**GUIA DE DESENVOLVIMENTO**



**BPM Game Engine**

**Versão 1.0**

Fabio Takeshi Ishikawa

Janeiro de 2020

Sumário

[**1. Introdução** 3](#_Toc36146202)

[**1.1 O que é BPM Game Engine** 4](#_Toc36146203)

[**2. Visão Geral da Arquitetura** 4](#_Toc36146204)

[**2.1 Camada de Hardware** 5](#_Toc36146205)

[**2.2 Camada de Sistema Operacionais** 5](#_Toc36146206)

[**2.3 Camada API de Terceiros** 6](#_Toc36146207)

[**2.4 Camada Multiplataforma** 6](#_Toc36146208)

[**2.5 Camada Core** 8](#_Toc36146209)

[**2.5.1 Loop principal** 8](#_Toc36146210)

[**2.5.2 Controlando o tempo** 8](#_Toc36146211)

[**2.5.3 Sistema de Janelas** 8](#_Toc36146212)

[**2.5.4 Eventos de Usuários** 9](#_Toc36146213)

[**3. Configuração do Ambiente de Desenvolvimento** 9](#_Toc36146214)

[4. Dependências 9](#_Toc36146215)

[5. Debugando seu código 9](#_Toc36146216)

[**6. Classes de BPM Game Engine** 10](#_Toc36146217)

[6.1 Classe Game Engine 10](#_Toc36146218)

[6.1.1 Atributos 10](#_Toc36146219)

[6.1.2 Métodos 10](#_Toc36146220)

[6.2 Classe GEWindow 10](#_Toc36146221)

[6.2.1 Atributos 11](#_Toc36146222)

[6.2.2 Métodos 12](#_Toc36146223)

[6.3 Classe ApiWrapper 13](#_Toc36146224)

[6.3.1 Atributos 13](#_Toc36146225)

[6.3.2 Métodos 13](#_Toc36146226)

[6.4 Classe WinApi Wrapper (herdade de ApiWrapper) 13](#_Toc36146227)

[6.4.1 Atributos 14](#_Toc36146228)

[6.4.2 Métodos 14](#_Toc36146229)

[6.5 Classe GEEventHandler 14](#_Toc36146230)

[6.5.1 Métodos 15](#_Toc36146231)

[7. Convenção de Código 16](#_Toc36146232)

# **1. Introdução**

Este documento tem como objetivo descrever em detalhes a arquitetura de BPM Game Engine, um motor de jogo para o desenvolvimento de jogos baseados em modelos de processos de negócio. São apresentadas também todas as classes que compõem o motor de jogo, assim como os seus relacionamentos.

Abordamos também as ferramentas e linguagens de programação utilizadas para o desenvolvimento, práticas e conceitos de engenharia de software aplicadas neste projeto, incluindo o gerenciamento de configuração, convenção de código, planejamento e execução de testes, ferramentas de depuração de código, o fluxo de trabalho adotado com *Git* (sistema de controle de versão) e integração contínua com *Travis CI*.

