos. Nele será destacado como se concretizou o termo *game engine*, estudando as diferenças entre as tecnologias utilizadas mais comumente em cada gênero e também citando os modelos mais conhecidos e influentes.

No terceiro capítulo é apresentado um estudo sobre a arquitetura de um motor para criação de jogos. Nele são apresentadas as camadas do *software* juntamente com a descrição de cada funcionalidade existente.

No quarto capítulo é detalhado a metodologia utilizada para obtenção dos resultados da pesquisa. Nele estão descritas as ferramentas utilizadas e a arquitetura do *software* criado, além dos procedimentos efetuados para testar a aplicabilidade do protótipo.

O quinto capítulo contém os testes e resultados do desenvolvimento do protótipo.

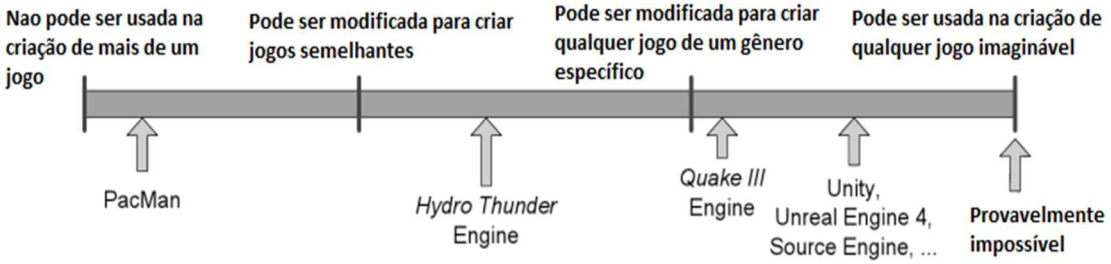
O sexto capítulo apresenta as considerações finais deste trabalho, juntamente com propostas futuras de pesquisa.

2 *GAME ENGINE*

De acordo com Gregory (2014), o termo “*game engine*" surgiu em meados dos anos 90 após o lançamento do popular jogo de tiro em primeira pessoa Doom[1], produzido pela Id *Software*[2]. O motivo dessa distinção se deve ao fato de que o *software* por trás do jogo tinha uma arquitetura muito bem definida em relação a separação de seus componentes, como por exemplo os sistemas de renderização[3] gráfica, detecção de colisão e o sistema de áudio, além do gerenciamento de *assets*[4], *level design* e a lógica do jogo. O autor ainda ressalta que essa separação foi essencial para que a popularidade dos motores de criação de jogos crescesse, pois, os desenvolvedores podiam reutilizar o *software* para a criação de conteúdos extras, novos jogos ou até mesmo vender a licença de uso para outras empresas desenvolvedoras.

Para Gregory (2014), pode-se classificar como "*game engine*" todo *software* de criação de jogos onde é possível utiliza-lo para a criação de diferentes jogos sem a necessidade de modificações expressivas. Quando a lógica de um jogo é feita embutida no código, ou as ferramentas utilizadas são específicas para o jogo em questão, o *software* usado no desenvolvimento acaba se tornando inutilizável para qualquer outro projeto que não seja o do jogo em questão. A Figura 1 contém uma escala imaginária dos níveis de classificação e reusabilidade de uma *game engine*.

Figura 1 – Escala de reutilização de uma *game engine*.



Fonte: Gregory (2014), adaptado pelo autor.

[1] Doom, http://doom.com/en-us/, último acesso em 16/11/2016
[2] Id *Software*, http://www.id*software*.com/, último acesso em 16/11/2016
[3] Renderização é um termo utilizado para representar um processamento digital, como o processo de criação de imagens digitais.
[4] Assets é nome dado a todo recurso de dados disponível no sistema, como arquivos de imagem, áudio ou texto.

2.1 Diferenças entre *game engine*s para diferentes gêneros de jogos

Diferentes jogos requerem diferentes tecnologias durante o seu desenvolvimento, porém muitos quesitos acabam sendo similares entre os mais diversos jogos. Gregory (2014) exemplifica essa semelhança em características como renderização de modelos 3D, animação de personagem, gerenciamento de *input* por teclado, mouse ou *joystick,* sistemas de *heads-up display*[5] (HUD), sistema de fonte e renderização de textos 2D e 3D, sistema de gerenciamento de áudio, além de vários algoritmos como os de busca de caminho (do inglês, *pathfinding*) (KHAN ACADEMY, 2016) e particionamento espacial (NYSTORM, 2014).

2.1.1 *First Person Shooter* (FPS)

Gregory (2014) afirma que, jogos de tiro em primeira pessoa (do inglês, *first person shooter* - FPS), primariamente eram caracterizados por serem ambientados em corredores em uma área fechada, e a movimentação do personagem era relativamente lenta e sem ajuda de veículos. Alguns dos exemplos mais conhecidos de jogos FPS são Quake[6], Unreal Tournment [7], Half-Life[8], Counter-Strike [9] (Figura 2) e Battlefield[10]. O autor também ressalta que muitos dos FPSs mais modernos já contam com cenários de mundo aberto, além de diferentes modos de navegação, como carros, aviões e barcos. Também é destacado que jogos FPS tem grande influência nas evoluções tecnológicas da indústria de jogos digitais. Isso se deve ao fato de que a maioria dos FPSs tendem a transmitir a ilusão do jogador estar imerso em um mundo super-realístico e detalhado, o que acaba exigindo muita pesquisa nas áreas audiovisuais, de animação de personagem, física, cinemáticas entre outras tecnologias para jogos digitais.

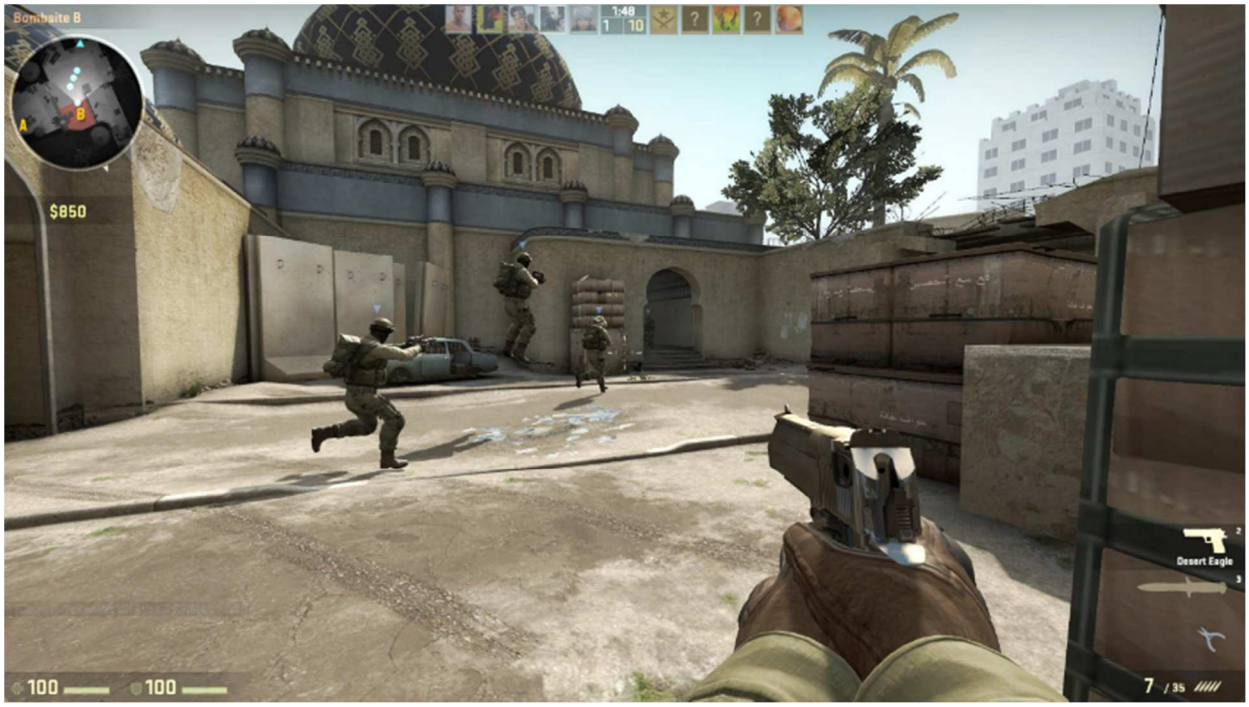
A ambientação de um FPS influencia na escolha da *game engine* utilizada no desenvolvimento do jogo. Gregory (2014) exemplifica essa variação, demonstrando o caso de jogos FPS em ambiente fechados comparados com jogos FPS em ambientes

---

[5] Head-Up Display é um termo em inglês usado para nomear todas as informações disponíveis na câmera do jogador.
[6] Quake, https://quake.bethesda.net, último acesso em 17/11/2016.
[7] Unreal Tournment, https://www.epic*game*s.com/unrealtournament/, último acesso em 17/11/2016.
[8] Half-Life 2, http://www.valve*software*.com/*game*s/hl2.html, último acesso em 17/11/2016.
[9] Counter-Strike Global Offensive, http://blog.counter-strike.net/, último acesso em 17/11/2016.
[10] Battlefield, https://www.ea.com/pt-br/*game*s/battlefield, último acesso em 17/11/2016.

abertos. No caso de ambiente fechado, costuma-se aplicar a técnica de *Binary Space Partitioning Tree* (BPS *Trees*) (NAYLOR, 2016) ou sistemas de renderização baseados em portais. Já os jogos em ambiente aberto, é comum utilizar-se a oclusão de objetos não visíveis na câmera do personagem.

Figura 2 - Counter-Strike Global Ofensive.



Fonte: http://www.airborne*gamer*.com/review/counter-strike-global-offensive-csgo-pc-review/ (2016)

2.1.2 Jogos de plataforma e em terceira pessoa

Gregory (2014) diz que o termo jogos de plataforma é aplicado a jogos de ação em que o controle do personagem é em terceira pessoa e a mecânica principal envolve mover-se de uma plataforma para outra. Alguns exemplos clássicos de jogos plataforma em 2D são Donkey Kong[11], Pitfall[12] e Super Mario[13]. Já a era 3D conta com jogos como Super Mario 64, Crash Bandocoot, Rayman[14] 2, Sonic the Hedgehog[15] e Super Mário Galaxy. No caso da tecnologia mais comumente utilizada nesses jogos, em geral é a mesma requisitada em jogos de aventura e ação em terceira pessoa e

[11] Donkey Kong, https://donkeykong.nintendo.com, último acesso em 17/11/2016.
[12] Pitfall, http://www.atari2600.com.br/Atari/Roms/01Gn/Pitfall, último acesso em 17/11/2016.
[13] Mario *Game*s, https://mario.nintendo.com/, último acesso em 17/11/2016.
[14] Rayman Legends, https://www.ubisoft.com/pt-BR/*game*/rayman-legends/, último acesso em 17/11/2016.
[15] Sonic the Hedgehog, http://www.sonicthehedgehog.com/en/, último acesso em 17/11/2016.

jogos de tiro em terceira pessoa, como Dead Space[16] 2, Gears of War[17] 3, Red Dead Redemption[18], as séries de Uncharted[19], as séries de Resident Evil[20] e The Last of Us[21] (Figura 3).

Figura 3 – The Last of Us.

Fonte: http://www.tudogeek.com.br/2015/01/09/review-the-last-of-us-remastered/ (2016)

Quando comparado com jogos de tiro em primeira pessoa, Gregory (2014) mostra que as semelhanças são várias, porém com maior ênfase na mecânica de movimentação do personagem principal. Com relação a animação e detalhes gráficos do personagem principal, os jogos em terceira pessoa também costumam ser mais bem trabalhados, visto que não são todos os jogos em primeira pessoa que expõem o corpo do personagem principal (em muitos casos somente são visíveis os membros superiores e a arma).

[16] Dead Space, https://www.ea.com/pt-br/*game*s/deadspace, último acesso em 17/11/2016.
[17] Gears of War, https://gearsofwar.com/pt-br, ultimo acesso em 17/11/2016.
[18] Red Dead Redemption, http://www.rockstar*game*s.com/reddeadredemption, ultimo acesso em 17/11/2016.
[19] Uncharted, www.unchartedthe*game*.com/pt-br/, último acesso em 17/11/2016.
[20] Resident Evil, http://residentevil.com.br/, último acesso em 17/11/2016.
[21] The Last of Us, http://www.thelastofus.playstation.com/, ultimo acesso em 17/11/2016.

2.1.3 Jogos de luta

Jogos de luta normalmente envolve a disputa entre dois ou mais personagens dentro de uma arena de combate de curta distância. Os jogadores controlam personagens de poder comparável e precisam dominar técnicas de sequência de ataque (também conhecida como combos) e defesa (GREGORY, 2014). Alguns exemplos de jogos de luta são: Soul Calibur[22], Tekken[23], Mortal Kombat[24], Street Fighter[25] e EA Sports UFC [26] (Figura 4).

