**GUIA DE DESENVOLVIMENTO**



**BPM Game Engine**

**Versão 1.0**

Fabio Takeshi Ishikawa

Janeiro de 2020

Sumário

[**1. Introdução** 4](#_Toc35541177)

[**1.1 O que é BPM Game Engine** 4](#_Toc35541178)

[**2. Visão Geral da Arquitetura** 5](#_Toc35541179)

[**2.1 Camada de Hardware** 5](#_Toc35541180)

[**2.2 Camada de Sistema Operacionais** 6](#_Toc35541181)

[**2.3 Camada API de Terceiros** 6](#_Toc35541182)

[**2.4 Camada Multiplataforma** 6](#_Toc35541183)

[**2.5 Camada Core** 7](#_Toc35541184)

[**2.5.1 Loop principal** 7](#_Toc35541185)

[**2.5.2 Controlando o tempo** 7](#_Toc35541186)

[**2.5.3 Janelas** 7](#_Toc35541187)

[**2.5.4 Eventos de Usuários** 7](#_Toc35541188)

[**3. Configuração do Ambiente de Desenvolvimento** 8](#_Toc35541189)

[4. Dependências 8](#_Toc35541190)

[5. Debugando seu código 8](#_Toc35541191)

[6. Classes e Objetos Globais 8](#_Toc35541192)

[6.1 Classe Game Engine 8](#_Toc35541193)

[6.1.1 Atributos 8](#_Toc35541194)

[6.1.2 Métodos 8](#_Toc35541195)

[6.2 Classe Window 9](#_Toc35541196)

[6.2.1 Atributos 9](#_Toc35541197)

[6.2.2 Métodos 9](#_Toc35541198)

[6.2 Classe ApiWrapper (classe abstrata) 10](#_Toc35541199)

[6.2.1 Atributos 10](#_Toc35541200)

[6.2.2 Métodos 10](#_Toc35541201)

[6.3 Classe WinApi Wrapper (herdade de ApiWrapper) 10](#_Toc35541202)

[6.3.1 Atributos 10](#_Toc35541203)

[6.3.2 Métodos 10](#_Toc35541204)

[6.4 Classe GameEngineEventHandler (classe abstrata) 10](#_Toc35541205)

[6.5 Classe Render 10](#_Toc35541206)

[6.6 Classe IO 10](#_Toc35541207)

[6.7 Classe Mouse 10](#_Toc35541208)

[6.8 Classe Teclado 11](#_Toc35541209)

[6.9 Classe AssetManager 11](#_Toc35541210)

[7. Personalização 11](#_Toc35541211)

[8. Convenção de Código 11](#_Toc35541212)

# **1. Introdução**

Este documento tem como objetivo descrever em detalhes a arquitetura de BPM Game Engine. São abordadas também as ferramentas e as linguagens de programação utilizadas para o seu desenvolvimento, assim como todas as classes e os seus relacionamentos que fazem parte deste motor de jogo. Abordamos também conceitos e práticas de engenharia de software aplicadas neste projeto como o gerenciamento de configuração, convenção de código, planejamento e execução de testes, ferramentas de depuração de código, fluxo de trabalho com *Git* (sistema de controle de versão) e integração contínua com *Travis CI*.