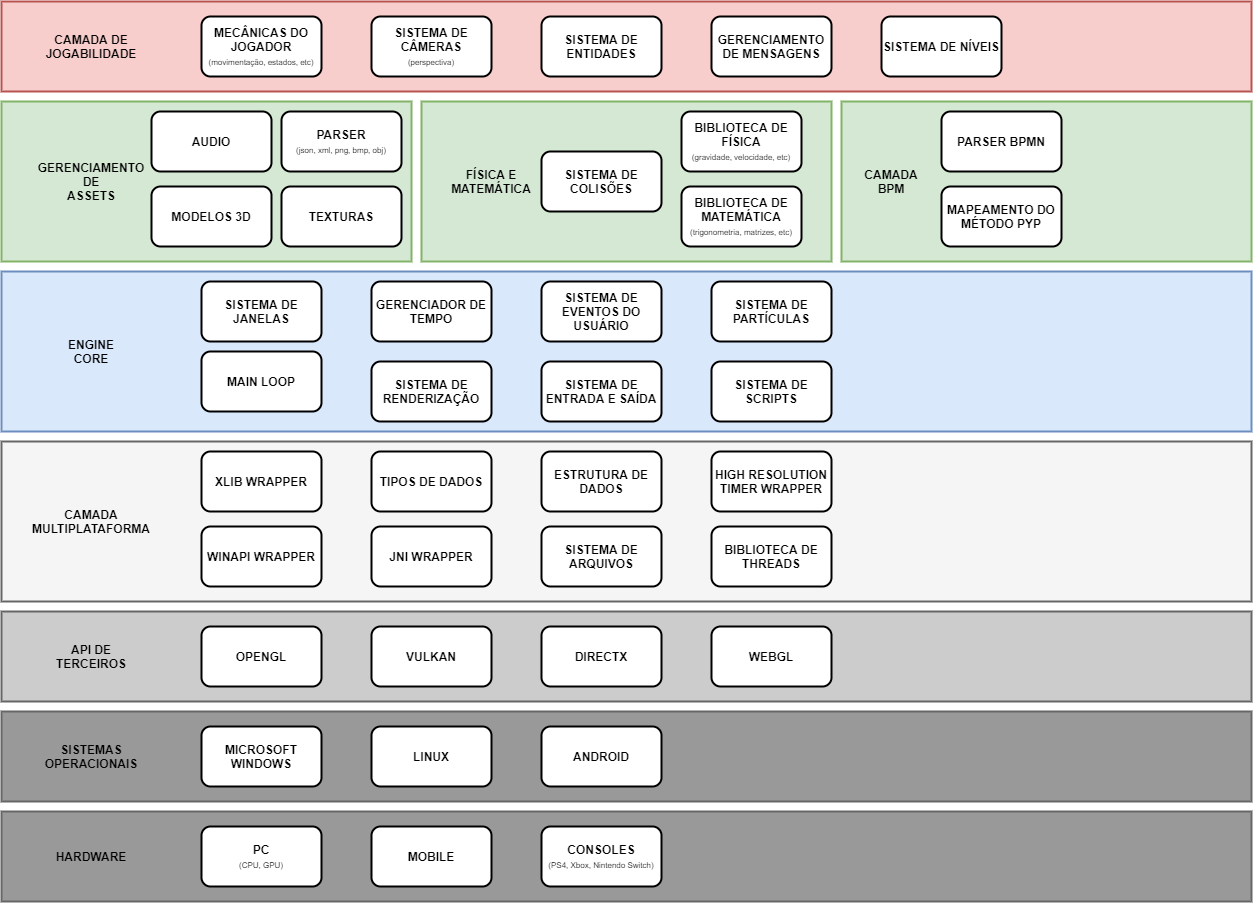
## **1.1 O que é BPM Game Engine**

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 à computação paralela tanto na parte de software como em hardware. A evolução do hardware chegou em um nível onde praticamente todos os sistemas computacionais modernos possuem processadores com diversos núcleos e a utilização de múltiplos GPUs. Por exemplo, o sistema 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**

Além do suporte a diferentes tipos de hardwares, o motor de jogo deverá lidar com diferentes tipos de sistemas operacionais. Cada sistema operacional possui o seu próprio gerenciamento de processos, gerenciamento de memória, sistema de arquivos, biblioteca de threads entre outros. [Detalhar mais esta parte e exemplificar aqui como funciona os itens citados no Windows e no Linux].

Além disso, cada sistema operacional gerencia a interface gráfica do usuário (GUI) de uma maneira diferente. Por exemplo, no Microsoft Windows, as aplicações que utilizam interface gráfica (isto é, aplicações com janelas) são aplicações orientadas a mensagens. O fluxo de execução é baseado em mensagens que o sistema operacional envia para a aplicação. (MORRISSON, 2002). A Microsoft oferece uma API chamada de WinAPI para criar aplicações com interface gráfica do sistema operacional enquanto no Linux é utilizada a API XWindow. O motor de jogo deve abstrair as funções dessas APIs para criar as aplicações de jogo independente de sistema operacional.

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.

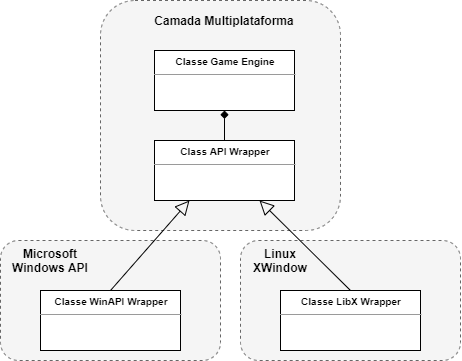
## **2.3 Camada API de Terceiros**

Este projeto utiliza algumas 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 responsável por definir os componentes que abstraem os componentes específicos de sistema operacional (por exemplo, WinAPi e X Window) e de hardware (por exemplo, registradores especiais como High Resolution Timer).

A ideia é utilizar o polimorfismo (conceito de programação orientado a objetos) para manipular esses componentes específicos de forma independente.

