Paulo Maria Neto

Desenvolvimento de um motor para criação de jogos em java

Araraquara/SP

junho, 2018

Paulo Maria Neto

Desenvolvimento de um motor para criação de jogos em java

Trabalho de conclusão de curso para obtenção do título de graduação em Ciência da Computação apresentado a Universidade Paulista - UNIP

UNIP - Universidade Paulista

Orientador: Profª. Mª. Daniele Colturato

Araraquara/SP
junho, 2018

Paulo Maria Neto
Desenvolvimento de um motor para criação de jogos em java/ Paulo Maria Neto. – Araraquara/SP, junho, 2018-
47 p. : il. (algumas color.) ; 30 cm.

Orientador: Profª. Mª. Daniele Colturato

Monografia (Graduação) – UNIP - Universidade Paulista, junho, 2018.

1. Computação gráfica. 2. Desenvolvimento de jogos. 3. Game engine. I. II. Universidade Paulista. III. Faculdade de Ciência da Computação. IV. Desenvolvimento de um motor para criação de jogos em java

CDU 02:141:005.7

Paulo Maria Neto

Desenvolvimento de um motor para criação de jogos em java

Trabalho de conclusão de curso para obtenção do título de graduação em Ciência da Computação apresentado a Universidade Paulista - UNIP

Trabalho aprovado. Araraquara/SP, 01 de junho de 2018:

Profª. Mª. Daniele Colturato
Orientador - Universidade Paulista - UNIP

Prof.
Universidade Paulista - UNIP

Prof.
Universidade Paulista - UNIP

Araraquara/SP
junho, 2018

Dedico esse trabalho a minha familia, por me apoiar incondicionalmente em todos os momentos importantes da minha vida e da minha formação.

Agradecimentos

Agradeço primeiramente à Deus, por ter me concedido saúde, força e disposição para fazer a faculdade e o trabalho de final de curso.

Agradeço aos meus pais Joaquim e Arabela, que me deram apoio e incentivo em todos os momentos da minha vida.

Agradeço a todos os professores que me proporcionaram o conhecimento para que eu pudesse chegar onde estou hoje.

Agradeço a professora Daniele Colturato, que me deu todo o suporte com suas correções e orientações.

Agradeço a universidade UNIP e a todos que direta ou indiretamente fizeram parte da minha formação.

“Nem tudo que se enfrenta pode ser modificado,
mas nada pode ser modificado até que seja enfrentado.“
(Albert Einstein)

Resumo

Com o passar dos anos, o avanço da tecnologia e a expansão do mercado de jogos, tornou-se cada vez mais necessária a reutilização de códigos e automatização de rotinas mais utilizadas no desenvolvimento de jogos como renderização, controle das entradas do usuário e gerenciamento de recursos. *Game Engines* ou Motores de Jogos são softwares que cuidam dessas rotinas, oferecendo mecanismos genéricos e reaproveitáveis para diminuir o custo de produção e simplificar as tarefas durante o desenvolvimento de jogos. Esta monografia apresenta o desenvolvimento de uma *game engine* chamada Pulsar Game Engine, que implementa os sistemas básicos necessários para o desenvolvimento de jogos em duas dimensões de qualquer gênero, fornecendo assim uma ferramenta de fácil customização e expansão para o desenvolvimento de jogos mais complexos.

Palavras-chaves: Computação gráfica, Desenvolvimento de jogos, Game engine.

Abstract

Over the years with the advancement of technology and the expansion of the gaming market, it has become increasingly necessary to reuse codes and automate routines that are more commonly used in game development such as rendering, user input control, and game management. resources. Game Engines are softwares that takes care of these routines, offering generic and reusable mechanisms to decrease the cost of production and simplify tasks during game development. This paper presents the development of a game engine called Pulsar Game Engine, which implements the basic systems needed to develop any kind of two-dimensional game, providing a tool for easy customization and expansion for game development more complex.

Key-words: Computer graphics , Game development , Game engine

Lista de ilustrações

Lista de abreviaturas e siglas

LWJGL	*Lightweight Java Game Library*
OpenGL	*Open Graphic Library*
OpenAL	*Open Audio Library*
GLSL	*OpenGL Shading Language*
HLSL	*High Level Shading Language*
API	*Application Programming Interface*
SDK	*Software Development Kit*

Sumário

1 Introdução

Os primeiros jogos de vídeo game foram produzidos por equipes pequenas, com gráficos minimalistas, utilizando linguagens de baixo nível e com praticamente nenhum reaproveitamento de código devido as limitações de hardware e ao alto custo de produção da época. Com o passar dos anos, hardwares mais potentes, equipes maiores e a busca por gráficos e comportamentos mais realistas, tornou-se inviável o desenvolvimento de jogos do zero sem uma ferramenta que reaproveitasse os elementos em comum entre os jogos, esses softwares são chamados de Motores de jogos ou *Game Engines*. Geralmente *Game Engines* são vistas como softwares extremamente complexos e de alta performance que oferencem dezenas de recursos para o desenvolvimento (como a Unreal Engine e a Unity por exemplo), porem a maioria desses recursos acabam servido apenas como ferramentas auxiliares e não fazem parte do produto final, dessa forma é possivel desenvolver uma *Game Engine* simples e amigável com muito menos código e mais simples de se utilizar.

1.1 Motivação

Apesar de existirem várias ferramentas que auxiliam no desenvolvimento, muitas delas são pagas, pouco customizáveis ou muito complexas para serem utilizadas por desenvolvedores iniciantes ou que não possuem conhecimento profundo sobre computação gráfica.

1.2 Objetivo

O objetivo deste trabalho é descrever o funcionamento dos sistemas básicos e essenciais de uma *game engine* e desenvolver um conjunto de ferramentas de código aberto que facilite o desenvolvimento de jogos oferecendo flexibilidade para que o usuário possa desenvolver jogos de qualquer estilo e com facilidade de extensão.

1.3 Objetivos Gerais

Desenvolver um motor que auxilie no desenvolvimento de jogos utilizando Java e JavaScript.

1.4 Objetivos Específicos

- Desenvolvimento de um sistema de renderização e animação 2D com suporte a OpenGL 3.0 ou superior
- Implementação de suporte a reprodução de audio e efeitos sonoros utilizando OpenAL
- Implementação de suporte a simulação de física 2D utilizando a biblioteca Box2D
- Implementação de controle sobre as entradas do usuário via teclado, mouse e controles USB
- Implementação de suporte a scripts desenvolvidos em JavaScript para o controle de lógica em tempo de jogo
- Desenvolvimento de um template base para a criação de jogos sem a necessidade de se alterar diretamente o código fonte da engine

Os elementos disponíveis e as ferramentas utilizadas para o desenvolvimento de uma *game engine* variam de acordo com o resultado que se deseja obter. Esse trabalho apresenta a seguinte estrutura:

- Capítulo 2 - Descrição de elementos comuns em *game engines*.
- Capítulo 3 - Descrição do ciclo de execução e funcionamento interno da Pulsar Game Engine.
- Capítulo 4 - Conclusões retiradas a partir deste trabalho.

2 Game Engine

Segundo Brito (2011), Gregory (2011) e Eberly (2001), uma *engine* ou motor de jogo é um software extensível e reutilizável responsável por simular a parte física do mundo real dentro do ambiente do jogo. Uma *game engine* deve fornecer abstração de hardware para que o desenvolvedor possa se concentrar mais na lógica executada em jogo e menos em gerenciar os recursos da plataforma de destino.