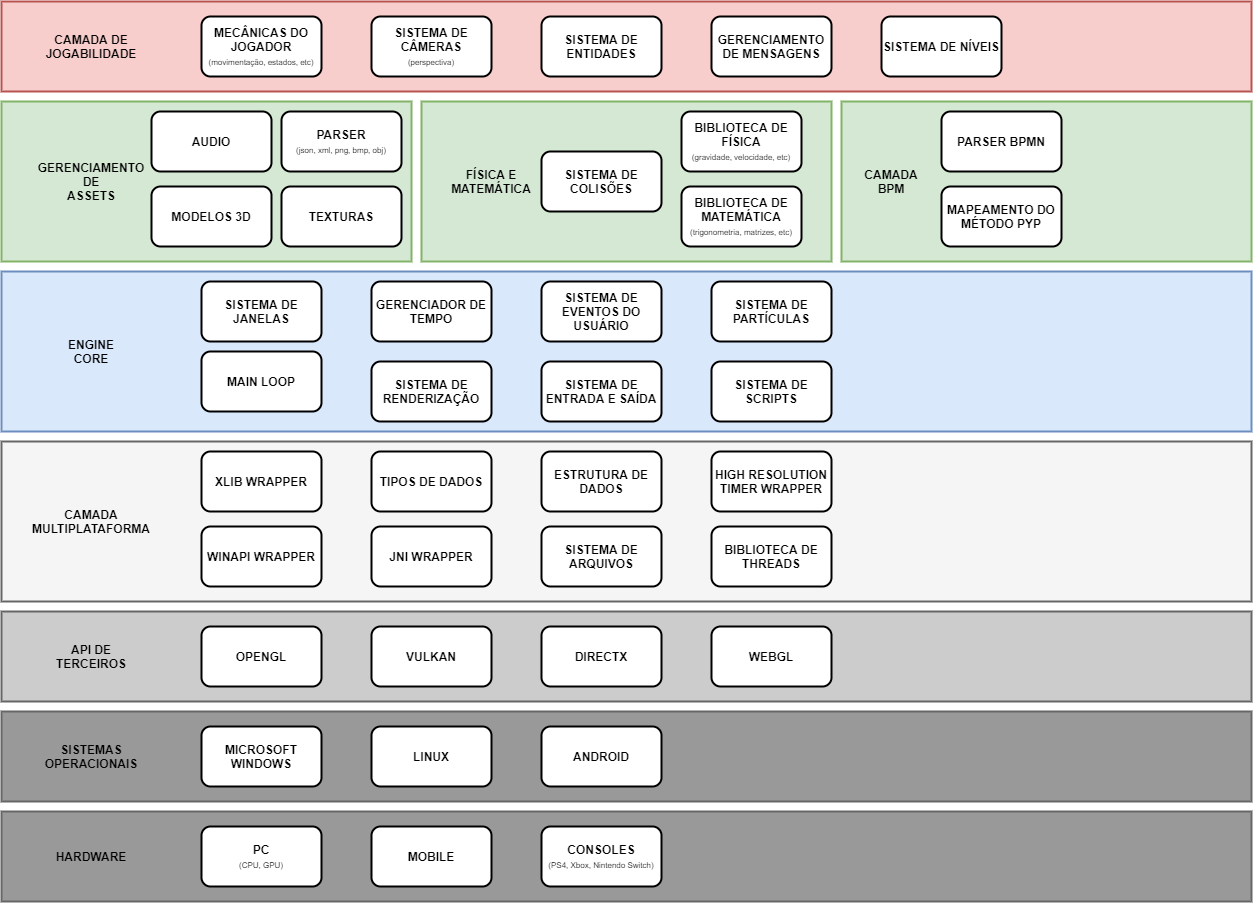
## **1.1 O que é BPM Game Engine**

BPM Game Engine é um motor de jogo que surgiu para apoiar a criação e desenvolvimento de jogos digitais baseados em modelos de processos de negócio. Este novo gênero de jogo surgiu a partir da pesquisa de doutorado “Play Your Process (PYP) – Um método de Design de Jogos Digitais Baseados em Modelos de Processos de Negócio”, onde é apresentado um método que permite transformar elementos de modelos de processos de negócio em elementos de design de jogos. [detalhar mais sobre os benefícios dos jogos deste gênero].

Além do suporte completo para o desenvolvimento de jogos baseados em modelos de processos de negócio, este motor de jogo também pode ser utilizado para criar jogos de diferentes gêneros e propósitos.

# **2. Visão Geral da Arquitetura**

A seguir, segue a visão geral dos componentes que compõe BPM Game Engine. Os componentes estão divididos em diversas camadas que representam o relacionamento e a dependências entre os componentes. Todos os componentes da figura são descritos nesta seção.



*Figura 01. Visão geral dos componentes de BPM Game Engine dividido em camadas.*

## **2.1 Camada de Hardware**

Este projeto nasceu com a ideia de possibilitar o desenvolvimento de jogos digitais para múltiplas plataformas. Portanto toda a sua arquitetura foi projetada considerando as questões que envolvem portabilidade para diversos tipos de hardware como PC (sistemas *multicores* combinados com *GPU*s), dispositivos móveis e consoles (Playstation 4, Xbox One e Nintendo Switch).