Figura 4 – UFC 2014.

Fonte: http://themania*game*r.blogspot.com.br/2014/02/ea-sports-ufc-*game*play-vs-realismo.html (2016).

Com relação ao uso da tecnologia, Gregory (2014) afirma que jogos de luta, por possuir cenários relativamente pequenos onde a câmera é centralizada, são raros os casos em que se é utilizada oclusão de objetos (processo pelo qual é filtrado quais objetos são visíveis ao jogador) ou divisão do espaço (sistema responsável por dividir a cena do jogo em áreas específicas). Outro aspecto destacado, é a propagação de áudio que não costuma ser baseada em física 3D. Quando se trata de animação e

[22] Soul Calibur, http://www.soulcalibur.com/, último acesso em 17/11/2016.
[23] Tekken, http://tk7.tekken.com/, último acesso em 17/11/2016.
[24] Mortal Kombat, https://www.mortalkombat.com/, último aceso em 17/11/2016.
[25] Street Fighter, http://www.streetfighter.com.br/, último acesso em 17/11/2016.
[26] EA Sports UFC, https://www.easports.com/ufc, último acesso em 17/11/2016.

detalhamento gráfico de personagens, os jogos de luta costumam ser estado da arte, com *shaders*[27] de pele realísticos (WILSON, 2016), efeitos de suor e machucados também muito convincentes, assim como mostrado na Figura 4 com o jogo UFC 2014 da EA Sports.

2.1.4 Jogos de corrida

Jogos de corrida são caracterizados por possuírem um veículo controlado por um jogador onde o mesmo tem o objetivo de vencer uma corrida. Tanto o veículo quanto a competição podem ser baseadas em elementos reais ou fictícios (GREGORY, 2014). Alguns exemplos de jogos de corrida são: Gran Turismo[28], Need for Speed[29], Forza Horizon[30] (Figura 5), The Crew[31] e Super Mario Kart.

Figura 5 – Forza Horizon 2.

Fonte: http://www.eggplante.com/2014/12/18/forza-horizon-2-review/ (2016).

Sobre jogos de corrida, Gregory (2014) afirma que não somente os jogos que simulam a realidade estão nessa categoria. Jogos como Mario Kart por exemplo, oferecem uma versão cartunesca em que o jogador pode coletar bônus mágicos

[27] Shader é o nome dado para um programa de criação cores.
[28] Gran Turismo, http://www.gran-turismo.com/br/, último acesso em 17/11/2016.
[29] Need for Speed, https://www.needforspeed.com/pt_BR, último acesso em 17/11/2016.
[30] Forza Horizon, http://forzamotorsport.net/en-us/*games*/FH, último acesso em 17/11/2016.
[31] The Crew, https://www.ubisoft.com/pt-BR/*game*/the-crew/, último acesso em 17/11/2016.

durante a corrida, além de usar poderes especiais para atrapalhar os adversários. Em alguns casos também há modos de jogo em que não há uma pista linear e o vencedor é tido como o último que permanecer vivo dentro da pista ou que obtiver maior pontuação ao final do tempo.

Com relação à tecnologia, o autor diz que é muito dependente do estilo do jogo. Jogos em que o personagem é exposto, como no caso dos karts, é preciso criar animação para o personagem. Contudo, em geral, os jogos de corrida são conhecidos por usarem truques de renderização de objetos distantes, como representações 2D de árvores, morros e montanhas. Na grande maioria dos casos a câmera sempre segue o carro do jogador em terceira pessoa, contando com a opção, em alguns casos, de mudá-la para dentro do *cockpit* em um estilo primeira pessoa. Por se tratar de cenários com longos corredores e caminhos alternativos, algoritmos de *pathfinding* (KHAN ACADEMY, 2016) são amplamente utilizados.

2.1.5 Jogos de estratégia em tempo real

Jogos de estratégia em tempo real (do inglês, *real time strategy* - RTS) descendem do gênero de estratégia com a diferença de que o jogo progride em tempo real ao invés de turnos, em que cada jogador controla suas unidades simultaneamente (GREGORY, 2014). Alguns exemplos de jogos conhecidos nesse gênero são: Warcraft[32], Age of Empires[33] (Figura 6), Command and Conquer[34] e Starcraft[35].

De acordo com Gregory (2014), os jogos de estratégia em tempo real mais antigos possuíam um sistema de divisão de espaço baseado em células, além de contar com uma projeção ortográfica na renderização do cenário, similar ao jogo Age of Empires 2, representado na Figura 6. Já os jogos mais recentes, costumam utilizar projeção em perspectiva e cenários verdadeiramente 3D, porém muitos desses jogos ainda contam com o sistema de células para facilitar a disposição de construções e outras unidades dentro do espaço do jogo. O autor também destaca outras características de jogos desse gênero, como o uso comum de modelos de baixa resolução, para facilitar a renderização de várias unidades simultaneamente, caso

[32] Warcraft: Orcs and humans, http://us.blizzard.com/pt-br/*games*/legacy/, último acesso em 17/11/2016.
[33] Age of Empires, https://www.ageofempires.com/, último acesso em 17/11/2016.
[34] Command and Conquer, http://www.commandandconquer.com/, último acesso em 17/11/2016.
[35] Starcraft, http://us.blizzard.com/pt-br/*games*/sc/, último aceso em 17/11/2016.

muito comum em jogos RTS, em que cada jogador constrói gigantescos exércitos. Outras características são a utilização de *height-maps* (DONNERSTAG, 2014) na construção dos cenários, e também a interface de usuário, que é sempre muito bem populada, com várias barras de ferramentas, comandos e informações de recursos e equipamentos.

Figura 6 – Age of Empire 2.

Fonte: https://www.ageofempires.com/*games*/aoeii/ (2016).

2.1.6 Jogos *Massively Multiplayer Online* (MMO)

Os jogos que se encaixam nessa categoria são caracterizados por possuir suporte para uma larga quantidade de jogadores jogando simultaneamente na mesma instância do jogo. Em geral, cada instância é persistente e de mundo aberto (GREGORY, 2014). Alguns exemplos de MMO são: Guild Wars[36] 2, EverQuest[37], World of Warcraft[38] e Runescape[39] (Figura 7).

Para Gregory (2014), MMOs são caracterizados por grandes porções de servidores responsáveis por manter um estado de autoridade sobre tudo o que acontece nas instâncias do jogo que executam. Esses servidores tem o trabalho de

[36] Guild Wars, https://www.guildwars.com/en/, último acesso em 17/11/2016.
[37] Everquest, https://www.everquest.com/home, último acesso em 17/11/2016.
[38] World of Warcraft, https://worldofwarcraft.com/pt-br/, último acesso em 17/11/2016.
[39] Runescape, https://www.runescape.com/, último acesso em 17/11/2016.

administrar toda conexão de usuários, processando todos os *inputs* e informando o cliente sobre o estado atual do jogo, conforme a localização do jogador, além de permitir comunicação entre jogadores, via chat ou até mesmo via áudio, como no caso das comunicações *voice-over-IP* (*VoIP*). O autor também afirma que a grande maioria dos MMOs possui como fonte primária de renda, a mensalidade paga por jogadores para ter acesso aos servidores, além de permitir micro transações entre benefícios do jogo e dinheiro real.

Figura 7 – Runescape Clan War.

Fonte: **https://www.sythe.org/threads/pvp-planet-rsps-clan-wars-unique-spawning-system-amazing-switcing/ (2016).**

Na questão gráfica, afirma-se que MMOs não costumam ser muito detalhados e em média são inferiores a outros jogos. Isto se deve ao fato de que o mundo dentro do jogo costuma ser muito maior, além da possibilidade de que vários jogadores interajam com o cenário simultaneamente, assim como a Figura 7, que demonstra um confronto entre dois clãs de jogadores no jogo Runescape.

2.1.7 Jogos de criação de conteúdo

Com a ascensão das mídias sociais, os jogos estão seguindo a tendência de se tornar mais colaborativos por natureza. Quem afirma isso é Gregory (2014), que

traz exemplos de jogos recentes, como Little Big Planet[40], um jogo de plataforma com quebra-cabeças, mas que também traz a opção para o jogador criar seus próprios cenários e disponibilizá-los publicamente para que outras pessoas possam experimentar.

Outro exemplo citado por Gregory (2014) é o popular Minecraft[41] (Figura 8). Para ele, o segredo do sucesso de Minecraft está na simplicidade, pois os elementos do jogo são simples voxeis (FOLEY ET AT. 1990) mapeados e texturizados com texturas de baixa resolução para representar objetos do jogo. Com isso, os jogadores podem gerar mundos completamente randômicos e explorá-los, tanto na superfície quando no subterrâneo, coletando recursos que podem ser utilizados para criar suas próprias estruturas e modificando o cenário do jogo como bem desejar. Os jogadores podem também compartilhar o mundo em que jogam com outros jogadores através da criação de servidores.

Figura 8 – Minecraft.

Fonte: http://articles.*game*rheadquarters.com/review204minecraftwindows10.html (2016).

[40] Little Big Planet, http://littlebigplanet.playstation.com/, último acesso em 17/11/2016.
[41] Minecraft, https://minecraft.net/pt-br/, último acesso em 17/11/2016.

2.2 Pesquisa sobre *game engines*

A fim de entender quais são os modelos de *game engines* mais usadas no mercado, foi feita uma pesquisa sobre alguns desses *software*s, dos mais antigos até os atuais, estudando a evolução e a diferença entre os mesmos.

2.2.1 Quake Engine

De acordo com Gregory (2014), o primeiro jogo do gênero FPS a ser criado foi o Castle of Wolfenstein 3D em 1992, desenvolvido pela empresa ID *Software*. Para Gregory (2014), o jogo foi responsável por mudar o rumo da indústria dos jogos eletrônicos. Com a criação das subsequentes franquias, Doom e Quake, todas desenvolvidas em uma arquitetura muito familiar, ID Sofware revolucionou o cenário de criação de jogos, dando origem a uma tecnologia reutilizável, que inclusive foi usada por outras empresas para criação de diferentes jogos e inclusive outras *game engines,* como no caso do jogo Medal of Honor[42], que foi criado a partir de uma sub *engine* de Quake 3.

2.2.2 Unreal Engine

A Unreal *Engine*, criada pela empresa Epic *Game*s[43] Inc., apareceu no cenário em 1998, como um competidor à então Quake Engine para jogos FPS. Desde sua criação, a Unreal *Engine* foi muito bem aceita no mercado, sendo utilizada tanto por empresas para desenvolver jogos comerciais, quando em universidades para aprendizado. O Sucesso da Unreal está na sua rica opção de ferramentas disponíveis dentro do *software*, ao lado de uma interface gráfica muito intuitiva (GREGORY, 2014). Atualmente a Unreal Engine encontra-se na quarta versão, com jogos como Gears of War sendo produzidos nela.

2.2.3 Source Engine

Gregory (2014) descreve brevemente a Source Engine, da empresa Valve[44], como sendo muito semelhante à Unreal Engine 4, principalmente por ter muitas

[42] Medal of Honor, http://www2.ea.com/pt/medal-of-honor, último acesso em 17/11/2016.
[43] Epic *Game*s, https://www.epic*game*s.com/pt-BR, último acesso em 17/11/2016.
[44] Valve, http://www.valve*software*.com/, último acesso em 17/11/2016.

ferramentas e uma boa interface de usuário. Alguns dos títulos criados a partir desta ferramenta são: Half-Life 2, Team Fortress[45] 2 e Portal[46].

2.2.4 Frostbite

Frostbite é uma *engine* desenvolvida dentro dos estúdios da empresa DICE[47], para o então jogo Battlefield Bad Company, em 2006 (GREGORY, 2014). Desde então, a *engine* se tornou uma das mais utilizadas e adaptadas em diferentes franquias de jogos da Eletronic Arts[48] (EA), incluindo Mass Effect[49], Battlefield, Need For Speed e Dragon Age[50]. Este motor conta com uma forte ferramenta de criação de *assets* embutida, chamada de FrostEd, além de possuir uma série de ferramentas chamadas Backend Services e ser uma excelente *runtime engine*. No momento em que este trabalho foi escrito, a Frostbite estava em sua terceira versão, com suporte para exportar jogos para Windows[51] (PC), Xbox[52] 360, Xbox One, Playstation[53] 3 e Playstation 4.

2.2.5 CryEngine

A empresa Crytek[54] primeiramente desenvolveu sua *engine,* conhecida como CryEngine, para uma demonstração das tecnologias fornecidas pela Nvidia[55]. Quando o grande potencial da *game engine* foi finalmente reconhecido, a Crytek transformou a demo em jogo completo, chamado Far Cry[56]. Uma das mais notáveis qualidades desta tecnologia é a qualidade gráfica (GREGORY, 2014). Atualmente, a versão mais recente é Cry*Engine* 3, com suporte para exportar jogos para PC, Xbox 360, Xbox One, Playstation 3, Playstation 4 e Nintendo[57] Wii U.