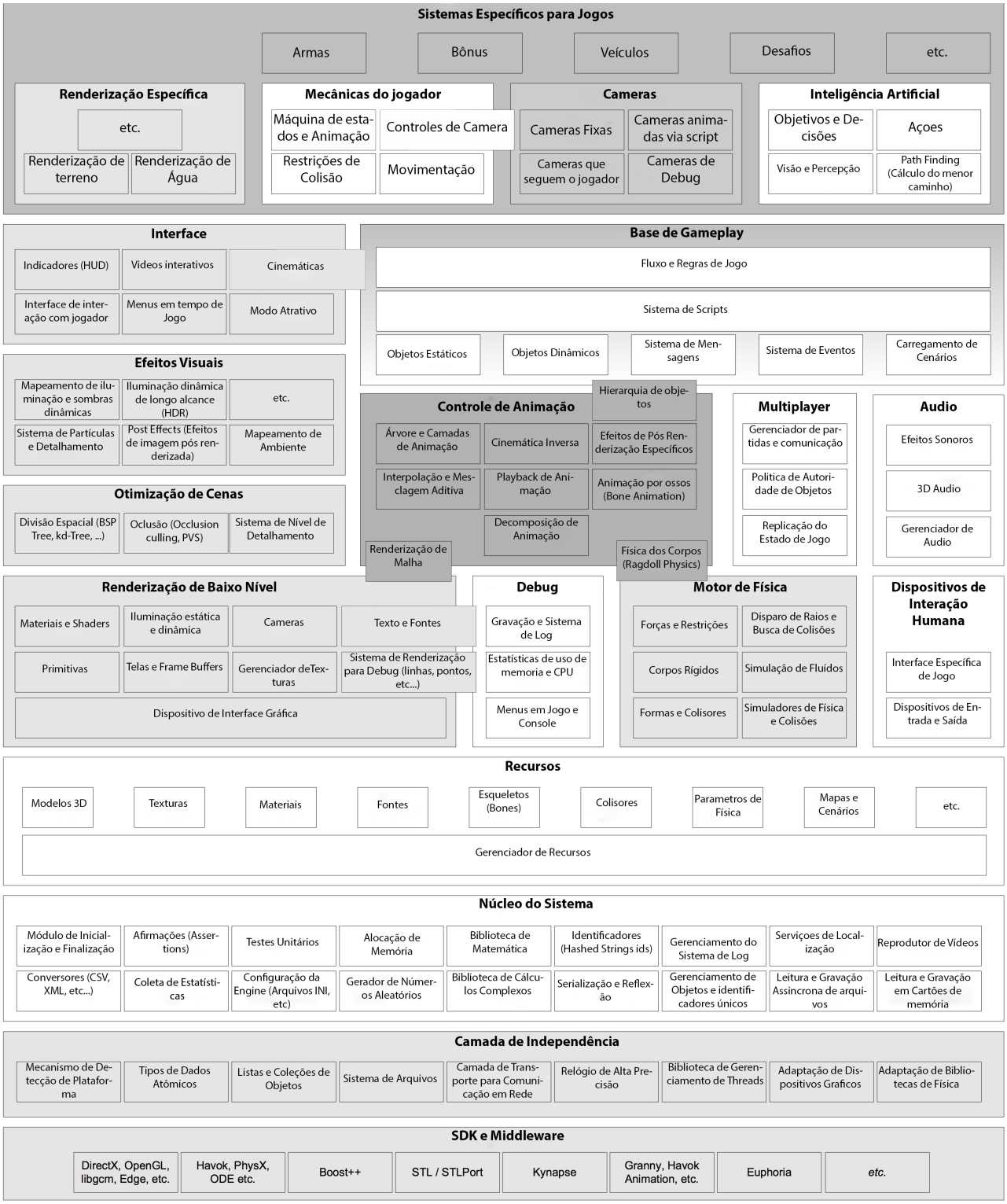
2.1 Arquitetura de uma Game Engine

Assim como a maioria dos softwares as *game engines* são geralmente desenvolvidas em camadas, onde cada camada depende da camada inferior para que possa executar a ação, mas nunca o contrário, dessa forma evitam-se dependências cíclicas que podem tornar o código intestável e causar comportamentos indesejáveis na aplicação.
As camadas podem ser adicionadas ou removidas dependendo dos recursos que estarão disponíveis nos jogos desenvolvidos pela engine, entretanto existem algumas camadas que são indispensáveis para o funcionamento correto de uma *engine*, estes elementos são:

- Núcleo do Sistema
- Motor de Renderização
- Motor de Física
- Motor de Áudio
- Motor de Lógica
- Dispositivos de interação com o usuário

A figura 1 mostra um exemplo de arquitetura em camadas utilizada em *game engines 3D*.

Figura 1: Arquitetura de uma game engine



Fonte: Game engine architecture (2011, Página 29)

2.1.1 SDK e Middleware

SDK e middlewares são bibliotecas que fornecem abstrações dos recursos de hardware ou software que serão utilizados pela aplicação, a maioria das game engines utilizam

essas bibliotecas afim de agilizar e facilitar o desenvolvimento. Nessas bibliotecas podem estar incluídos algoritmos de inteligência artificial, acesso e manipulação de dispositivos de IO e gráficos, estruturas de dados, cálculos de complexos de física e outros componentes conforme ilustrado na figura 2.

Figura 2: SDK e middleware

Fonte: Game engine architecture (2011, Página 29)

2.1.2 Camada de Independência

A maioria das *game engines* necessitam ser executadas em diferentes plataformas, tanto em hardware como em software, para que isso seja possível é adicionada uma camada de independência, conforme ilustrada na figura 3. Esta camada garante que a engine tenha o mesmo comportamento em todas as plataformas.

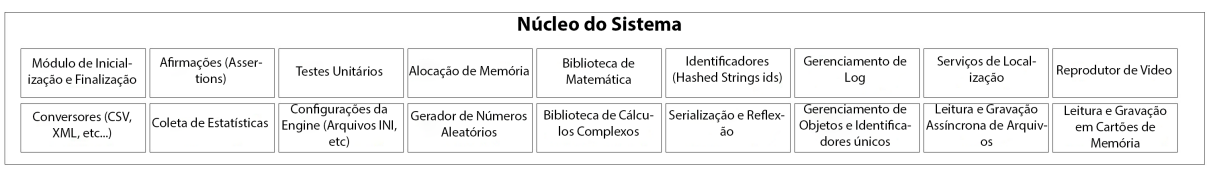
Figura 3: Camada de independência

Fonte: Game engine architecture (2011, Página 29)

2.1.3 Núcleo do Sistema

As *game engines*, assim com qualquer outro software, necessitam de um conjunto de ferramentas que sirvam de base para as outras partes do sistema, nas *game engines* estas ferramentas ficam localizadas no núcleo do sistema tambem conhecido como *Core Engine*. No núcleo são definidas as estruturas de dados que não são fornecidas pelos *middlewares*, algoritmos de gerenciamento de memória customizados, bibliotecas que efetuam cálculos matemáticos complexos e algoritmos que auxiliam no *debug* e *log* da aplicação.
O núcleo também pode ser responsável pela inicialização e finalização de outros subsistemas, controlar o loop principal de eventos e sincronização de processos adicionais. A figura 4 ilustra alguns componentes presentes nesta camada.