Neste projeto há também diversas considerações relacionadas a computação paralela. A evolução do hardware chegou em um nível onde praticamente todos os sistemas computacionais atuais possuem processadores com diversos núcleos e a utilização de múltiplos GPUs. Por exemplo, sistemas como Playstation 4 possui oito núcleos divididos em dois clusters onde cada cluster de quatro núcleos compartilham a mesma memória cache (L2) e todos os núcleos compartilham a mesma memória RAM formando uma arquitetura de memória NUMA (*non uniform memory access*). A utilização de paralelismo é fundamental para a criação de jogos de alto desempenho e está intimamente relacionada com o hardware. (GREGORY, 2018).

## **2.2 Camada de Sistema Operacionais**

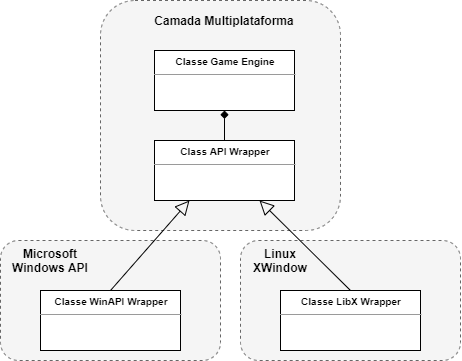
Para suportar diferentes tipos de hardware é necessário lidar também com diferentes tipos de sistemas operacionais. Existem muitos elementos de um software que são dependentes do sistema operacional. Por exemplo, cada sistema operacional possui a sua própria maneira de gerenciar as janelas de aplicações. Microsoft Windows utiliza WinAPI [explicar o que é WinAPi] e o Linux utiliza XWindow [explicar o que é XWindow] para criar as janelas das aplicações. Além disso, cada sistema operacional possui a sua própria biblioteca de threads, sistema de arquivos, gerenciamento de mensagens e assim por diante. O foco deste projeto é permitir a construção de jogos para os seguintes sistemas operacionais: Microsoft Windows 10, Linux, Android e, futuramente, para iOS e MacOS.

## **2.3 Camada API de Terceiros**

O projeto utiliza APIs (*Application Program Interface*) de terceiros para acessar o componente gráfico (GPU) do sistema. OpenGL é uma API para o hardware gráfico que permite a renderização de modelos tridimensionais em tempo real. Ela é extremamente popular e, além de tudo, é portável. Existe também a possibilidade de integrar DirectX e Vulkan em versões futuras de BPM Game Engine.

## **2.4 Camada Multiplataforma**

A camada multiplataforma é formada por um conjunto de componentes que encapsulam os objetos e funções específicas dos sistemas operacionais e do hardware (por exemplo, registradores especiais), fornecendo assim, funcionalidades para o motor de jogo de forma independente de plataforma. A ideia é utilizar o polimorfismo (conceito de programação orientado a objetos) para manipular esses componentes específicos de forma independente.



*Diagrama de classes para representar os componentes multiplataforma.*

Outro componente essencial desta camada é a classe TimeHandler responsável por calcular o tempo de duração de cada loop do motor de jogo em microssegundos (µ). De fato, existem diversas formas de se obter o tempo percorrido desde o momento em que a máquina foi ligada como, por exemplo, time() da biblioteca padrão de C que retorna o tempo percorrido em segundos ou GetTickCount() da WinAPI que retorna em milissegundos. Entretanto, para o nosso motor de jogo, utilizamos uma outra forma de obter o tempo com a maior resolução possível: tempo em ciclos da CPU (unidade de processamento central). Nos processadores modernos, existe um registrador especial chamado de *High Resolution Timer* que é basicamente um contador de ciclos da CPU desde o momento em que foi ligado. Com ele, conseguimos facilmente realizar uma conversão de ciclos por segundo para microssegundos (µ).

Demonstração do cálculo da duração do quadro para Microsoft Windows:

// loop do motor de jogo

while(isDone)

{

// início do loop

QueryPerformanceCounter(&startTime); // obtém o tempo do registrador

...

// fim do loop

QueryPerfomanceCounter(&endTime); // obtém o tempo novamente

frameTime.QuadPart = endTime.QuadPart – startTime.QuadPart; // tempo do quadro em ciclos

// conversão para microssegundos

QueryPerformanceFrequency(&frequency);

microSeconds = frameTime.QuadPart \* 1000000 / frequency.QuadPart;

}

A ideia da conversão de ciclos para microssegundos é multiplicar primeiro o número de ciclos por 1000000 microsegundos (equivalente a 1 segundo) e depois dividir pela frequência de ciclos por segundo.