---

[45] Team Fortress, http://www.teamfortress.com/, último acesso em 17/11/2016.
[46] Portal, http://www.valve*software*.com/*game*s/portal.html, último acesso em 17/11/2016.
[47] DICE, http://www.dice.se/, último acesso em 17/11/2016.
[48] Eletronic Arts, https://www.ea.com/pt-br, último acesso em 17/11/2016.
[49] Mass Effect, http://masseffect.bioware.com/, último acesso em 17/11/2016.
[50] Dragon Age, https://www.dragonage.com/pt_BR/home, último acesso em 17/11/2016.
[51] Windows, https://www.microsoft.com/pt-br/windows/, último acesso em 17/11/2016.
[52] Xbox, http://www.xbox.com/pt-BR/, último acesso em 17/11/2016.
[53] Playstation, https://www.playstation.com/pt-br/, último acesso em 17/11/2016.
[54] Crytek, http://www.crytek.com/, último acesso em 17/11/2016.
[55] Nvidio, http://www.nvidia.com.br/page/home.html, último acesso em 17/11/2016.
[56] Far Cry, http://www.crytek.com/*game*s/far-cry/overview, último acesso em 17/11/2016.
[57] Nintendo, http://www.nintendo.com/, último acesso em 17/11/2016.

2.2.6 PhyreEngine

Como uma forma de tornar o desenvolvimento de jogos para o Playstation 3 mais acessível, a Sony[58] introduziu em 2008 a Phyre*Engine* (GREGORY, 2014). Desde sua criação, esta *engine* evoluiu para um *software* com várias ferramentas e tecnologias embutidas. Alguns dos estúdios que se utilizaram desta *engine* foram a that*game*company[59] com os títulos flOw, Flower e o sucesso Journey, e também o estúdio From *Software*[60] com os *game*s Demon's Souls e Dark Souls (GREGORY, 2014). As plataformas suportadas pela PhyreEngine são: Playstation, Playstation 2, Playstation 3, Playstation 4, Playstation Vita e o portátil da Sony PSP.

2.2.7 MonoGame

MonoGame é uma implementação de código aberto do Microsoft XNA *framework,* que foi criado com o objetivo de permitir os desenvolvedores do XNA exportar seus jogos para diferentes plataformas entre dispositivos móveis, consoles, portáteis e computadores pessoais. O projeto começou com o simples intuito de criar uma extinção que fosse capaz de exportar jogos 2D para dispositivos móveis. O nome inicial do projeto era XNA Touch (MONO*GAME*, 2016).

O *framework* XNA, criado pela Microsoft, tem como forte característica o encorajamento de compartilhar os jogos criados nela. Essa filosofia foi criada pela Microsoft com o intuito de povoar a Xbox Live com diversos jogos criados por estúdios independentes (GREGORY, 2014). O XNA é baseado na linguagem C# (MICROSOFT, 2016) e funciona em conjunto com o ambiente de desenvolvimento integrado (do inglês Integrated Development Environment - IDE) Visual Studio[61], que por sua vez é responsável por manipular todas ferramentas, *assets* e código.

2.2.8 Unity 3D

A Unity 3D tem como principal característica a grande variedade de plataformas suportados para exportação de jogos. Entre essas plataformas há consoles, PC, dispositivos móveis e navegadores de internet. Uma outra característica muito

[58] Sony, http://www.sony.com.br/, último acesso em 17/11/2016.
[59] That*game*company, http://that*game*company.com/, último acesso em 17/11/2016.
[60] From *Software*, http://www.from*software*.jp/pc_en/, último acesso em 17/11/2016.
[61] Visual Studio, https://www.visualstudio.com/, último acesso em 17/11/2016.

importante da Unity, é a presença de um editor integrado e muito simples de se usar. Com esse editor, o usuário pode criar cenas muito rapidamente, manipular *assets* e acessar ferramentas de maneira simples e intuitiva. Além de possuir suporte para diversas exportação em diversas plataformas, é possível também rodar o jogo em modo de teste em cada plataforma além de contar com ferramentas que ajudam na análise de desempenho em cada dispositivo alvo. A Unity também conta com a possibilidade de o código ser escrito em mais de uma linguagem de programação, que sã: C#, Javascript (MOZILA, 2016) e Boo (BOO, 2016).

2.2.9 OGRE

A Ogre *engine* é um motor *open source* e gratuito de fácil uso e aprendizado. Nela é possível encontrar sistemas avançados de renderização 3D com efeitos de iluminação e sombra, além de contar com um ótimo sistema de animação de esqueleto, um sistema 2D para HUDs e a opção de aplicar efeitos *post-processing* (GREGORY, ,2014).

2.2.10 Comparação entre as *Game Engines*

Na Tabela 1 são comparadas algumas características fundamentais entre as *game engines* estudadas.

Tabela 1 – Comparação de *game engines*.

Engine	**Plataformas**	**Multi-gênero**	**Gratuita**	**Código aberto**	**Origem**
Quake	Windows PC	Nao	Sim		Jogo
Unreal	Windows PC, Playstation 4, Xbox One, Mac OS X, iOS, Android, VR*	Sim	Parcialmente*	Sim	Jogo
Source	Windows PC, Xbox360	Sim	Parcialmente*	Não	*Engine*
Frostbite	Windows PC, Xbox One, Playstation 4	Sim	Não	Não	*Engine*
Cry*Engine*	Windows PC, Xbox One, Playstation 4, Wii U	Sim	Não	Não	*Engine*
Phyre*Engine*	Playstation 4, Playstation Vita, Playstation PSP	Sim	Sim	Não	*Engine*
Mono*Game*	iOS, Android, MacOS, Linux, Windows Phone, Windows PC, Xbox One, OUYA, Playstation 4, Playstation Vita	Sim	Sim	Sim	*Engine*
Unity	iOS, Android, Windows Phone, PC, MacOS, Linux, HTML 5, Playstation 4, Playstation Vita, Xbox One, Wii U, Nintendo 3DS, VR,	Sim	Sim	Somente em alguns planos	*Engine*
OGRE	Windows PC, Linux, MacOS, Android, iOS, Windows Phone	Sim	Sim	Sim	*Engine*

Fonte: Próprio autor.

*VR = Realidade Virtual (do inglês Virtual Reality).

*Parcialmente é quando o uso é gratuito porem para exportar jogos comerciais é necessário dividir os royalties ou pagar uma taxa subscrição.

3. ARQUITETURA DE EXECUÇÃO DE UMA *GAME ENGINE*

De acordo com Gregory (2014), uma *game engine* geralmente consiste de um conjunto de ferramentas juntamente com um componente de execução. Neste trabalho será estudado apenas o que compõe a execução da *game engine*. A Figura 9 contém um diagrama com os componentes mais conhecidos de uma *game engine,* ordenado por dependência, sendo os componentes presentes no topo do diagrama dependentes dos componentes inferiores do mesmo. Segundo Gregory (2014), essa divisão por camadas é muito comum em qualquer *software* de grande porte, como no caso de *game engines*. Em uma arquitetura em camadas, os componentes presentes nos níveis mais baixos das camadas não costumam depender de componentes de nível superior, pois isso acaba causando dependência circular entre os componentes, que por sua vez tende a dificultar a reutilização, modificação e expansão do sistema.

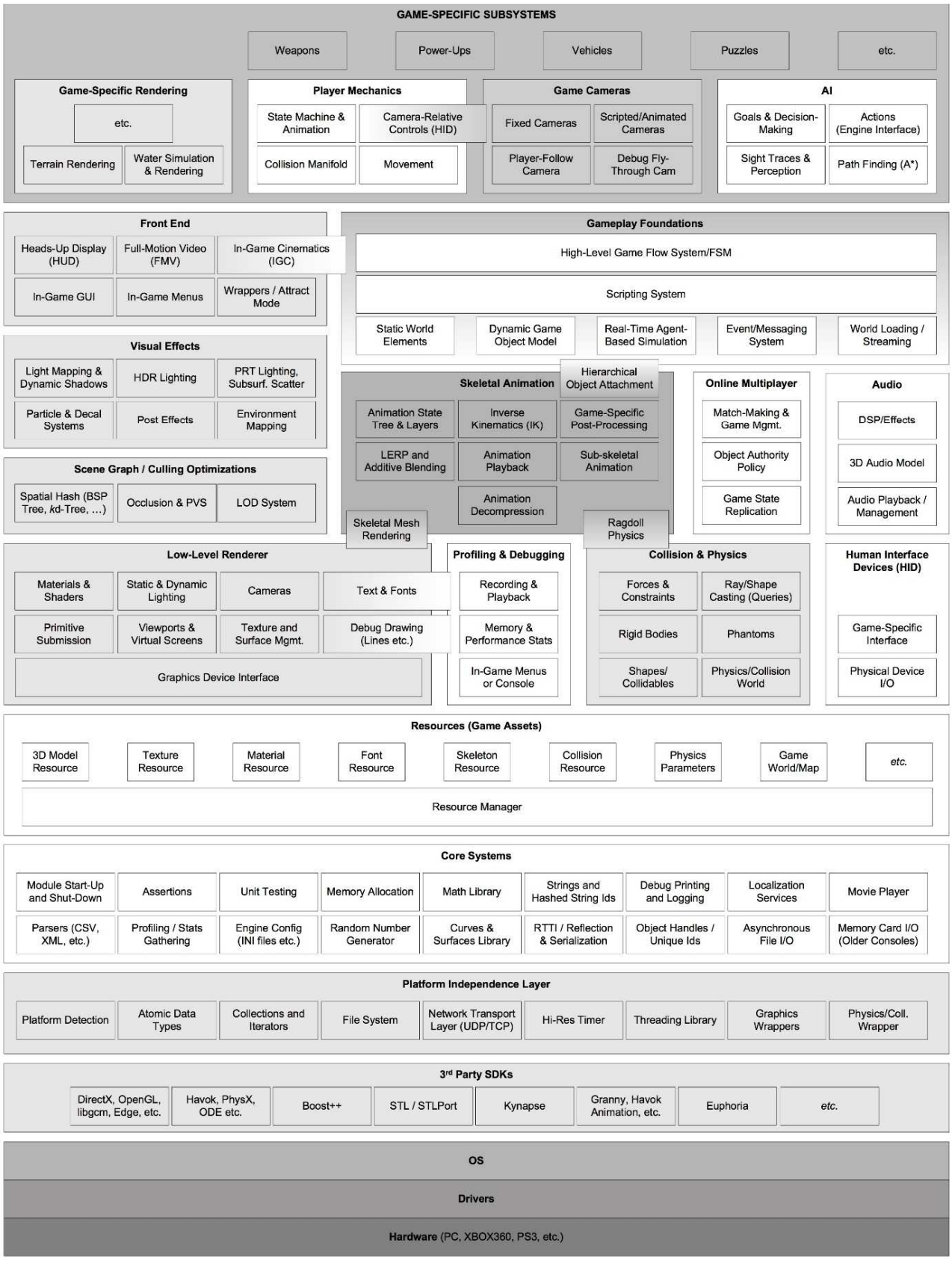
3.1 *Hardware*

O *hardware*, localizado na camada mais inferior da Figura 9, representa o sistema computacional em que o jogo será executado (GREGORY, 2014). Existem vários tipos de dispositivos diferentes no mercado com capacidade de executar um jogo. Alguns exemplos são: PCs, consoles e dispositivos móveis. Cada um dos diferentes dispositivos possui *hardwares* diferenciados e podem ser executados em com diferentes sistemas operacionais.

3.2 *Driver*

Os *drivers* são programas responsáveis por controlar dispositivos. Todo dispositivo precisa de um *driver* que possa se comunicar com o sistema operacional, servindo como um tradutor entre o dispositivo e o sistema operacional, pois cada dispositivo interpreta comandos de maneira diferente. Isso significa que o *driver* fica responsável por interpretar um comando enviado do sistema operacional para o dispositivo ou vice e versa (BEAL, 2016).

Figura 9 – Arquitetura de execução de uma *game engine*.



Fonte: Gregory, 2014

3.3 Sistema Operacional

Existem diferentes tipos de sistemas operacionais (SO), cada um desenvolvido para um tipo de plataforma. No caso de sistemas operacionais para PCs, o acesso aos *hardwares* é compartilhado entre as várias aplicações sendo executadas simultaneamente, incluindo os jogos. Isso significa que um programa nunca terá controle total sobre um *hardware*. Já no caso de um console, o jogo tem controle total sobre os *hardwares* disponíveis no aparelho, com exceção das versões mais novas de consoles, como no caso do Playstation 3 e sucessores e também do Xbox 360 e sucessores, onde o sistema operacional consegue interromper a execução do jogo para abrir um menu de opções de comandos. Essa diferença no modo como os SOs são aplicados nos consoles demonstra que a diferença entre o SO de um PC e o de console está cada vez menor (GREGORY, 2014).