Figura 4: Núcleo do Sistema

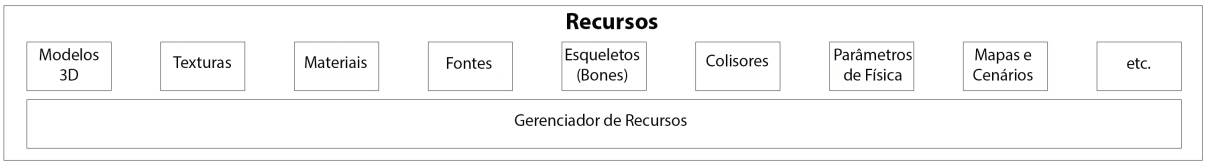


Fonte: Game engine architecture (2011, Página 29)

2.1.4 Gerenciador de Recursos

Toda *game engine* inclui um sistema de gerenciamento de recursos (ilustrado na figura 5) que é utilizado para padronizar a forma de acesso aos dados externos que necessitam ser incluídos no jogo (com modelos, texturas, *shader*, etc) e controlar o carregamento, liberação e reutilização desses recursos durante a execução.

Figura 5: Gerenciador de Recursos



Fonte: Game engine architecture (2011, Página 29)

2.1.5 Motor de Renderização

O motor renderização (ou *rendering engine*) é a camada da *game engine* responsável pelo acesso e manipulação dos recursos gráficos utilizados pelos jogos. Para a manipulação desses recursos são utilizadas bibliotecas de baixo nível (geralmente implementadas em linguagem C ou C++) que fornecem recursos para a total customização dos aspectos do jogo. O motor de renderização é a maior, mais complexa e mais importante parte da *game engine*, e pode ser dividida em várias camadas.

2.1.5.1 Renderização de Baixo Nível

A renderização de baixo nível (ilustrada na figura 6) é a camada responsável pela renderização de primitivas básicas (geralmente são utilizados triângulos), alocação e configuração de buffers e cálculos de coloração dos pixels. É nesta camada que são feitas as chamadas às bibliotecas que manipulam os dispositivos gráficos como OpenGL e Direct3D.

Figura 6: Motor de Renderização - renderização de baixo nível

Fonte: Game engine architecture (2011, Página 29)

2.1.5.2 OpenGL e Direct3D

OpenGL é uma biblioteca desenvolvida em C que fornece acesso aos recursos gráficos de vários tipos de dispositivos, podendo ser implementado em hardware (como placas de video) ou em software caso não exista um hardware gráfico. (SHREINER, 2013) OpenGL é uma das bibliotecas mais utilizadas no desenvolvimento de jogos pelo fato de ser portável para várias plataformas e por prover acesso e controle de baixo nível ao hardware mesmo em linguagens de nível mais alto como Java e Python.
Segundo Sellers (2011) esse nível de acesso ao hardware da uma vantagem muito grande a OpenGL em relação a outras API pois permite não só re-implementar as técnicas de renderização já utilizadas, mas também fornece liberdade para que o desenvolvedor teste e invente novas técnicas. OpenGL possui uma sintaxe de programação própria chamada GLSL (*OpenGL Shading Language*) para a produção e controle de efeitos e renderização de imagens.
Direct3D é uma parte da API DirectX voltada para a renderização de gráficos 2D e 3D nas plataformas da Microsoft. Apesar de também fornecer acesso de baixo nível aos recursos de hardware, Direct3D é fortemente orientado a objetos, o que torna o desenvolvimento e a integração com linguagens de alto nível um pouco mais simples. Assim como OpenGL, Direct3D também possui a sua própria sintaxe de programação chamada HLSL (*High Level Shading Language*) e é a linguagem mais utilizada dentro das plataformas da Microsoft devido ao seu desempenho otimizado. Ao contrario de OpenGL, Direct3D contem embutido em sua biblioteca um mecanismo nativo de criação de janelas e captura de inputs chamado XInput e não é portável para outras plataformas.
Tanto OpenGL quando Direct3D fornecem duas formas básicas de renderização, através de um *pipeline* fixo, também conhecido como *Legacy Mode* (Modo Legado) e um *pipeline* programável utilizando *shaders*.

- *Pipeline* fixo: Consiste em uma serie de etapas necessárias para que o dispositivo gráfico consiga renderizar as imagens desejadas. Estas etapas consistem na especificação dos vértices e primitivas, transformações geométricas, iluminação, rasterização e renderização. Por se tratar de uma rotina fixa, a quantidade de efeitos que podem ser aplicados na imagem final é bastante limitada.

- *Pipeline* programável: Consiste no mesmo processo que o *pipeline* fixo, mas permite um controle mais fino por parte do programador, já que permite alterações em sua saída através de *shaders* específicos.

Shaders são programas que são executados internamente nos dispositivos gráficos para manipular os resultados de saída do *pipeline* de renderização. Os *shader* são divididos em três grupos:

- *Vertex Shader*: O *Vertex Shader* tem por responsabilidade substituir o processo de manipulação dos vértices presente no *pipeline* fixo, permitindo ao programador manipular as transformações geométricas, o posicionamento e cor dos vértices, bem como o processo de iluminação e geração de coordenas de textura na parte dos vértices.

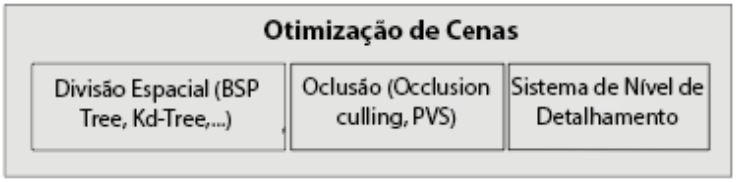
- *Geometry Shader*: O *Geometry Shader* é um recurso opcional no *pipeline* programável. Sua principal função é a de criar ou remover vértices, permitindo um maior controle na geometria que passa pelo *pipeline* programável, aumentando ou diminuindo o nível de detalhe conforme necessário, permitindo com que uma enorme variedade de efeitos sejam possíveis, como melhor interpolação de vértices com a adição de novos vértices quando necessário, renderização de silhueta, entre outros.

- *Fragment Shader* (ou *Pixel Shader*): O *Fragment Shader* (ou *Pixel Shader*) tem a função de manipular a coloração dos pixels, calcular a iluminação e manipular as texturas.

2.1.5.3 Otimização de Cenas

A camada de Otimização (ilustrada na figura 7) é a camada responsável pela remoção de objetos que estão muito distantes, localizados atras da câmera ou escondidos atras de outros objetos da lista de renderização, podendo aumentar drasticamente a performance da aplicação.

Figura 7: Motor de Renderização - Scene Graph e Culling Optimization

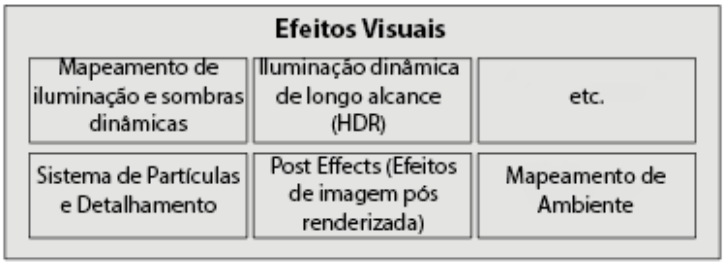


Fonte: Game engine architecture (2011, Página 29)

2.1.5.4 Efeitos Visuais

As *game engines* mais modernas suportam um grande numero de efeitos visuais (conforme ilustrado na figura 8). Esses efeitos variam de acordo com o tipo de jogo e a plataforma alvo, geralmente é nesta camada onde são realizados os cálculos mais complexos e pesados da engine.