## **2.5 Camada Core**

Nesta camada se concentra os componentes principais do motor de jogo. Pode se dizer que eles formam o coração do motor de jogo.

### **2.5.1 Loop principal**

### **2.5.2 Controlando o tempo**

### **2.5.3 Janelas**

### **2.5.4 Eventos de Usuários**

# **3. Configuração do Ambiente de Desenvolvimento**

# 4. Dependências

# 5. Debugando seu código

# 6. Classes e Objetos Globais

## 6.1 Classe Game Engine

### 6.1.1 Atributos

Objeto APIWrapper (classe abstrata)

Este objeto é responsável por executar métodos específicos do sistema operacional. Para cada sistema operacional, é necessário criar uma classe derivada de APIWrapper e definir os métodos virtuais. O tipo do objeto é definido no momento da compilação através de uma constante de compilação.

Objeto GameEngineEventHandler (classe abstrata)

Este objeto especifica as funções de callback da game engine como, por exemplo, as funções de inicialização, renderização e de finalização. Como padrão, a game engine cria as suas próprias funções de callback vazias. Elas devem ser configuradas com funções de callback definidas pelo usuário.

String Name

Especifica o nome do aplicação ou jogo e será utilizada como texto na barra de títulos de uma janela.

### 6.1.2 Métodos

SetGameEngineEvents

Método utilizado para configurar as funções de callback.

mainLoop

Função de execução da game engine. Assim que esse método é chamado, a game engine obtem o controle total da aplicação. Obviamente, ela respeitará as funções de callback do usuário para responder aos eventos conforme o desejo do usuário.

InitGameEngine

Método para inicializar a game engine.

## 6.2 Classe Window

### 6.2.1 Atributos

String titulo

Especifica o texto que será exibido na barra de títulos da janela de aplicação.

Int largura

Especifica a largura da janela em pixels

Int altura

Especifica a altura da janela em pixels

Int estilo

Especifica o estilo da janela. Os estilos possívels para uma janela são: com barra de título, sem barras de titilo, diferentes tipos de botões do sistema, todas as opções, janela redimensionavel, e não redimensionavel, janela maximizada, minimzada)

Int posicaoInicialX

Especifica a posição inicial da janela no eixo X.

Int posicaçãoInicialY

Especifica a posição inicial da janela no eixo Y.

### 6.2.2 Métodos

CreateWindow

Método para criar uma janela de aplicação.

DestroyWindow

Método para destruir uma janela de aplicação.

ResizeWindow

Método para redimensionar o tamanho de uma janela de aplicação.

setWindowPosition

Método para posicionar a janela

## 6.2 Classe ApiWrapper (classe abstrata)

Esta classe é responsável por tornar portável os atributos e métodos específicos de um sistema operacional. Ela é uma classe abstrata e os seus métodos devem ser definidos pelas classes derivadas.

### 6.2.1 Atributos

### 6.2.2 Métodos

## 6.3 Classe WinApi Wrapper (herdade de ApiWrapper)

Esta classe define os atributos e os métodos específicos do sistema operacional Windows.

### 6.3.1 Atributos

WGLOpenGL

WindowClass

### 6.3.2 Métodos

Window Procedure

Método responsável por tratar os eventos do sistema operacional.

## 6.4 Classe GameEngineEventHandler (classe abstrata)

Esta classe é responsável por definir funções de callback que serão chamadas pela game engine implicitamente. O usuário deverá criar uma classe derivada e definir as suas próprias funções de callback e configurar a game engine.

## 6.5 Classe Render

Esta classe é responsável por armazernar diversas configurações relacionadas renderização.

## 6.6 Classe IO

Esta classe é responsável por armazenar diversas informações a respeito de entrada e saída de dados.

## 6.7 Classe Mouse

Esta classe é responsável por receber os dados de entrada do mouse e disponibilizar para a game engine.

## 6.8 Classe Teclado

Esta classe é responsável por receber os dados de entrada do teclado e disponibilizar para a game engine.

## 6.9 Classe AssetManager

Esta classe é responsável por gerenciar diversos recursos como assets.

# 7. Personalização

# 8. Convenção de Código