3.4 SDKs terceirizados e *Middlewares*

Muitas das *game engines* trazem consigo uma dependência em kits de desenvolvimento de *software* (do inglês *software development kit* - SDK) de terceiros e também em *middlewares* (SCHANTZ; SCHMIDT, 2016). A interface fornecida por essas bibliotecas baseadas em classes ou funcional são comumente chamadas de interface de programação de aplicações (do inglês *application programming interface* - API) (GREGORY, 2014). Alguns exemplos de SDKs geralmente encontradas em *game engines* são:

3.4.1 API Gráfica

A grande maioria dos motores de renderização é construído em cima de uma interface fornecida por um *hardware* gráfico. Algumas dessas interfaces existentes são: *Open*GL (KESSENICH ET AL., 2016) e DirectX (LUNA, 2016). *Open*GL é uma API gráfica, disponível em diferentes plataformas, capaz de fornecer funções de renderização, mapeamento de texturas e efeitos de visualização com qualidade e velocidade. Diferente do *Open*GL, DirectX não conta apenas com uma API gráfica, ele é uma coleção de APIs de multimídia, com funções gráficas, de áudio e *inputs*, que opera em plataformas da Microsoft.

3.4.2 Estrutura de dados e algoritmos

Assim como qualquer outro *software*, jogos eletrônicos dependem muito de estruturas de dados e algoritmos (GREGORY, 2014). Um exemplo muito utilizado são as bibliotecas padrão de cada linguagem, como no caso da *Standard Template Library* (STL) em C++ (CPLUSPLUS, 2016). Segundo Gregory (2014), muitos desenvolvedores e programadores de jogos dividem opinião quanto ao uso de bibliotecas de estrutura de dados e algoritmos. Para os que discordam, a razão é que a forma com que a memória e processamento é manipulada pode não ser a mais otimizada para todas as plataformas.

3.4.3 Colisão e Física

Tanto a parte de teste de colisão entre objetos quanto toda a dinâmica dos mesmos são consideradas, na comunidade de desenvolvimento de jogos, como a física de um jogo (GREGORY, 2014). Para executar esses cálculos, muito utiliza-se de bibliotecas matemáticas com essas funções especificas. Existe uma biblioteca fornecida pela empresa NVIDIA gratuitamente (GREGORY, 2014), chamada PhysX (NVIDIA, 2016). Uma outra biblioteca comumente usada em jogos de grande orçamento é a Havok (HAVOK, 2016). Para o caso de jogos especificamente em duas dimensões, existe a biblioteca Box2D (BOX2D, 2016), que conta com código fonte totalmente aberto e fornece funções de simulação de física 2D em corpos rígidos

3.5 Camada de independência de plataforma

A maioria das *game engines* produzidas são projetadas com a capacidade de exportar jogos para diferentes plataformas. Cada plataforma possui um método de comunicação com seus artifícios. Para que o desenvolvedor possa, com uma simples chamada de uma função, executar um comando similar sem se preocupar com a plataforma alvo, é necessário desenvolver uma camada que identifica e filtra a função relativa à plataforma em que o jogo está sendo construído ou executado no momento. Esse filtro facilita bastante a codificação do jogo, visto que o desenvolvedor pouco precisa se preocupar com qual dispositivo está executando a função por ele desejada. Alguns exemplos de funções comuns em diferentes plataformas são: acesso ao sistema de arquivos, redes e API gráfica (GREGORY, 2014).

3.6 Sistemas central

Do mesmo modo que acontece com outros *softwares*, Gregory (2014) afirma que uma *game engine* também necessita de uma larga seleção de bibliotecas com funções úteis. Alguns exemplos de funcionalidades centrais do sistema são:

- Asserção: são funcionalidades para identificar e tratar erros e violações possíveis no sistema
- Gestão de memória: responsável por gerenciar a alocação de memória da maneira mais otimizada possível.
- Biblioteca matemática: amplamente utilizada em jogos, uma vez que se envolve trigonometria, operações com diferentes formas geométricas além da utilização de vetores e matrizes.
- Estrutura de dados e algoritmos: são utilizadas para gerenciar listas dinâmicas de forma otimizada para cada jogo e plataforma, além de normalmente contar com algoritmos de busca entre outros.

3.7 Gerenciador de recursos

O gerenciador de recursos é o responsável por carregar e fornecedor qualquer *asset* ou outro tipo de dados de arquivo disponível. Alguns exemplos de recursos fornecidos pelo gerenciador são: modelos 3D, materiais, texturas, fontes, parâmetros de física, animação de esqueleto e mapas dos cenários (GREGORY, 2014).

3.8 Motor de renderização

Segundo Gregory (2014), o motor de renderização é um dos componentes mais complexos de se implementar em uma *game engine*, e pode ser construído de várias maneiras diferentes. O autor também afirma que não há uma maneira mais eficiente de se desenvolver o motor de renderização, porém alguns padrões de *design* (NYSTROM, 2014) são amplamente seguidos em várias das *engines* já implementadas, dependendo do *hardware* gráfico da plataforma alvo. A seguir será introduzida a arquitetura mais utilizada na implementação desta camada.

3.8.1 Renderizador de baixo nível

O renderizador de baixo nível é responsável pela criação, manipulação e exibição de todas as formas geométricas primitivas recebidas, da maneira mais rápida e eficiente possível, sem se preocupar com quais componentes estão visíveis na cena.

3.8.2 *Scene-graph* e oclusões

Scene-graph[62] e oclusão é a subcamada responsável por administrar o posicionamento e hierarquia dos objetos, além de filtrar os componentes visíveis, que por sua vez devem ser enviados para a renderização. Esse componente pode ser implementado de várias maneiras, dependendo muito da escala do jogo, podendo conter, por exemplo, subdivisões espaciais.

3.8.3 Efeitos visuais

Os efeitos visuais variam dependendo da capacidade gráfica da *game engine*. Alguns efeitos geralmente utilizados são: partículas, efeitos de luz e sombras dinâmicas. A Figura 10 contém um exemplo de efeito de brilho de lente.

Figura 10 - Lens Flares.

Fonte: http://http.developer.nvidia.com/GPUGems/gpugems_ch29.html (2016).

[62] Termo em inglês, sem tradução existente.

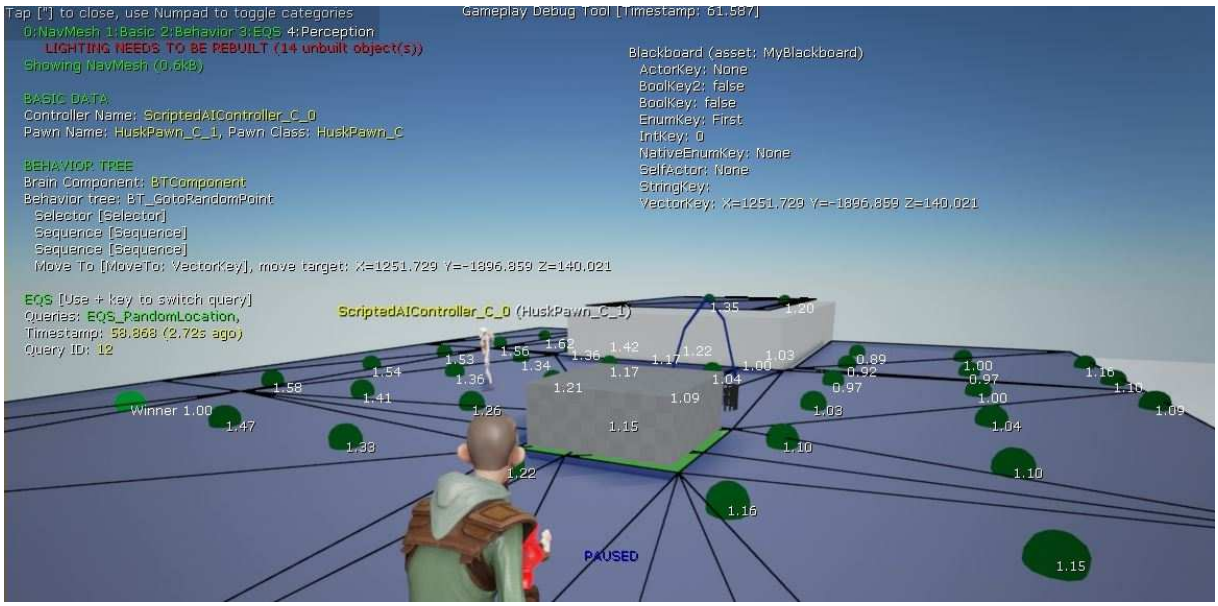
3.8.4 Front-End

O *font-end*[63] é responsável por exibir qualquer tipo de sobreposição de tela, como menus e HUDs. Esses menus podem ser tanto componentes pertencentes ao jogo, como menus, mapas e informações do jogador, quanto específicos da plataforma, como o menu de opções de um console.

3.9 Ferramentas de depuração

Do mesmo modo que em outros *softwares* a camada de ferramentas de depuração tem como objetivo, avaliar o desempenho dos mais diversos recursos do jogo. Gregory (2014) diz que há diversas ferramentas interessantes que podem ser implementadas para depurar um jogo. Algumas das ferramentas existentes em várias *game engines* contam com linhas e contornos, menu com informações e funções de teste e em alguns casos, até mesmo a capacidade de gravar *replays* que podem ser analisados várias vezes modificando variáveis. A Figura 11 contém um exemplo de tela com informações de depuração durante *gameplay* na Unreal *Engine*.

Figura 11 - Unreal *Engine Game*play Debugger.



Fonte: https://docs.unreal*engine*.com/latest/INT/*Game*play/Tools/*Game*playDebugger/ (2016)

[63] Termo em inglês, sem tradução existente.

3.10 Física

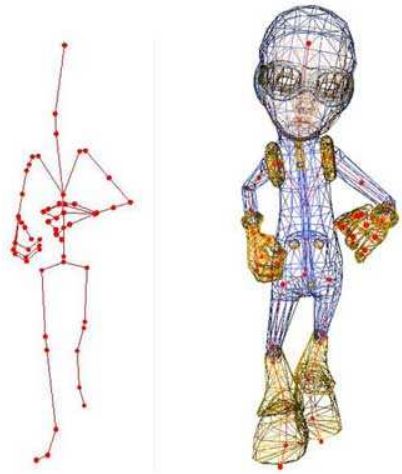
Simular a física demonstra ser importante em qualquer jogo. Sem a simulação da física, segundo Gregory (2014), seria impossível aplicar qualquer interação dinâmica com os objetos do jogo, que também afirma, que jogos podem ter um sistema de física em vários níveis, dependendo do nível de realismo desejado. Por se tratar de um sistema muito complexo e ao mesmo tempo essencial, as empresas que decidem optar por físicas realistas, não costumam criar seu próprio sistema. Ao invés disso, elas se utilizam de SDKs de terceiros especializados nesse tipo de trabalho.

3.11 Animação

Qualquer jogo que possui personagens de natureza orgânica ou semi- orgânica (como animais, humanos e robôs), necessita de uma animação de personagem (GREGORY, 2014).

Um exemplo comum de animação em jogos 3D é a Animação por Esqueleto. Neste tipo de animação o personagem possui um sistema de ossos que se conectam (Figura 12) e movimentam-se respeitando as conexões ósseas. Este modelo de animação é fortemente ligado com a física de jogo, pois grande parte dos movimentos gerados dependem da movimentação do personagem (GREGORY, 2014).

Figura 12 – Skeletal Animation.



Fonte: http://www.wazim.com/Collada_Tutorial_1.htm (2016).

No caso de jogos 2D, um modelo comumente utilizado é a animação por texturas. Neste caso de animação, cada estado do personagem é representado por uma sequência de imagens que mudam em um determinado período de tempo,

trazendo a impressão de movimentação para o personagem (GREGORY, 2014). A Figura 13 é um exemplo de grupo de imagens que, quando trocadas em sequência, trazem a impressão de que o personagem está executando um salto.

Figura 13 – Mega Man Jump Sprite Sheet.

Fonte: http://*game*dev.stackexchange.com/questions/16172/how-to-properly-handle-the-landing-part-in-a-jump-animation (2016).

3.12 Dispositivos de Interface do Usuário

Os componentes responsáveis pelo envio de *input* do usuário necessitam ser interpretados pelo jogo. Esses componentes recebem o nome de Dispositivos de Interface do Usuário (do inglês *Humam Interface Device* - HID). Alguns exemplos de dispositivos amplamente usados são *mouse*, teclado e *joystick*. Os dados obtidos por esses dispositivos podem variar de simples pressionamento de botões, a movimentos de joystick e acelerômetros. De modo a ofuscar a comunicação de baixo nível com cada um desses dispositivos, uma *game engine* costuma trazer uma interface que generaliza as ações e estados de cada HID, facilitando o uso dos mesmos em várias plataformas (GREGORY, 2014).