Figura 8: Motor de Renderização - Efeitos Visuais

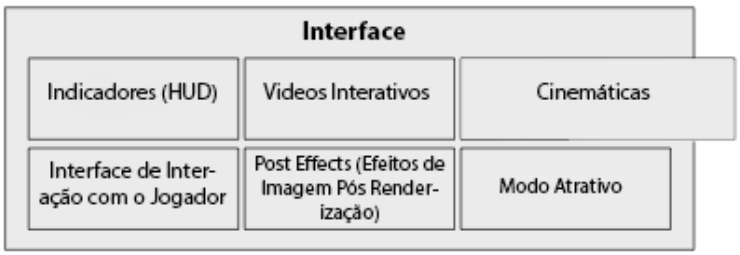


Fonte: Game engine architecture (2011, Página 29)

2.1.5.5 Interação com o Usuário

A maioria dos jogos necessitam de uma interface 2D para a interação com o usuário, seja em um menu de opções ou para exibir a pontuação. A camada de interação com o usuário (ilustrada na figura 9) é responsável por apresentar essas informações independente da técnica de renderização utilizada.

Figura 9: Motor de Renderização - Interface com o Usuário

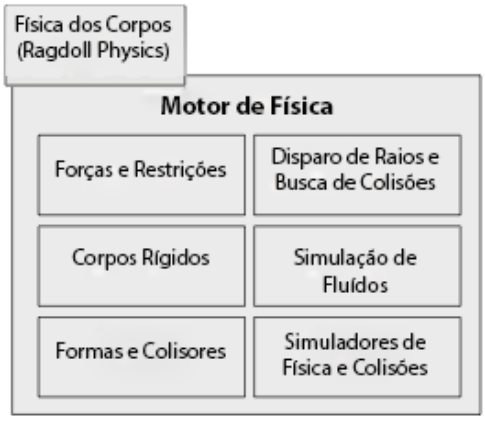


Fonte: Game engine architecture (2011, Página 29)

2.1.6 Motor de Física

O motor de física é uma das camadas indispensáveis em uma *game engine*, pois é ela que provê um comportamento realista ao jogo. A maioria dos motores de física consiste em um mecanismo detector de colisões, um sistema de simulação de corpos rígidos e fluidos, podendo eles serem baseados nas regras do mundo real ou não. A figura 10 mostra alguns componentes presentes nesta camada.

Figura 10: Motor de Física



Fonte: Game engine architecture, 2011

2.1.7 Dispositivos de Interação Humana

A camada de dispositivos de interação humana (ilustrado na figura 11) é responsável por capturar, processar e redirecionar as entradas do usuário para as demais áreas da engine. Estas entradas podem ser feitas através do teclado, mouse, *joystick* ou qualquer outro dispositivo conectado ao hardware onde será executado o jogo.

Figura 11: Dispositivos de Interação Humana

Fonte: Game engine architecture (2011, Página 29)

2.1.8 Motor de Áudio

O motor de áudio (figura 12) é a camada responsável por acessar e manipular os recurso de áudio utilizados pelos jogos. Geralmente são utilizadas bibliotecas de baixo nível desenvolvidas em C que são capazes se simular áudios em 2D e 3D. AS API‘s mais utilizadas são OpenAL e DirectX Audio.

Figura 12: Motor de Áudio

Fonte: Game engine architecture (2011, Página 29)

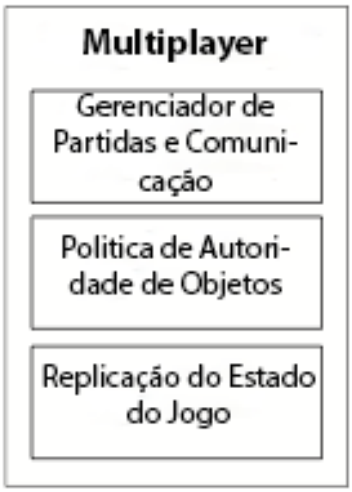
2.1.8.1 OpenAL e DirectX Audio

OpenAL é uma biblioteca de audio multi-plataforma que possibilita a reprodução de audio em três dimensões, o que a torna ideal para varias aplicações de audio, principalmente para jogos. (HIEBERT, 2007)
DirectX Audio é uma parte da API DirectX que combina duas bibliotecas DirectSound e DirectMusic, onde uma é utilizada para efeitos sonoros e a outra para reprodução de áudio em alta qualidade.

2.1.9 Motor de Comunicação em Rede

O motor de comunicação em rede (ilustrado na figura 13) é uma camada opcional que permite a troca de informações entre duas instâncias do mesmo jogo. Devido a complexidade de implementar a troca de informações em alta velocidade sem afetar performance dos jogos, esta camada é implementada somente em jogos que necessitam desse tipo de comunicação.

Figura 13: Motor de Comunicação em Rede

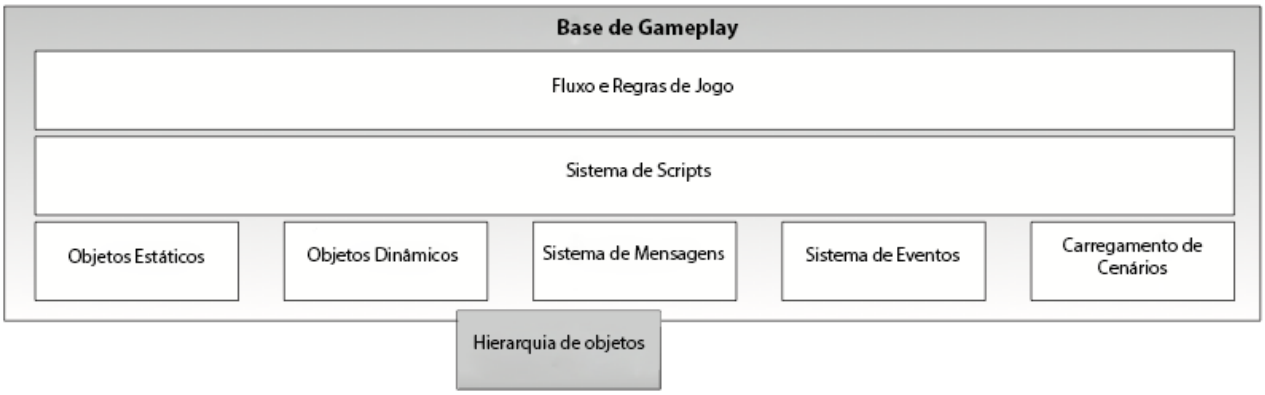


Fonte: Game engine architecture (2011, Página 29)

2.1.10 Base de Gameplay

A Base de gameplay (ilustrada na figura 14) é uma camada responsável pela lógica executada em tempo de jogo e base para a criação de objetos. A organização desses objetos varia de acordo com a necessidade, mas na maioria dos casos os objetos são organizados em cenas, onde cada cena pode ser uma fase ou uma sala especifica dentro do jogo. A hierarquia dos objetos dentro da cena pode ser feita de duas maneiras, através de um sistema de herança ou baseado em componentes.