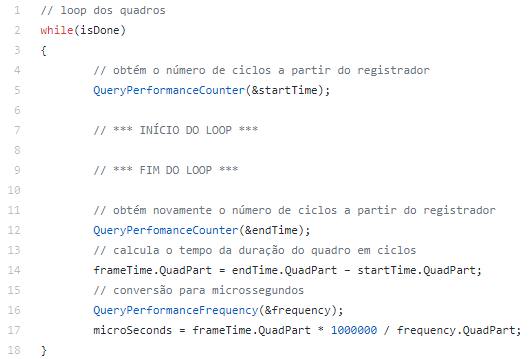


*Diagrama de classes para representar os componentes multiplataforma.*

A classe ApiWrapper é responsável por disponibilizar métodos específicos de cada sistema operacional de forma abstraída e transparente. Dessa forma, para cada sistema operacional, é necessário criar uma classe derivada de APIWrapper e definir os seus métodos virtuais com funções correspondentes ao sistema operacional. Com o polimorfismo, a utilização de funções específicas do sistema operacional fica transparente com ApiWrapper. O tipo do objeto é definido no momento da compilação através de uma constante de compilação.

Outro componente essencial da Camada Multiplataforma é o registrador *High-Resolution-Timer*. Ele é um registrador especial implementado em nível de hardware onde retorna o número de ciclos decorridos da CPU desde o momento em que foi ligado. (GREGORY, 2018). A classe TimeHandler da camada Core utiliza este componente para realizar todos os cálculos relacionados ao tempo. De fato, existem diversas formas de se obter o tempo decorrido da máquina como, por exemplo, a função time() da biblioteca padrão de C que retorna o tempo percorrido em segundos ou a função GetTickCount() da WinAPI que retorna o tempo percorrido em milissegundos. Entretanto, nenhum deles retorna o tempo com uma resolução tão alta quanto esses registradores – resolução de pelo menos um microssegundo (µ).

Demonstração do cálculo da duração do quadro utilizando os registradores High Resolution Timer para Microsoft Windows:



A ideia da conversão de ciclos para microssegundos é multiplicar primeiro o número de ciclos por 1000000 microssegundos (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.

### **2.5.1 Loop principal**

### **2.5.2 Controlando o tempo**

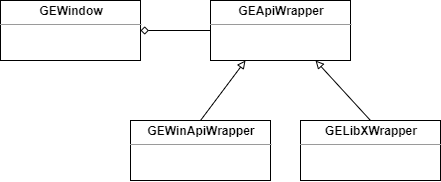
TimeHandler

responsável por calcular o tempo de duração de cada quadro do jogo em microssegundos (µ),

### **2.5.3 Sistema de Janelas**

O sistema de janelas deste motor de jogo é formado pelas classes GEWindow e GEApiWrapper. A classe GEWindow define atributos como largura e altura em pixels da área da janela (sem considerar o tamanho da borda e barra de título), estilo da janela, posição em pixels nos eixos x e y da tela e o nome que será exibido na barra de títulos da janela. Esta classe também pode ser utilizada para criar uma janela de aplicação, entretanto, ela depende de funções específicas do sistema operacional. Portanto, a classe GEWindow deve possuir uma referência a um objeto da classe GEApiWrapper.

A classe GEApiWrapper foi criada para abstrair as funções específicas do sistema operacional, fornecendo funções como criação de janelas de forma independente do sistema operacional. Estas funções são métodos virtuais que devem ser sobreescritas pelas classes herdadas de GEApiWrapper. Por exemplo, para criar uma janela de aplicação no sistema operacional Microsoft Window, o relacionamento das classes ficaria:



*Diagrama de classes que representa o componente Sistema de Janelas*

**Nota:** De fato, é possível criar uma janela de aplicação utilizando apenas a classe GEApiWrapper ou uma das classes herdadas. Isso evita overhead, porém perde muito em questões de reutilização e organização e manutenibilidade.

### **2.5.4 Eventos de Usuários**

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.

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

# 4. Dependências

# 5. Debugando seu código

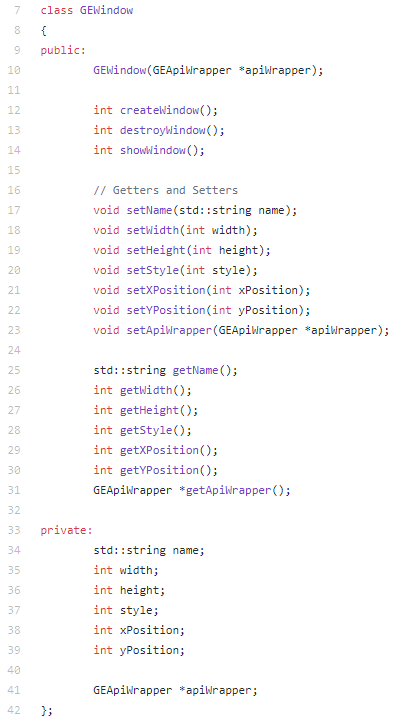
# **6. Classes de BPM Game Engine**

## 6.1 Classe Game Engine

### 6.1.1 Atributos

### 6.1.2 Métodos

## 6.2 Classe GEWindow



### 6.2.1 Atributos

std::string name

Texto que será exibido na barra de título da janela de aplicação.

int width

Largura em pixels da área da janela de aplicação (o tamanho das bordas e da barra de título não são consideradas).

int height

Altura em pixels da área da janela de aplicação (o tamanho das bordas e da barra de título não são consideradas).

int style

Estilo de uma janela de aplicação. Com este atributo é possível definir vários estilos como, por exemplo, uma janela sem bordas e sem barra de título, janelas que podem ou não ser redimensionáveis ou uma janela com todos os botões do sistema (minimizar, maximizar e fechar).

int xPosition

Posição inicial em pixel no eixo x da tela.

int yPosition

Posição inicial em pixel no eixo y da tela.