3.13 Áudio

Apesar de não receber a atenção merecida, o sistema de áudio de um jogo é tão importante quanto a animação, física e renderização, para trazer mais detalhes realista e ambientação de cenário. Gregory (2014) diz que algumas das *game engines* mais usadas no mercado, deixam a desejar neste aspecto, forçando os desenvolvedores que optaram por utilizar-se do *software* a estenderem as funcionalidades de áudio. Algumas ferramentas, como a API DirectX, trazem consigo um excelente leque de manipulação e reprodução de áudio de alta qualidade (GREGORY, 2014).

3.14 Multijogadores

Gregory (2014) lista as opções de multijogadores em quatro possibilidades. São elas:

- Multijogadores em tela única: dois ou mais jogadores, podendo compartilhar ou não seus respectivos HIDs, jogam olhando para a mesma tela enquanto a câmera focaliza em um determinado local.
- Multijogadores em tela única dividida: dois ou mais jogadores jogam simultaneamente, cada um com seu respectivo HID, em uma mesma tela que compartilha a visão de cada jogador.
- Multijogadores em rede: neste caso cada jogador conectado ao jogo joga em um dispositivo diferente.
- Multijogadores massivos (MMOG): uma grande quantidade de jogadores jogam simultaneamente, cada um em seu dispositivo, todos conectados em um servidor persistente.

Um importante aspecto citado por Gregory (2014) é que a existência de um sistema para multijogadores em um jogo pode não afetá-lo visualmente ou em questões de jogabilidade, porém o esforço por trás desta funcionalidade é enorme, afetando todo o sistema de leitura de HIDs, renderização, animação e a lógica do jogo. O autor também destaca que é muito mais simples criar um jogo já com suporte a multijogadores, do que adaptar um jogo já desenvolvido para suportar este sistema.

3.15 Sistema de jogabilidade

A camada de sistemas de jogabilidade é responsável pelas funcionalidades que dão vida ao jogo. Nesta camada podem-se incluir todos os objetos existentes no jogo, tanto dinâmicos quanto estáticos. Para que os objetos possam ser introduzidos no *game* sem alteração no projeto, algumas *game engines* trazem diferentes ferramentas que fazem esse papel. A mais comum é o sistema de leitura de *scripts* responsáveis por definir tanto o comportamento de cada objeto quanto a sua criação e remoção.

3.16 Subsistemas específicos do jogo

A camada de subsistemas específicos de cada jogo é dependente de qualquer outra funcionalidade do sistema. Nesta camada estão as funcionalidades que não são utilizadas em todos os jogos, pois são mecânicas específicas para cada estilo. Alguns exemplos de subsistemas inclusos em diversas *game engines* são: funções de navegação da câmera, renderização de diferentes terrenos, movimentação, física para veículos, armas e algoritmos de inteligência artificial.

4. PROJETO DE UMA *GAME ENGINE* MODULAR

Neste capítulo são apresentadas as informações técnicas da implementação da *game engine* modular desenvolvida. As informações contidas neste capítulo são referentes aos requisitos necessários para a criação e funcionamento do *software*, com descrição das ferramentas utilizadas. Por fim é relatado o processo de desenvolvimento em si.

4.1 Requisitos

Medeiros (2016) descreve em seu artigo que no início da engenharia de *software*, os requisitos de um *software* eram nada mais do que todas as funcionalidades que o mesmo era capaz de reproduzir. No entanto, atualmente assume-se que os requisitos vão além de apenas funções, sendo classificados também como objetivos, propriedades e restrições que um sistema pode possuir. Portanto, Medeiros (2016) conclui que requisitos são, na verdade, aspectos que o sistema proposto deve satisfazer, tanto durante o desenvolvimento, quanto em relação à funcionalidade, com objetivos centrais de estabelecimento de concordância com o cliente e/ou outros envolvidos. Os requisitos também necessitam conter informações sobre o que o sistema deve cumprir, fornecendo aos desenvolvedores, projetistas e testadores desse sistema, uma compreensão melhor dos requisitos.

Em engenharia de *software*, como descrito por Sommerville (2011), pode-se classificar os requisitos de um sistema como Requisitos Funcionais (RF) e Requisitos Não Funcionais (RFN). Os requisitos funcionais vão identificar as funcionalidades do sistema. No caso dos requisitos não funcionais, os mesmos representam as qualidades do *software*, como performance, usabilidade, confiabilidade e robustez, e também qualidades do processo de criação do *software*, como requisitos de entrega e implementação.

4.1.1 Requisitos funcionais

Os requisitos funcionais preocupam-se com a funcionalidade e os serviços do sistema. Pode-se dizer que os RFs são as funções que se espera que o *software* forneça em determinadas situações, de acordo com os tipos de entrada e saída de dados (SOMMERVILLE, 2011). Os RFs identificados para o sistema central foram:

- RF01: O sistema deve fornecer suporte para a implementação de módulos.
- RF02: O sistema deve fornecer suporte para a criação de objetos (entidades do jogo).
- RF03: O sistema deve suportar a criação de componentes e adição dos mesmos para qualquer objeto do jogo.
- RF04: O sistema deve fornecer suporte para criação, armazenamento e manipulação de *assets*.
- RF05: O sistema deve permitir a leitura de *inputs* do usuário.
- RF06: O sistema deve permitir a atualização do estado de cada componente e objeto através de um ciclo de tempo configurável.
- RF07: O sistema deve conter bibliotecas matemáticas com operações entre vetores, matrizes e quatérnios, além de fornecer suporte à geração de números randômicos.
- RF08: O sistema deve suportar a organização de objetos na cena como uma árvore espacial.
- RF09: O sistema necessita permitir a criação de objetos em duas e três dimensões.
- RF10: O sistema deve suportar a interação direta com as funções da API gráfica, assim como a criação de *shaders*.
- RF11: O sistema deve fornecer acesso ao sistema de arquivos
- RF12: O sistema deve fornecer suporte à reprodução de áudio.
- RF13: O sistema deve fornecer suporte ao acesso à rede de computadores.
- RF14: O sistema deve permitir acesso às funções padrão da linguagem, assim como o uso de qualquer API e bibliotecas de terceiros.

No caso dos módulos testes criados, os requisitos funcionais identificados foram:

- Módulo de *Inputs*:
    - RF15: O módulo deve permitir o mapeamento de botões ou outro tipo de periférico responsável por determinada ação.
    - RF16: Cada ação pode ser ativada por uma ou mais série de *inputs*.
    - RF17: O módulo deve oferecer informações sobre o estado da ação (ativado/desativado).
    - RF18: O módulo deve permitir o mapeamento de *inputs* analógicos com valores difusos (podendo ir de 0 a 1).
- Módulo de Renderização:

    - RF19: O módulo deve permitir a renderização de quaisquer formas trigonométricas em duas ou três dimensões.
    - RF20: O módulo deve suportar o uso de qualquer *shader* requisitado pelo usuário.
    - RF21: O módulo deve suportar o uso de texturas.
- Módulo de Objetos 2D:
    - RF22: O módulo deve suportar a criação de *sprites*.
    - RF23: O módulo deve implementar a renderização de *sprites* usando texturas.
    - RF24: O módulo deve permitir a mudança de cor dos *sprites*
    - RF25: O módulo deve fornecer informações sobre as dimensões do objeto sprite e também informações espaciais como posição, escala e rotação.
- Módulo de Colisão:
    - RF26: O módulo deve fornecer informações do resultado do teste de colisão.
    - RF27: O módulo deve fornecer teste de colisão entre:
        - Segmentos de reta
        - Ponto com retângulo
        - Ponto com circunferência
        - Retângulos
        - Circunferências
        - Retângulo e Circunferência
- Módulo de Projétil:
    - RF28: O módulo deve disponibilizar a criação e destruição de projéteis
    - RF29: Os projéteis devem possuir elementos de física como velocidade e colisão.

4.1.2 Requisitos não funcionais

Requisitos não funcionais definem restrições do sistema, que são geralmente mensuráveis, como tempo, espaço e capacidade de armazenamento. Outras restrições que se enquadram como RNFs são versões de *software*, sistema operacionais, compiladores e outras ferramentas utilizadas (SOMMERVILLE, 2011). Alguns dos requisitos não funcionais do sistema criado serão apresentados ao longo

deste capitulo, como no caso das especificações dos *hardwares,* softwares e ferramentas

4.2 Especificações

O projeto é testado em um computador pessoal com sistema operacional Windows 10 (64 bits). As configurações de *hardware* são: 8 Giga Bytes de memória RAM, com capacidade de processamento de 3.2 Giga-Hertz e placa de vídeo com 4 Giga Bytes de memória interna e processamento gráfico de até 1178 Mega-Hertz, compatível com *Open*GL 4.4 e DirectX 12. Essas configurações fazem parte dos RNFs

4.3 Ferramentas

As ferramentas utilizadas na criação da *game engine* são: a linguagem de programação Java, o *framework* libGDX, o compilador de projetos Maven e *software* Eclipse. Essas ferramentas e suas versões fazem parte dos RNFs.

4.3.1 Java

Java é uma linguagem de programação orientada a objetos (do inglês *object-oriented programming* - OOP), foi inicialmente implementada em 1991 sob o nome de Oak. Em 1995 a linguagem foi oficialmente anunciada com o nome Java. A principal motivação na sua criação foi a necessidade de uma linguagem que fosse independente de plataformas e que pudesse ser utilizada para a criação de *softwares* aptos a serem embutidos em vários dispositivos eletrônicos diferentes (SCHILDT, 2006).

A linguagem foi escolhida para a criação da *engine* pelo fato de ser suportada por todas as outras ferramentas também utilizadas no projeto. Java também conta com um grande número de documentações e exemplos disponíveis com fácil acesso. Outro motivo da escolha desta linguagem é seu suporte para criação de *softwares* multi-plataforma. A versão usada no protótipo é o Java 8.

4.3.2 LibGDX

A libGDX é um *framework* multi-plataforma utilizado na criação de jogos ou outros conteúdos de visualização. Este *framework* possibilita a criação de *softwares*

para várias plataformas, como Windows, Linux, Mac OS X, Android, Blackberry, iOS e HTML5 sem nenhuma alteração no código, podendo por exemplo, testar os jogos em uma dessas plataformas e mais tarde exportar para todas as outras. A linguagem utilizada na libGDX é o Java, por consequência, todas as funções padrões da linguagem são suportadas. Com relação as funções de baixo nível, a libGDX garante suporte à API gráfica *Open*GL através de uma interface compatível com *Open*GL ES 2.0 e 3.0. Outras funções de baixo nível existentes são: a leitura de *inputs* de diferentes aparelhos, o acesso ao sistema de arquivos e também o acesso ao sistema de áudio (LIBGDX, 2016).

Por se tratar de um *framework* com uma vasta quantidade de funcionalidades e também garantir suporte a várias plataformas e dispositivos diferentes, a libGDX se mostrou pr*open*sa a ser utilizada como pilar de todas as funcionalidades que a *game engine* oferece.

4.3.3 Maven

Maven é uma ferramenta de compilação de projetos baseada em uma arquitetura *project object model* (POM) desenvolvida para uso em projetos na linguagem Java. O principal objetivo desta ferramenta é auxiliar no processo de compilação de um sistema, mantando cada projeto separado um do outro, apenas apontando a dependência entre eles. Maven também é capaz de trazer informações como versões e histórico de alterações de cada projeto e uma lista de dependência entre os mesmos (MAVEN, 2016).

No projeto de *game engine*, a ferramenta Maven tem como função atribuir as dependências entre módulos, onde cada módulo é posto separado em um projeto. A versão utilizada no projeto é a Maven 2.

4.3.4 Eclipse IDE

A Eclipse IDE é um ambiente de desenvolvimento integrado (do inglês *integrated development environment* (IDE)) que tem suporte para a linguagem Java e também integra a ferramenta Maven (Eclipse, 2016). A versão usada no projeto é o Eclipse Mars (Eclipse 4.5).

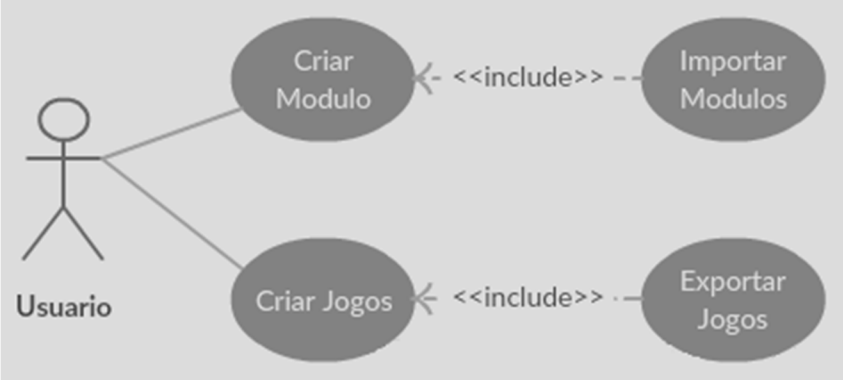
4.4 Diagramas

Os diagramas seguintes trazem informações das ações que o usuário está apto a efetuar no sistema, do modo como são estruturadas as classes do sistema principal da *game engine* e também a relação entre os módulos testes e o sistema principal.