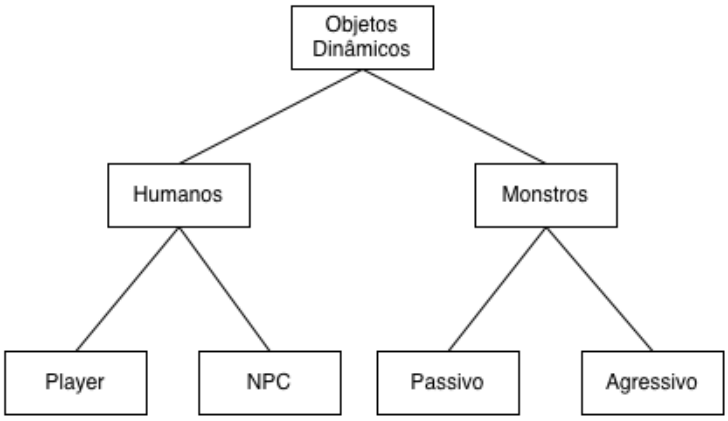
Figura 14: Base de Gameplay



Fonte: Game engine architecture (2011, Página 29)

- Sistema baseado em herança: É criada uma herança de classes, onde cada objeto tem o seu comportamento definido pelas funções implementadas nele e em seus predecessores. (figura 15)

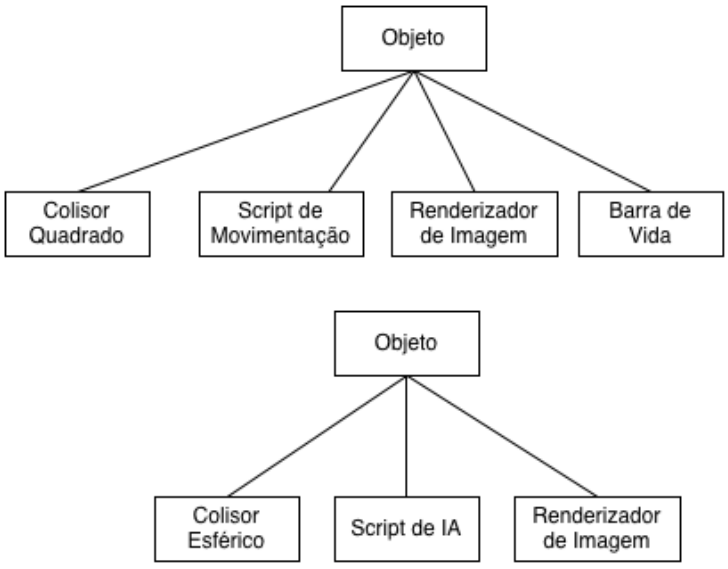
Figura 15: Base de Gameplay - arquitetura baseada em herança



Fonte: Elaborado pelo autor

- Sistema baseado em componentes: Cada objeto funciona como um conteiner, onde os componentes adicionados a ele definem o seu comportamento e atributos.(figura 16)

Figura 16: Base de Gameplay - arquitetura baseada em componentes



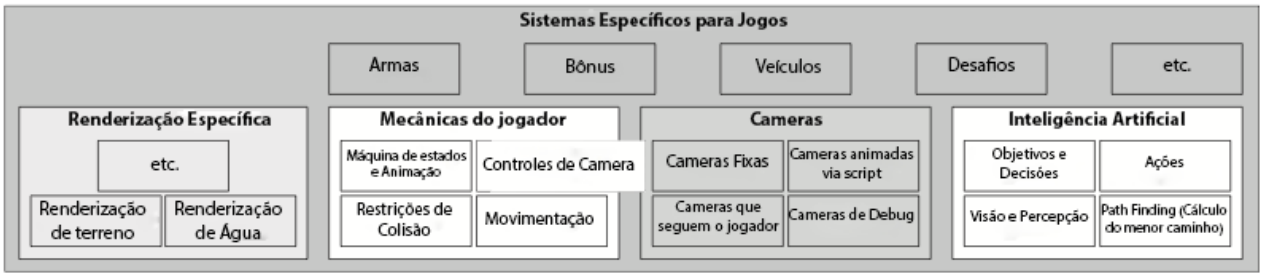
Fonte: Elaborado pelo autor

Geralmente esta camada inclui um motor de manipulação de *scripts* que funciona como uma maquina virtual que interpreta *scripts* desenvolvidos em uma linguagem diferente do resto da *engine*, a fim de oferecer flexibilidade, facilidade e velocidade de execução. Praticamente qualquer linguagem pode ser implementada na *scripting engine*, as mais utilizadas são LUA, JavaScript, Python e C#.

2.1.11 Sistemas Específicos para Jogos

O objetivo principal de uma *game engine* é fornecer um conjunto básico de funcionalidades que são comuns na maioria dos jogos, acima de todas as camadas anteriores são construídos os sistemas únicos para cada jogo. Nesta camada são definidos os modelos, as texturas e toda a lógica implementada para um jogo, conforme ilustrado na figura 17.

Figura 17: Sistemas Específicos para Jogos



Fonte: Game engine architecture (2011, Página 29)

Com a arquitetura em camadas é possível adicionar e remover funcionalidades sem que seja necessário rescrever todo o software. Para que esta arquitetura funcione cada camada deve ter apenas uma função e deve ser ligada a uma das camadas essenciais para o funcionamento da *engine*. Uma *game engine* pode ser implementada com praticamente qualquer linguagem, tendo ela acesso nativo ao hardware ou não, caso a linguagem não tenha acesso ao hardware, o controle pode ser feito via software (geralmente fazendo chamadas ao sistema operacional ou a algum outro software que tenha acesso direto), impossibilitando o controle completo das camadas de nível mais baixo por parte do desenvolvedor.
No capítulo 3 estão descritas as ferramentas e técnicas utilizadas no desenvolvimento da Pulsar Game Engine.

3 Pulsar Game Engine

3.1 Projeto

As tecnologias utilizadas nesse projeto foram escolhidas visando a maior portabilidade possível sem que fosse necessário recompilar a *engine*, para isso foram escolhidas as linguagens Java e JavaScript para o desenvolvimento do núcleo e dos *scripts* executados durante o jogo. Devido a complexidade do projeto algumas bibliotecas auxiliares foram escolhidas para agilizar o desenvolvimento, para controle sobre o hardware gráfico e som foi escolhido uma adaptação das bibliotecas OpenGL e OpenAL para Java chamada LWJGL (*Lightweight Java Game Library*), para as simulações de física em duas dimensões foi escolhida a biblioteca Box2D e para compilação e gerenciamento das dependencias foi utilizado Gradle.
A *game engine* recebeu o nome de Pulsar Game Engine e seu código fonte está disponível no link *https://github.com/Netoaoh/pulsar-game-engine* sob a licença GPL-3.0.

3.2 Arquitetura

A Pulsar Game Engine possui uma arquitetura baseada em *Game Objects* e *Game Components*, onde os *Game Objects* funcionam como containers e os *Game Components* definem as propriedades e o comportamento desses objetos dentro do jogo, com isso é possível criar uma variação grande de objetos utilizando o mesmo código.
Para facilitar a extensão e customização da *engine*, todos os sub-sistemas são implementados em cima de classes abstratas, que definem os métodos principais para o funcionamento correto da *engine*. Essas classes possuem o prefixo "I"seguido do nome do sub-sistema (figura 18) e são instanciados e registrados antes da inicialização do *loop* principal (figura 19), desta forma é possivel alterar o comportamento da engine sem que seja necessário alterar a biblioteca principal.