GEApiWrapper \*apiWrapper

Referência para um objeto da classe GEApiWrapper.

### 6.2.2 Métodos

GEWindow(GEApiWrapper \*apiWrapper)

Construtor da classe. Para instanciar um objeto desta classe é necessário passar uma referência a um objeto da classe GEApiWrapper.

int createWindow()

Método para criar uma janela baseada nos atributos da classe. Se o método não conseguir criar uma classe, ela retorna 0. Caso contrário, ela retorna um valor diferente de 0.

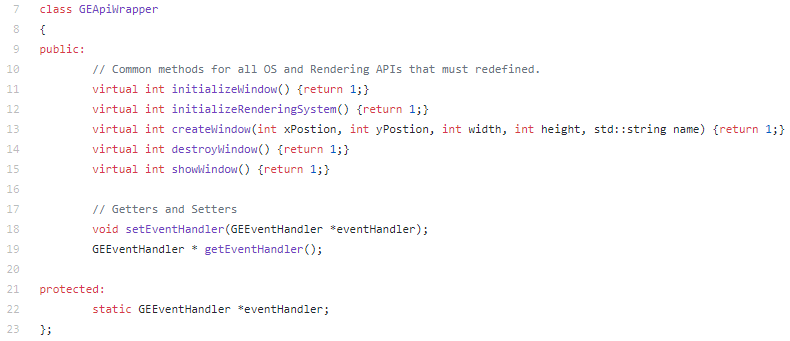
int destroyWindow()

Método para destruir uma janela de aplicação. Este método realiza a limpeza de objetos específicos do sistema operacional ao destruir uma janela.

int showWindow()

Método para exibir a janela de aplicação na tela.

## 6.3 Classe ApiWrapper



### 6.3.1 Atributos

static GEEventHandler \*eventHandler

Referência para um objeto GEEventHandler. Esta referência é importante para que o usuário do motor de jogo consiga acessar as funções específicas do sistema operacional.

### 6.3.2 Métodos

virtual int initializeWindow();

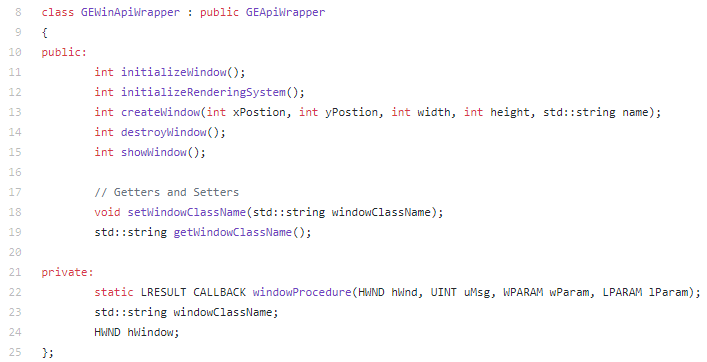
virtual int initializeRenderingSystem();

virtual int createWindow(int xPostion, int yPostion, int width, int height, std::string name);

virtual int destroyWindow();

virtual int showWindow();

## 6.4 Classe WinApi Wrapper (herdade de ApiWrapper)



### 6.4.1 Atributos

std::string windowClassName

HWND hWindow

### 6.4.2 Métodos

int initializeWindow()

int initializeRenderingSystem()

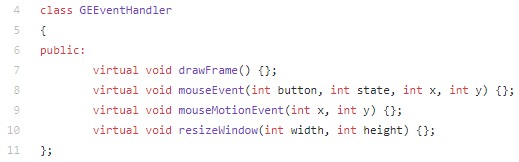
int createWindow(int xPostion, int yPostion, int width, int height, std::string name)

int destroyWindow()

int showWindow()

static LRESULT CALLBACK windowProcedure(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

## 6.5 Classe GEEventHandler



### 6.5.1 Métodos

virtual void drawFrame() {}

virtual void mouseEvent(int button, int state, int x, int y) {}

virtual void mouseMotionEvent(int x, int y) {}

virtual void resizeWindow(int width, int height) {}

# 7. Convenção de Código