4.4.1 Diagrama de Casos de Uso

Um diagrama de casos de uso tem como objetivo mostrar, do ponto de vista do usuário, quais são as funcionalidades que o sistema lhe oferece, sem fornecer detalhes da estrutura interna do sistema (RUMBAUGH ET. AL, 2004). A Figura 14 apresenta um diagrama de caso de uso da *game engine* desenvolvida, com as ações que podem ser executados pelo usuário.

Figura 14 – Diagrama Casos de Uso.



Fonte: Próprio autor.

As tabelas a seguir trazem detalhes de cada ação:

- Caso de Uso Criar Módulo

Tabela 2 – Caso de Uso Criar Módulo.

Caso de uso	Criar módulo
Descrição	Neste caso de uso, o usuário pode criar um novo módulo que pode ser exportado e referenciado.
Pré-condição	A *game engine* deve estar referenciada em um novo projeto criado com o Maven.

Pós- condição	O módulo contém um arquivo do Maven com informações de dependência.

Fonte: Próprio autor.

- Caso de Uso Importar Módulo

Tabela 3 – Caso de Uso Importar Módulo.

Caso de uso	Importar módulo
Descrição	Neste caso de uso, o usuário pode importar um módulo de outros projetos, incluindo de terceiros.
Pré-condição	A *game engine* deve estar referenciada em um projeto de jogo criado com Maven.
Pós- condição	O projeto do jogo contém referência de dependência do módulo importado.

Fonte: Próprio autor.

- Caso de Uso Criar Jogo

Tabela 4 – Caso de Uso Criar Jogo.

Caso de uso	Criar Jogo
Descrição	Neste caso de uso, o usuário pode criar jogos, utilizando-se de módulos importados.
Pré-condição	A *game engine* e todos os módulos utilizados devem estar referenciados em um projeto de jogo criado com Maven.
Pós- condição	O projeto do jogo contém uma classe de instância do jogo que pode ser usada para exportar o mesmo para qualquer plataforma suportada.

Fonte: Próprio autor.

- Caso de Uso Exportar Jogo

Tabela 5 – Caso de Uso Criar Jogo.

Caso de uso	Exportar Jogo

Descrição	Neste caso de uso, o usuário pode exportar jogos para diferentes plataformas.
Pré-condição	O projeto do jogo a ser exportado deve estar referenciado em um projeto criado com Maven. Este projeto contém necessariamente uma interface da libGDX referente à plataforma a ser exportada.
Pós- condição	Todos os arquivos necessários para a execução do jogo na plataforma escolhida serão exportados.

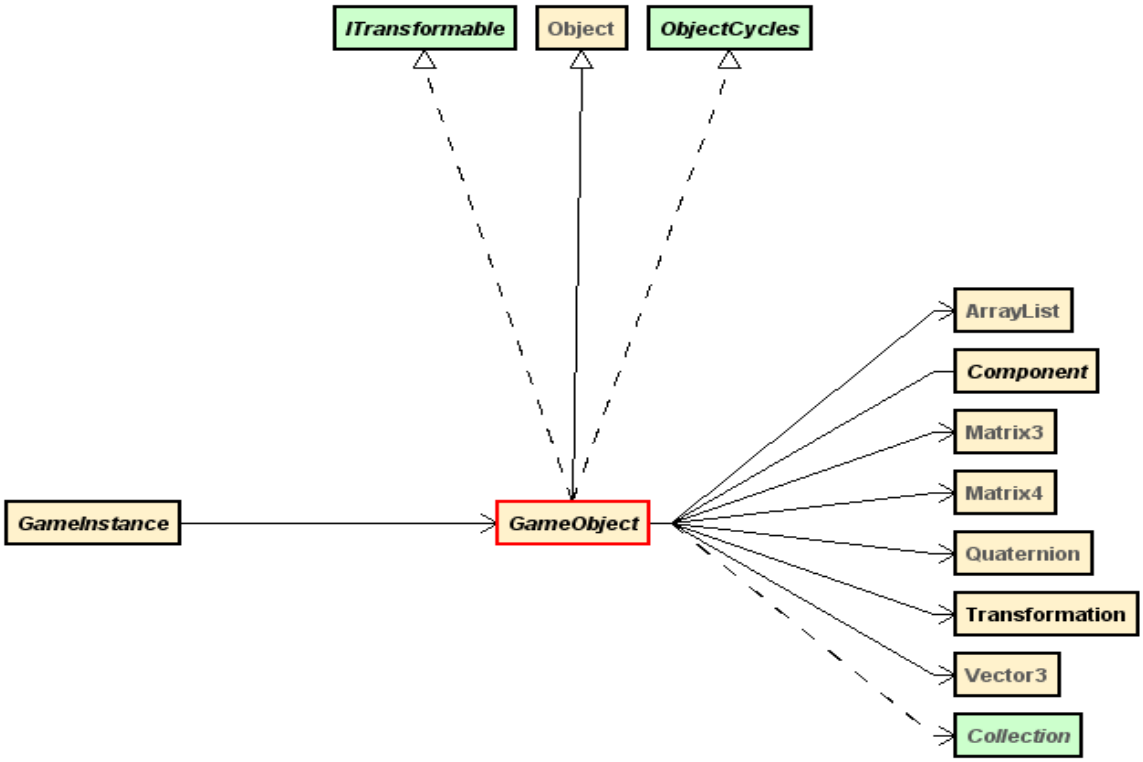
Fonte: Próprio autor.

4.5.1 Diagrama de Classes

Um diagrama de classes demonstra os componentes de cada classe e suas relações com outras classes (MARTIN, 2002). Na *game engine* desenvolvida, existem 3 classes que compõem o sistema principal na criação de módulos e jogos. As figuras a seguir são diagramas que mostram os detalhes e referências entre elas.

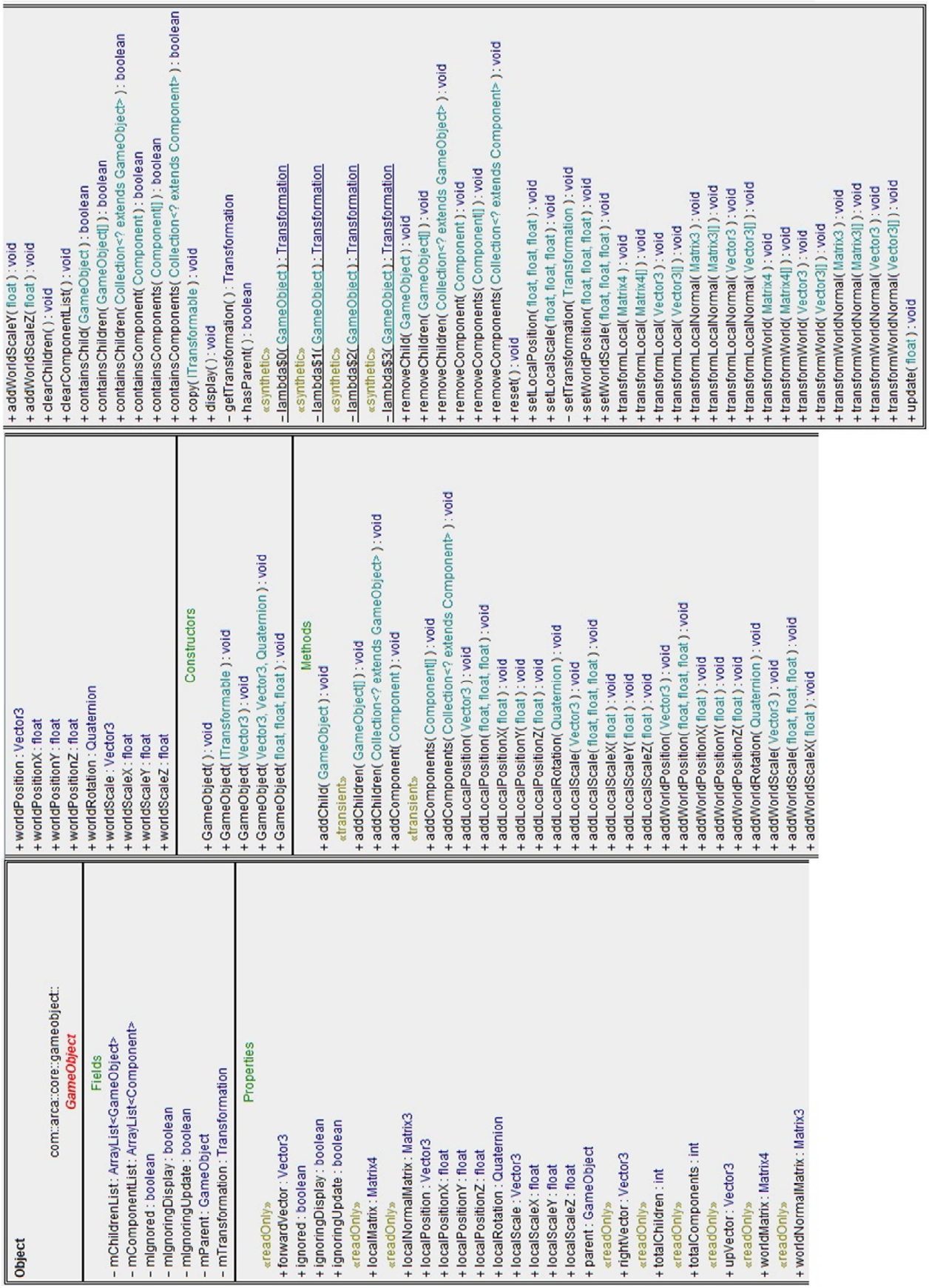
A classe *Game* Object é a base para qualquer entidade que exista dentro da cena. A Figura 15 traz as referências da classe. A Figura 16 traz os componentes da classe.

Figura 15 – *Game* Object class diagram referencies



Fonte: Próprio autor.

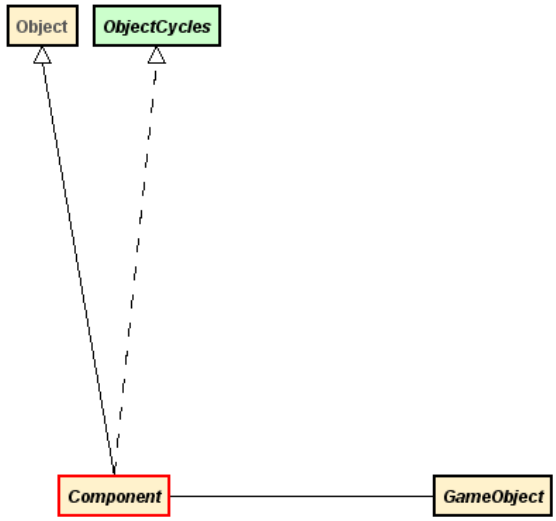
Figura 16 – *Game* Object class diagram components



Fonte: Próprio autor.

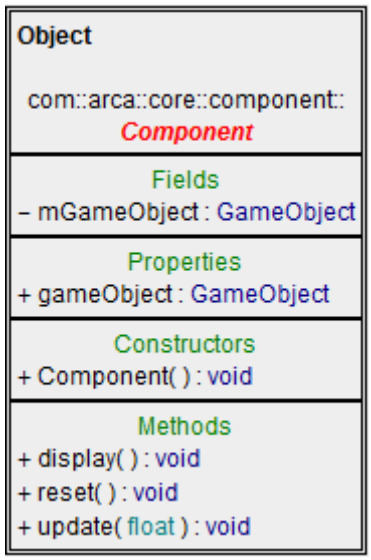
A classe Component é a base de qualquer componente pertencente a um objeto do jogo. Esta classe permite atualizar o estado do componente a cada período de tempo. A Figura 17 traz as referências da classe. A Figura 18 traz os componentes da classe.

Figura 17 – Component class diagram referencies.



Fonte: Próprio autor.

Figura 18 – Component class diagram components.