Figura 18: Classe base de sub-sistema

```
public abstract class IAudioEngine {
    protected static IAudioEngine instance;

    protected IAudioEngine() { instance = this; }

    public abstract void initialize();
    public abstract void shutdown();

    public abstract Vector3f getListenerPos();
    public abstract void setListenerPos(Vector3f listenerPos);
}
```

Fonte: Elaborado pelo autor

Figura 19: Código da classe principal

```
public static void main(String[] args){

    // Inicializa LWJGL library
    setupLwjgl();

    // Cria uma instancia da engine principal
    Engine engine = new Engine();

    // Configura sub engines
    engine.setRenderingEngine(new RenderingEngine());
    engine.setAudioEngine(new AudioEngine());
    engine.setPhysicsEngine(new PhysicsEngine());
    engine.setScriptEngine(new JSEngine());


    // Adiciona cenas ao jogo
    engine.getGameInstance().addScene(new DemoScene());
    engine.getGameInstance().loadScene( name: "Demo Scene");

    // Inicia o Game Loop da engine
    engine.run( gameName: "Demo Game",  width: 800,  height: 600,  fullscreen: false);

    // Finaliza o processo
    System.exit(EXIT_WITHOUT_ERROR);
}
```

Fonte: Elaborado pelo autor

3.3 Componentes

Componentes ou *Game Components* são classes que definem as propriedades e o comportamento dos objetos em jogo, cada *Game Object* pode conter um ou mais componentes.
Cada Componente possui um ou mais metodos que são executados em momentos específicos do fluxo de execução da *engine*. Esses métodos são:

- *Awake*: Executado quando o componente é adicionado ao *Game Object*.
- *Start*: Executado quando a cena é carregada e exibida.
- *Update*: Executado a cada iteração do fluxo de execução.

- *FixedUpdate*: Executado a cada iteração da engine de física (30 vezes por segundo).
- *Render*: Executado 60 vezes por segundo.
- *OnCollisionEnter*: Executado quando há uma colisão entre dois ou mais objetos.

Os componentes disponíveis por padrão na Pulsar Game Engine são:

- *AnimationComponent*: Componente utilizado para controlar animações de *sprites*.
- *AudioListenerComponent*: Componente utilizado como receptor de som dentro da *engine*.
- *AudioSourceComponent*: Componente utilizado como fonte de som dentro da *engine*.
- *BoxCollider*: Componente que define e controla a caixa de colisão de um objeto.
- *Camera*: Componente que define a posição e as propriedade de visão dentro da *engine*.
- *ScriptComponent*: Componente que provê controle sobre a lógica de um objeto em tempo de execução.
- *SpriteRenderer*: Componente que possibilita a renderização de um *sprite*.
- *Transform*: Componente que armazena as informações de posição, rotação e escala de um objeto (este componente é padrão e obrigatório para todos os objetos em cena).

Novos componentes podem ser criados e integrados a *engine* estendendo a classe *Component*.

3.4 Lógica em tempo de jogo

Para controlar a lógica executada durante o jogo a Pulsar Game Engine possui um sistema de *scripts* que interpreta a linguagem JavaScript em tempo de execução utilizando o motor ja imbutido na linguagem Java.
O sistema de script expõe os métodos *Awake*, *Start*, *Update*, *FixedUpdate* e *OnCollisionEnter* para serem manipulados no *script*.

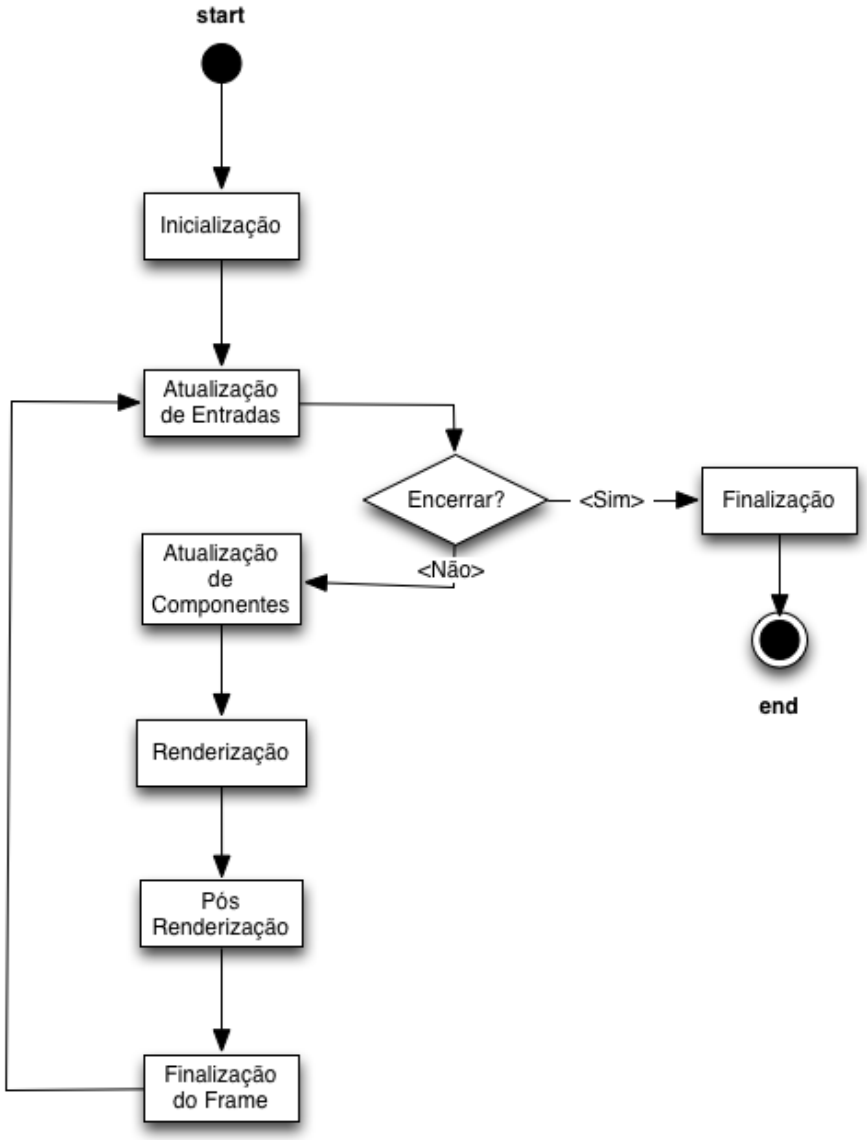
3.5 Técnicas de Renderização

Com a utilização dos *shaders* é possível implementar diversas técnicas de renderização, para este projeto foi escolhida a técnica *multi-pass forward rendering*, que consiste em renderizar os objetos várias vezes aplicando diferentes *shaders* (um shader para cada efeito ou ponto de luz presente na cena) e posteriormente combinando as imagens utilizando a técnica de mesclagem aditiva (*Additive Blending*) para formar a imagem final. Esta técnica foi escolhida devido a facilidade de implementação e a performance superior em cenas simples comparado a outras técnicas.

3.6 Fluxo de Execução

Esta seção apresenta o fluxo de execução da engine em desenvolvimento (figura 20)

Figura 20: Loop principal



Fonte: Elaborado pelo autor

3.6.1 Inicialização

Na inicialização do sistema são carregadas todas as configurações e inicializados todos os subsistemas necessários para que o jogo funcione corretamente, é nesta etapa que são criados os *framebuffers* e carregadas as definições das cenas.