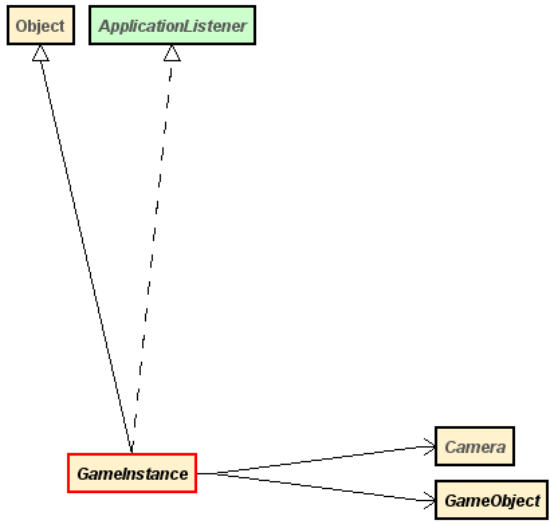


Fonte: Próprio autor.

A classe *Game* Instance é responsável por administrar a instância do jogo. Esta instancia é totalmente independente da plataforma, por consequência é a classe

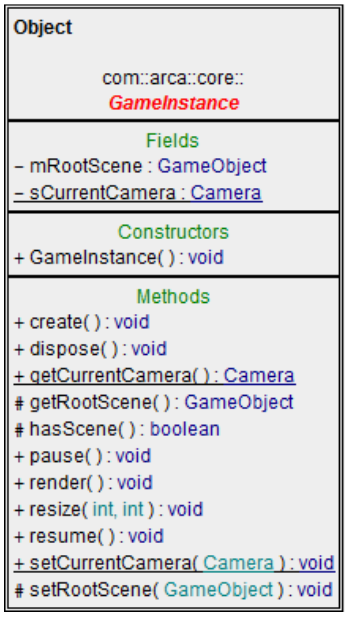
referenciada nos projetos de exportação para as diferentes plataformas. A Figura 19 traz as referências da classe. A Figura 20 traz os componentes da classe.

Figura 19 – *Game* Instance class diagram referencies.



Fonte: Próprio autor.

Figura 20 – *Game* Instance class diagram components.

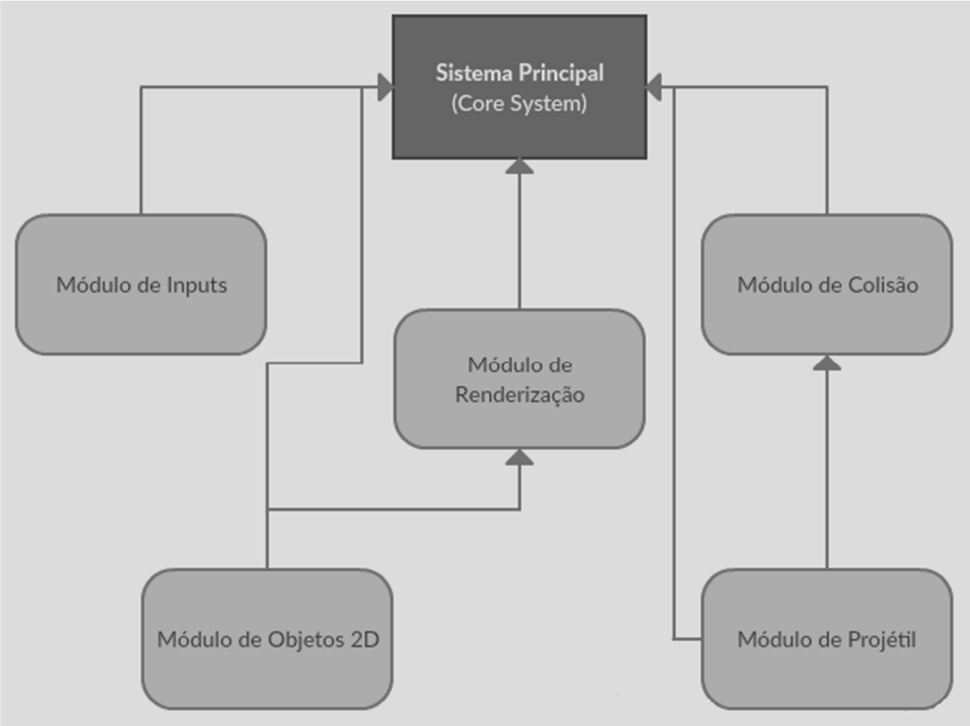


Fonte: Próprio autor.

4.5.2 Dependências

O diagrama representado na Figura 21 identifica a relação de dependência de cada módulo com o sistema principal e entre os mesmos.

Figura 21 – Diagrama de dependencia dos módulos.



Fonte: Próprio autor.

5 TESTES

De modo a observar a eficiência do modelo de *game engine* modular. Foi criado um jogo parcial, com intuito de utilizar-se de todos os módulos testes desenvolvidos e também o sistema principal da *game engine*.

5.1 Distribuição dos módulos

Para serem utilizados no protótipo de jogo, os módulos necessitavam ser referenciados como dependência para que suas funcionalidades pudessem ser exploradas. A Figura 22 exemplifica como ficaram os arquivos de dependência de cada módulo no modelo Maven.

Figura 22 – Arquivos de dependência.

Módulo de *Inputs*

```
<groupId>com.arca.inputs</groupId>
<artifactId>arca-inputs</artifactId>
<version>1.0</version>

<dependencies>
    <dependency>
        <groupId>com.arca</groupId>
        <artifactId>arca</artifactId>
        <version>1.0</version>
    </dependency>
</dependencies>
```

Módulo de Renderização

```
<groupId>com.arca.renderer</groupId>
<artifactId>arca-renderer</artifactId>
<version>1.0</version>

<dependencies>
    <dependency>
        <groupId>com.arca</groupId>
        <artifactId>arca</artifactId>
        <version>1.0</version>
    </dependency>
</dependencies>
```

Módulo de Colisão

```
<groupId>com.arca.collision</groupId>
<artifactId>arca-collision</artifactId
<version>1.0</version>

<dependencies>
    <dependency>
        <groupId>com.arca</groupId>
        <artifactId>arca</artifactId>
        <version>1.0</version>
    </dependency>
</dependencies>
```

Módulo de Objetos 2D

```
<groupId>com.arca-g2d</groupId>
<artifactId>arca-g2d</artifactId>
<version>1.0</version>

<dependencies>
    <dependency>
        <groupId>com.arca</groupId>
        <artifactId>arca</artifactId>
        <version>1.0</version>
    </dependency>
    <dependency>
        <groupId>com.arca.renderer</groupId>
        <artifactId>arca-renderer</artifactId>
        <version>1.0</version>
    </dependency>
</dependencies>
```

Módulo de Projéteis

```
<groupId>com.arca.projectiles</groupId>
<artifactId>arca-projectiles</artifactId>
<version>1.0</version>

<dependencies>
    <dependency>
        <groupId>com.arca</groupId>
        <artifactId>arca</artifactId>
        <version>1.0</version>
    </dependency>
    <dependency>
        <groupId>com.arca.collision</groupId>
        <artifactId>arca-collision</artifactId>
        <version>1.0</version>
    </dependency>
</dependencies>
```

Fonte: Próprio autor.

5.2 Implementação

A implementação consiste em criar elementos de um jogo de tiro no espaço (do ingles *space shooter*). Para isso foi implementado algumas classes fundamentais, como uma classe responsável pelo jogador, uma classe que representa um inimigo, e uma classe responsável pela cena do jogo, esta última se encarrega de atualizar o

estado do sistema, checando *inputs*, modificando o posicionamento de objetos e instanciando ou excluindo os mesmos quando necessário.

5.2.1 Jogador e Inimigo

De modo a testar a funcionalidade do módulo de renderização e do módulo de objetos 2D, foram criadas duas classes de entidades do jogo, uma representando o jogador, chamada de PlayerSpaceship e a outra representando um inimigo, chama EnemySpaceship. Ambas as classes herdam a classe base chamada Spaceship. Nesta classe base são aplicadas algumas funcionalidades do módulo de renderização e do módulo de objetos 2D.

Figura 23 – Spaceship Class.

```
public abstract class Spaceship extends GameObject{

	protected RectangularSprite mSprite;

	public Spaceship(){

	}
}
```

Fonte: Próprio autor.

A Figura 23 mostra a implementação da classe abstrata Spaceship, que possui uma referência de objeto do tipo RectangularSprite. Este objeto pertence ao módulo de objetos 2D. Nele é implementado, com auxílio das funcionalidades do módulo de renderização, funções responsáveis pela manipulação e renderização de objetos retangulares e texturizados em duas dimensões.

Figura 24 – PlayerSpaceship Class.

```
public class PlayerSpaceship extends Spaceship{

	private GameObject mBulletSpawnPoint;
	private Timer mShootInterval;

	public PlayerSpaceship(){
		TextureRegion texture =
TCCGame.getAssetsManager().getTexture("Player Spaceship");

		mSprite = new
RectangularSprite(TCCGame.getAssetsManager().getMesh("Player Spaceship"),
texture);
		addChild(mSprite);

		setWorldScale(3, 3, 1);

		mBulletSpawnPoint = new GameObject() {
		};
		mBulletSpawnPoint.setWorldPositionY((texture.getRegionHeight()
/ 2f) + 1);
		addChild(mBulletSpawnPoint);

		mShootInterval = new Timer(0.2f);
		addComponent(mShootInterval);
	}

	public Bullet[] shoot() {
		Bullet bullet = new Bullet(new Vector3(0, 500, 0));
		bullet.setWorldPosition(mBulletSpawnPoint.getWorldPosition());

		mShootInterval.start();

		return new Bullet[] { bullet };
	}

	public boolean canShoot(){
		return !mShootInterval.isRunning();
	}
}
```

Fonte: Próprio autor.

A Figura 24 traz a implementação da classe do jogador, chamada PlayerSpaceShip. Nesta classe há duas propriedades privadas, a primeira, chamada de mBulletSpawnPoint é do tipo *Game*Object e a segunda, chamda mShootImterval é do tipo Timer, ambas são responsáveis, respectivamente, pelo ponto de geração do projétil e o intervalo de geração do mesmo.

No construtor da classe nota-se que é instanciado a propriedade mSprite, anteriormente citada na classe pai Spaceship. Para que este objeto seja instanciado é necessário acessar a ferramenta de *assets* que contém as informações geométricas

e de textura da nave espacial do jogador que será posteriormente renderizada por essa propriedade.

Por fim a classe contém dois métodos de acesso público, responsáveis por autorizar o disparo de projeteis e também por cria-los.

Figura 25 – EnemySpaceship Class.

```
public class EnemySpaceship extends Spaceship{

	public EnemySpaceship() {
		TextureRegion texture =
TCCGame.getAssetsManager().getTexture("Pawn Spaceship");

		mSprite = new
RectangularSprite(TCCGame.getAssetsManager().getMesh("Pawn Spaceship"),
texture);
		addChild(mSprite);

		setWorldScale(3, 3, 1);
		setLocalRotation(new Quaternion(new Vector3(0, 0, 1), 180));
	}
}
```

Fonte: Próprio autor.

A implementação da classe de inimigo, chamada EnemySpaceship, está representada na Figura 25. Assim como ocorre da classe PlayerSpaceship, no construtor desta classe é instanciado a propriedade mSprite, com as informações geométricas e de textura do espaço nave inimiga. A única diferença notável nesta classe é a ausência de funcionalidades de disparo de projeteis, pois neste teste, esta função é exclusiva do jogador.

5.2.2 *Input*

Para que fosse possível testar a funcionalidade do módulo de *inputs*, foram criadas 3 ações possíveis de serem executadas pelo jogador. São elas: movimentar-se na horizontal, movimentar-se na vertical e atirar projéteis. No caso das ações de movimentação, onde envolve-se direções, foi utilizado vetores que indicam a direção que o usuário deseja movimentar seu personagem. Esta direção é definida pela combinação de teclas acionadas, que por sua vez é calculada com ajuda da classe InputAxis. No caso da ação de atirar projéteis, foi criado um objeto da classe InputAction, que é responsável por determinar quando algum *input* responsável por executar a ação foi acionado. Tanto a classe InputAxis quanto a InputAction fazem

parte do módulo de *inputs*. A Figura 26 demonstra como foi implementado estas funcionalidades dentro da cena principal do jogo.

Figura 26 – Implementação dos *inputs*.

```
private InputAxis mSideInput, mForwardInput;
private InputAction mFireInput;

public Scene() {
            mSideInput = new InputAxis(new InputValueGetter() {

                  @Override
                  public float getValue() {
                        return ((Gdx.input.isKeyPressed(Keys.LEFT) ||
Gdx.input.isKeyPressed(Keys.A)) ? -1f : 0f) +
                                    ((Gdx.input.isKeyPressed(Keys.RIGHT) ||
Gdx.input.isKeyPressed(Keys.D)) ? 1f : 0f);
                  }
            });

            mForwardInput = new InputAxis(new InputValueGetter() {

                  @Override
                  public float getValue() {
                        return ((Gdx.input.isKeyPressed(Keys.DOWN) ||
Gdx.input.isKeyPressed(Keys.S)) ? -1f : 0f) +
                                    ((Gdx.input.isKeyPressed(Keys.UP) ||
Gdx.input.isKeyPressed(Keys.W)) ? 1f : 0f);
                  }
            });

            mFireInput = new InputAction(new InputStateGetter() {

                  @Override
                  public boolean isPressed() {
                        return Gdx.input.isKeyPressed(Keys.SPACE);
                  }
            });
      }
```

Fonte: Próprio autor.

5.2.3 Projétil e Colisão

O módulo de projétil contém uma classe denominada de Bullet, que é responsável por simular a funcionalidade de um projétil genérico. Pode-se observar na Figura 27, que a há uma propriedade denominada mVelocity do tipo Vector3. Esta propriedade é um vetor que define a velocidade em que o projétil viaja pela cena a cada quadro, ou seja, nela está definido a direção e módulo da velocidade. Na classe Bullet, representada na Figura 27, também é possível observar a implementação de um método chamado hasHit, que retorna um valor de falso ou verdadeiro. Este método tem o objetivo de dizer se algum corpo retangular está colidindo com os limites do

projétil. Para que esse cálculo seja feito, utiliza-se uma função fornecida pelo módulo de colisão, chamada checkAABBS, que por sua vez é encarregado de calcular a intersecção entre retângulos de eixos alinhados.

Figura 27 – Implementação da classe Bullet.

```
public class Bullet extends GameObject{

	private RectangularSprite mSprite;
	private Vector3 mVelocity;

	public Bullet(Vector3 pVelocity){
		TextureRegion texture =
TCCGame.getAssetsManager().getTexture("Player Spaceship Bullet");

		mSprite = new
RectangularSprite(TCCGame.getAssetsManager().getMesh("Player Spaceship
Bullet"), texture);
		addChild(mSprite);

		mVelocity = pVelocity;
	}

	public boolean hasHit(RectangleBounds bounds){
		return Collision.checkAABBs(bounds.getLeft(),
bounds.getRight(), bounds.getBottom(), bounds.getTop(),
				mSprite.getBounds().getLeft(),
mSprite.getBounds().getRight(), mSprite.getBounds().getBottom(),
mSprite.getBounds().getTop());
	}

	public Vector3 getVelocity() {
		return mVelocity.cpy();
	}
}
```

Fonte: Próprio autor.

A Figura 28 contém um trecho do código da atualização da cena, onde checa-se o *input* de atirar mísseis. Caso o jogador tenha acionado esta ação, um projétil é instanciado.

Figura 28 – Criação de projéteis.

```
if(mFireInput.getState() == InputState.PRESSED && mPlayer.canShoot()){
	List<Bullet> bullets = Arrays.asList(mPlayer.shoot());
	mBullets.addAll(bullets);
	addChildren(bullets);
}
```

Fonte: Próprio autor.

A Figura 29 demonstra a parte do código responsável por atualizar o estado de cada projétil existente na cena. Um a um os projéteis são movimentados de acordo com o intervalo de tempo entre um quadro e outro e logo em seguida é testado se houve alguma colisão com o inimigo.

Figura 29 – Atualização do estado dos projéteis.

```
//Atualizacao de cada projétil todo quadro
for (Bullet bullet : mBullets) {

      bullet.addWorldPosition(bullet.getVelocity().scl(pTimeElapsed));


      if(bullet.hasHit(mEnemySpaceship.getSprite().getBounds())){
                        destroyEnemy(bullet, mEnemySpaceship);
      }
}
```

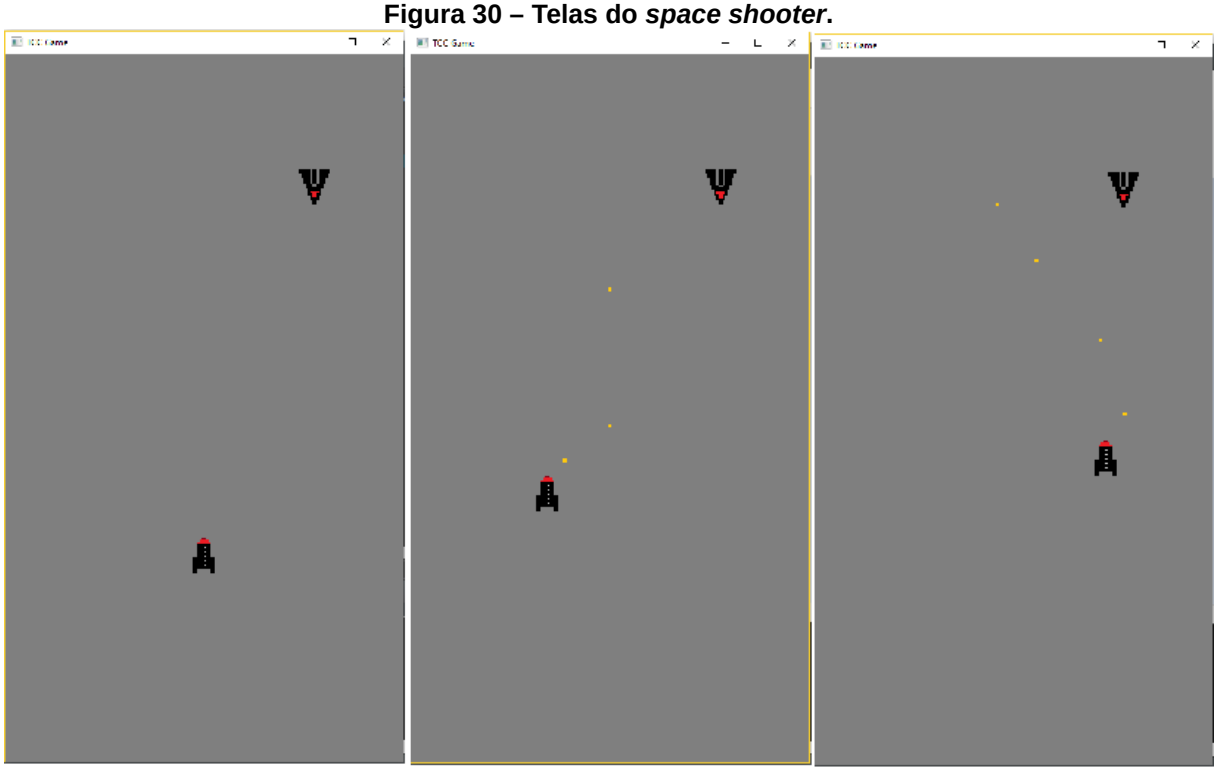
Fonte: Próprio autor.

5.3 Resultado

Após o término da implementação de todo o código necessário para a jogabilidade, o projeto de exportação teve como alvo a plataforma PC, que com ajuda da interface fornecida pelo *framework* libGDX, foi exportado com sucesso.

O resultado obtido foi exatamente como esperado. O protótipo de *space shooter* contou com um personagem controlável por comandos de *input* do jogador que era capaz de disparar mísseis em linha reta. Caso algum desses mísseis atingissem o objeto que representava um inimigo, este era destruído. A Figura 30 demonstra uma sequência de telas extraídas do jogo.

Figura 30 – Telas do *space shooter*.



Fonte: Próprio autor.

6 CONSIDERAÇÕES FINAIS

Com base nas informações coletadas através da análise de algumas das *game engines* disponíveis no mercado, juntamente com um estudo dos padrões de arquitetura de um motor de criação de jogos, foi possível criar um protótipo de *game engine* com sucesso no modelo modular.

Através da criação de um jogo teste, foi possível identificar a viabilidade de uma *game engine* modular. Os testes foram executados com sucesso, mostrando o quanto uma *game engine* pode ser modularizada em infinitas partes a fim de proporcionar ao usuário mais opções de ferramentas que não estejam embutidas no sistema principal do *software*. Portanto, a criação de uma *game engine* modular mostra-se possível de ser implementada e utilizada na criação de jogos digitais.

Para fins de estudos futuros, indica-se a implementação de mais módulos, como por exemplo, módulos com funções de inteligência artificial, física, objetos tridimensionais, animações e suporte a multijogadores. Outro fator a ser levado em conta em trabalhos futuros seria a comparação do desempenho, tanto em questões de performance computacional quanto em tempo de desenvolvimento, entre uma *game engine* desenvolvida no modelo modular com outros tipos de arquiteturas existentes.

Por fim, fica aberta a possibilidade de implementações deste modelo de *game engine* utilizando outras ferramentas e linguagens de programação, afim de aprimorar os resultados e encontrar versões cada vez mais viáveis deste modelo.

REFERÊNCIAS BIBLIOGRÁFICAS

BEAL, Vangie. Driver. Webopedia. Disponível em: http://www.webopedia.com/TERM/D/driver.html. Acessado em 17 de nov. de 2016.

BOO. Site Oficial. Disponível em: http://boo-lang.org/. Acessado em 25 de out. de 2016.

BOX2D. Site Oficial. Disponível em http://box2d.org/. Acessado em 17 de nov. de 2016.

CPLUSPLUS. Containers. Disponível em; http://www.cplusplus.com/reference/stl/. Acessado em 03 de nov. de 2016.

DONNERSTAG. Procedural Terrain Heightmap Generation using DLA (Diffusion Limited Aggregation). Voxel Game Engine Development: Sparse Voxel Octree Raycasting based Game Development, 2014. Disponível em: http://voxels.blogspot.com.br/2014/01/procedural-terrain-heightmap-generation.html. Acessado em 23 de out. de 2016.

ECLIPSE. Site oficial. Disponível em: https://eclipse.org/. Acessado em 15 de nov. de 2016.

FOLEY, James D. et al. “Spatial-partitioning representations; Surface detail”. Computer Graphics: Principles and Practice. The Systems Programming Series. Addison-Wesley, 1990.

GREGORY, Jason. Game Engine Architecture. Second Edition. 6000 Broken Sound Parkway NW, Suite 300. Taylor & Francis Group, 2014. 1014 p.

HAVOK. Site Oficial. Disponível em: http://www.havok.com/. Acessado em 04 de nov. de 2016.

KESSENICH, John M.; SELLERS, Graham; SHREINER, Dave. OpenGL Programming Guide. 9 Edition. Addison-Wesley Professionao, 2016. 976 p.

KHAN Academy. Route-finding. Disponível em: https://www.khanacademy.org/computing/computer-science/algorithms/intro-to-algorithms/a/route-finding. Acesso em 23 de out. de 2016.

LIBGDX. Bad Logic Games. Disponível em: https://libgdx.badlogicgames.com/. Acessado em 15 de nov. de 2016.

LUNA, Frank. Introduction to 3D Game Programming with DirectX 12. Mercury Learning & Information, 2016.

MARTIN, C. Robert. UML for Java Programmers. Prentice Hall, Englewood Cliffs, New Jersey 07632. Object Mentor Inc., 2002. 246 p.

MAVEN. What is Maven?. Apache Maven Project. Disponível em: https://maven.apache.org/what-is-maven.html. Acessado em 15 de nov. de 2016.

MEDEIROS, Higor. Introdução a Requisitos de Software. DevMedia. Disponível em: http://www.devmedia.com.br/introducao-a-requisitos-de-software/29580. Acessado em 07 de nov. de 2016.

MICROSOFT. Developer Network. Guia de Programacao em C#. Disponível em: https://msdn.microsoft.com/pt-br/library/67ef8sbd.aspx. Acessado em 25 de out. de 2016.

MONOGAME. About MonoGame. Disponível em: http://www.monogame.net/about/. Acessado em 17 de nov. de 2016.

MOZILA. Developer Network. Javascript. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript. Acessado em 25 de out. de 2016.

NAYLOR, F. Bruce. A Tutorial on Binary Space Partitioning Trees. Spatial Labs Inc. 25p. Disponível em: http://www.bowdoin.edu/~ltoma/teaching/cs350/fall13/Material/bsp_tutorial.pdf. Acessado em 23 de out. de 2016.

NVIDIA. PhysX SDK. Disponível em: https://developer.nvidia.com/physx-sdk. Acessado em 04 de nov. de 2016.

NYSTORM, Robert. Game Programming Patterns. 1 Edition. Genever Benning, 2014. 354 p.

RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. The Unified Modeling Language Reference Manual. Second Edition. Addison-Wesley, 2004. 741 p.

SCHANTZ, E. Richard; SCHMIDT, C. Douglas. Middleware for Distributed Systems. 9 p. Disponível em: https://www.dre.vanderbilt.edu/~schmidt/PDF/middleware-encyclopedia.pdf. Acessado em 17 de nov. de 2016.

SCHILDT, Herbert. Java the Complete Reference. Seventh Edition. Mc Graw Hill, Osborne, 2006. 1056 p.

SOMMERVILLE, Ian. Engenharia de Software. Edição: 9ª. Pearson, 2011. 529 p.

WILSON, Joe. Tutorial: Skin Shader. Marmoset Toolbag 2. Disponível em: http://www.marmoset.co/toolbag/learn/skin-shader. Acessado em 23 de out. de 2016.