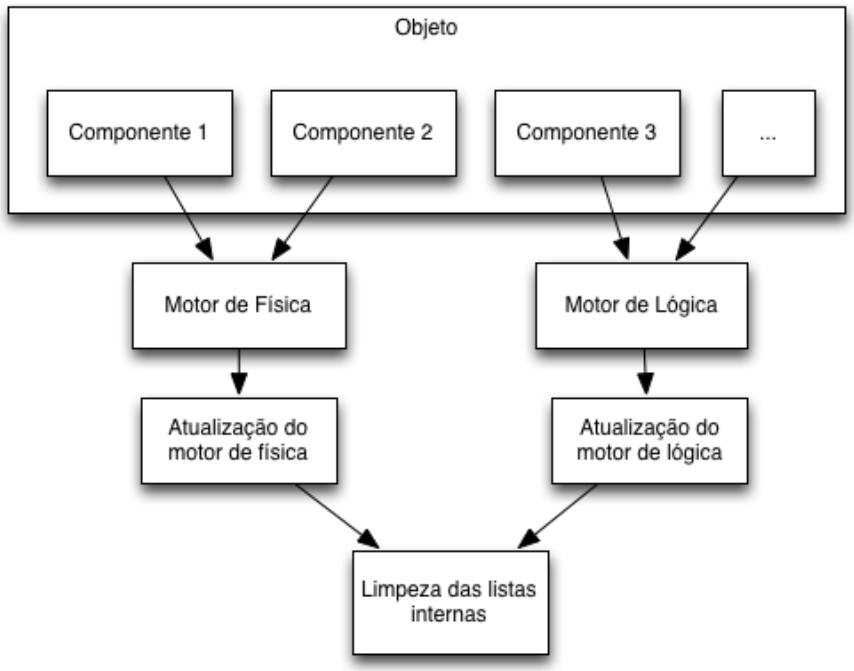
3.6.2 Atualização de entradas

Nesta etapa do loop principal são capturadas e armazenadas as entradas do usuário para serem utilizadas por qualquer componente que necessite dessas informações. Nesta etapa também é feita uma verificação para que o jogo seja finalizado ou não, saindo assim do loop infinito.

3.6.3 Atualização de componentes

Nessa etapa todos os componentes de todos os objetos instanciados na cena são atualizados, conforme ilustrado na figura 21. Cada componente é enviado e armazenado temporariamente no subsistema que é responsável pela sua execução, este subsistema realiza a atualização dos componentes e limpa a sua lista interna de atualização.
Por questões de performance os componentes de renderização são agrupados em lotes (*batches*) que são enviados para o motor de renderização, onde serão renderizados de acordo com esses grupos, conforme descrito na seção 3.6.4.

Figura 21: Ciclo de atualização dos componentes

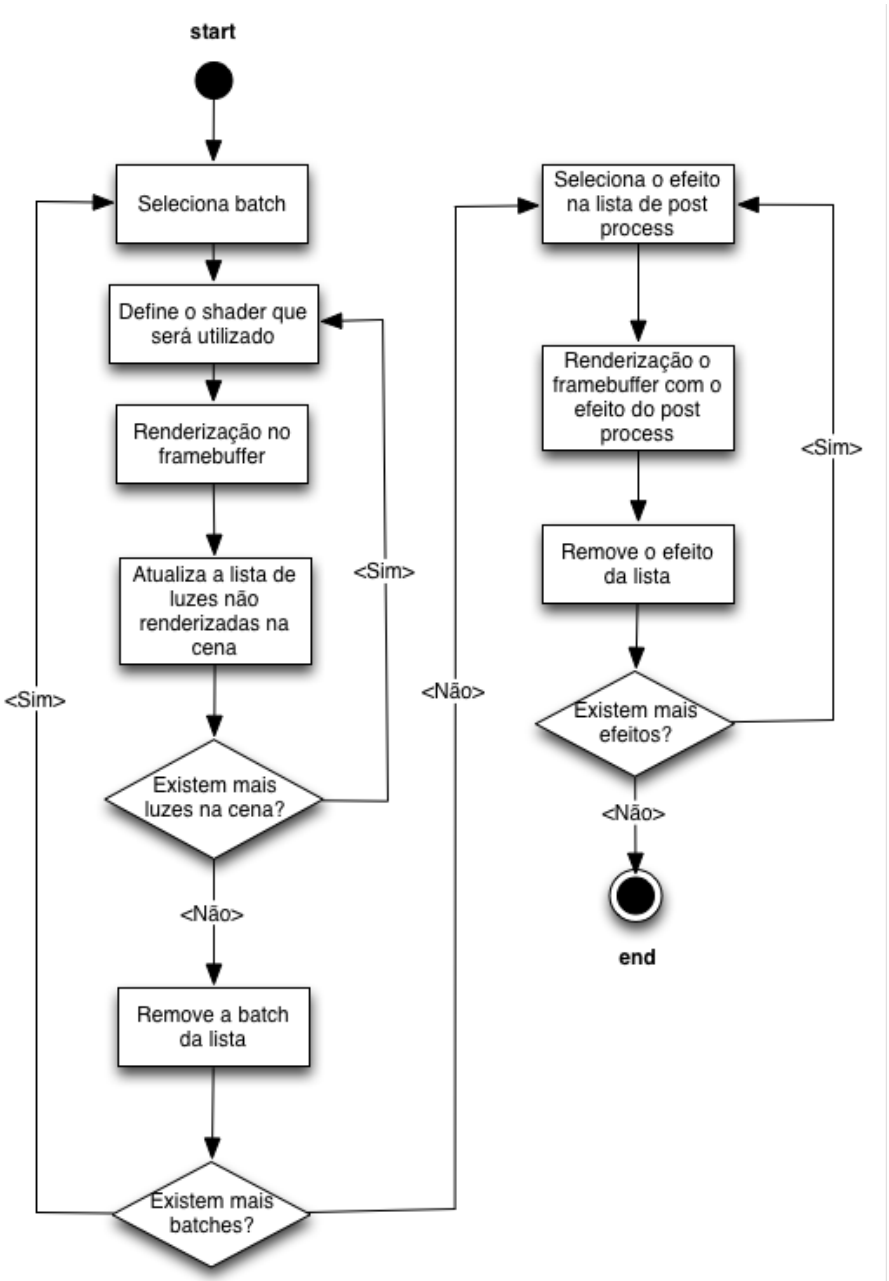


Fonte: Elaborado pelo autor

3.6.4 Renderização

Na etapa de renderização os dados dos objetos agrupados são enviados para os *buffers* da placa de video e renderizados na tela conforme ilustrado na figura 22.

Figura 22: Ciclo de renderização dos componentes



Fonte: Elaborado pelo autor

3.6.5 Pós Renderização

A etapa de pós renderização é utilizada para atualizações que necessitam ser feitas após os objetos serem renderizados na tela (como a movimentação da câmera por exemplo), afim de prevenir erros de renderização.

3.6.6 Finalização do Frame

Etapa que finaliza o frame limpado os *buffers* de renderização da janela e liberando recursos que não serão reutilizados no próximo frame (como objetos e componentes removidos da cena).

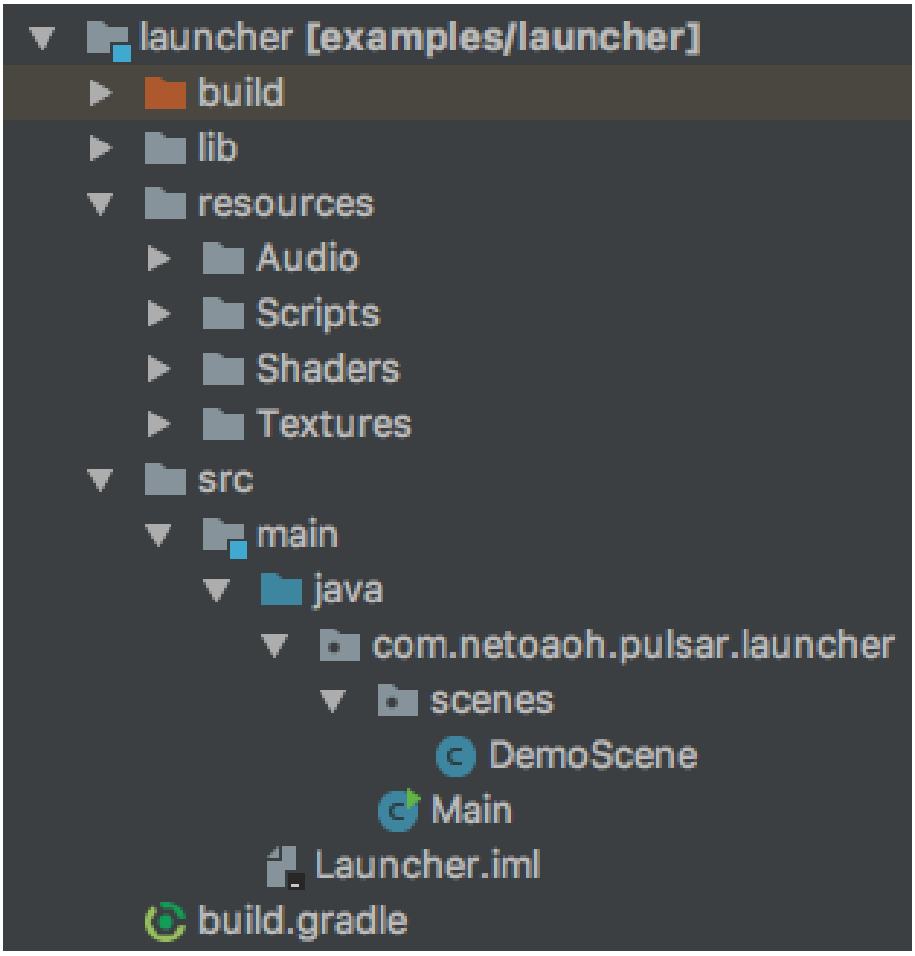
3.6.7 Finalização da Engine

Caso seja enviado um comando de finalização todos os recursos alocados pela engine são liberados, garantindo a finalização do programa de maneira segura e sem desperdício de memória (*memory leaks*).

3.7 Utilização

A Pulsar Game Engine foi desenvolvida utilizando a ferramenta *Gradle* para compilar e gerenciar as dependencias do projeto, desta forma para criar um novo jogo utilizando a *engine*, basta utilizar o template disponível nos exemplos do repositório (figura 23), definir os objetos iniciais de cada cena disponível no jogo (figura 24) e caso necessário estender ou reimplementar os sistemas internos da *engine*. Toda a lógica executada durante o jogo deve ser implementada em *scripts* utilizando a linguagem JavaScript.

Figura 23: Estrutura de pastas de um projeto utilizando a Pulsar Game Engine



Fonte: Elaborado pelo autor

Figura 24: Definição de objetos em uma cena

```
public class DemoScene extends IScene {

    public DemoScene() { super( sceneName: "Demo Scene"); }

    public void start() {

        GameObject cameraObject = new GameObject( name: "Camera");
        cameraObject.addComponent(new Camera(new Matrix4f().initOrthographic( left: -80.0f,  right: 80.0f,
                bottom: -60.0f,  top: 60.0f,  near: -1,  far: 100)));
        cameraObject.addComponent(new ScriptComponent( filename: "camera.js"));
        cameraObject.addComponent(new AudioListenerComponent());
        addToScene(cameraObject);

        GameObject obj = new GameObject( name: "player");
        obj.addComponent(new SpriteRenderer(new Sprite( width: 16, height: 16),
                new Material( name: "Default", new Texture( filename: "Charizard.png",  numberOfRows: 4))
        ));
        obj.addComponent(new AudioSourceComponent( filename: "bounce.wav"));
        obj.addComponent(new ScriptComponent( filename: "player.js"));
        obj.addComponent(new BoxCollider( width: 20,  height: 20));

        SpriteAnimation idleAnim = new SpriteAnimation( name: "idle");
        idleAnim.addFrame( index: 0,  timestep: 0.25f);

        SpriteAnimation walkRightAnim = new SpriteAnimation( name: "left");
        walkRightAnim.addFrame( index: 4,  timestep: 0.25f);
        walkRightAnim.addFrame( index: 5,  timestep: 0.25f);
        walkRightAnim.addFrame( index: 6,  timestep: 0.25f);
        walkRightAnim.addFrame( index: 7,  timestep: 0.25f);

        SpriteAnimation walkLeftAnim = new SpriteAnimation( name: "right");
        walkLeftAnim.addFrame( index: 8,  timestep: 0.25f);
        walkLeftAnim.addFrame( index: 9,  timestep: 0.25f);
        walkLeftAnim.addFrame( index: 10,  timestep: 0.25f);
        walkLeftAnim.addFrame( index: 11,  timestep: 0.25f);

        AnimationComponent anim = new AnimationComponent();
        anim.addAnimation( name: "idle", idleAnim);
        anim.addAnimation( name: "left", walkLeftAnim);
        anim.addAnimation( name: "right", walkRightAnim);
        anim.setAnimation("idle");
        obj.addComponent(anim);
        addToScene(obj);
```

Fonte: Elaborado pelo autor

4 Conclusão

O Desenvolvimento de uma *game engine* é uma tarefa difícil e trabalhosa que traz muitos benefícios pelo fato de serem implementações abstratas que servem de base para vários jogos, diminuindo assim o custo e o tempo de produção. O fato das *game engines* serem organizadas em camadas facilita a visualização de dependências e customização de seus componentes internos, desta forma pode-se escolher quais camadas implementar de acordo com o objetivo final. A Pulsar Game Engine implementa os componentes básicos necessários para a criação de jogos e uma arquitetura de objetos baseada em componentes, desta forma a customização da *engine* é feita de maneira simples, adicionando componentes e alterando o seu funcionamento conforme a necessidade, resultando em um software polido e enxuto.

Apesar de se tratar de uma engine simples, a Pulsar Game Engine consegue produzir jogos de genero variado sem a necessidade de conhecimentos avançados de programação e computação gráfica.

Referências

BRITO, A. *Blender 3D: jogos e animações interativas*. São Paulo: Novatec, 2011. Citado na página 23.

EBERLY, D. H. *3D game engine design: a practical approach to real-time computer graphics*. São Francisco: Morgan Kaufmann, 2001. Citado na página 23.

GREGORY, J. *Game engine architecture*. Wellesley, Massachusetts: A. K. Peters, Ltd, 2011. Citado na página 23.

HIEBERT, G. *OpenAL Programmer's Guide*. [S.l.: s.n.], 2007. Citado na página 31.

SELLERS, G. *OpenGL SuperBible 5th Edition: Comprehensive Tutorial and Reference*. New Jersey: Addison-Wesley, 2011. Citado na página 27.

SHREINER, D. S. *OpenGL Programming Guide 8th Edition: The Official Guide to Learning OpenGL*. New Jersey: Addison-Wesley, 2013. Citado na página 27.