**GUIA DE DESENVOLVIMENTO**



**BPM Game Engine**

**Versão 1.0**

Fabio Takeshi Ishikawa

Janeiro de 2020

Sumário

[**1. Introdução** 4](#_Toc37853865)

[**1.1 O que é BPM Game Engine** 4](#_Toc37853866)

[**1.2 Motivação do Projeto** 4](#_Toc37853867)

[**1.3 Objetivo deste documento** 4](#_Toc37853868)

[**1.4 Público-alvo deste documento** 4](#_Toc37853869)

[**2. BPM Game Engine** 5](#_Toc37853870)

[**2.1 Diferença entre outros motores de jogos** 5](#_Toc37853871)

[**2.2 Arquitetura Geral de BPM Game Engine** 5](#_Toc37853872)

[2.2.1 Camada de Hardware 6](#_Toc37853873)

[2.2.2 Camada de Sistema Operacionais 6](#_Toc37853874)

[2.2.3 Camada API de Terceiros 7](#_Toc37853875)

[2.2.4 Camada Multiplataforma 7](#_Toc37853876)

[**2.5 Camada Core** 9](#_Toc37853877)

[2.5.1 Loop do Jogo 9](#_Toc37853878)

[2.5.2 Gerenciador de Tempo 9](#_Toc37853879)

[2.5.3 Sistema de Janelas 9](#_Toc37853880)

[**2.5.4 Eventos de Usuários** 10](#_Toc37853881)

[2.5.5 Entrada e Saída com Teclado, Mouse e Joystick 10](#_Toc37853882)

[2.5.6 Motor de Renderização 10](#_Toc37853883)

[2.5.7 Animação 10](#_Toc37853884)

[2.5.8 Sistema de Scripts 10](#_Toc37853885)

[**3. Configuração do Ambiente de Desenvolvimento** 11](#_Toc37853886)

[4. Dependências 11](#_Toc37853887)

[5. Debugando seu código 11](#_Toc37853888)

[**6. Classes de BPM Game Engine** 12](#_Toc37853889)

[6.1 Classe Game Engine 12](#_Toc37853890)

[6.1.1 Atributos 12](#_Toc37853891)

[6.1.2 Métodos 12](#_Toc37853892)

[6.2 Classe GEWindow 12](#_Toc37853893)

[6.2.1 Atributos 13](#_Toc37853894)

[6.2.2 Métodos 15](#_Toc37853895)

[6.3 Classe ApiWrapper 15](#_Toc37853896)

[6.3.1 Atributos 15](#_Toc37853897)

[6.3.2 Métodos 16](#_Toc37853898)

[6.4 Classe WinApi Wrapper (herdade de ApiWrapper) 16](#_Toc37853899)

[6.4.1 Atributos 16](#_Toc37853900)

[6.4.2 Métodos 16](#_Toc37853901)

[6.5 Classe GEEventHandler 17](#_Toc37853902)

[6.5.1 Métodos 17](#_Toc37853903)

[7. Convenção de Código 18](#_Toc37853904)

# **1. Introdução**

Este documento tem como objetivo servir como um guia de desenvolvimento para o projeto BPM Game Engine, um motor de jogo para o desenvolvimento de jogos baseados em modelos de processos de negócio.

Este guia está dividido em diversos capítulos, onde é descrito o que é BPM Game Engine e as suas principais funcionalidades; a arquitetura geral e os principais componentes que fazem parte do motor de jogo; todas as ferramentas e linguagens de programação utilizadas para o desenvolvimento, práticas e conceitos de engenharia de software como convenção de código, depuração de código, gerenciamento de configuração, fluxo de trabalho com o sistema de controle de versão e a integração contínua empregada neste projeto.

## **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 gênero de jogo, que apresenta um processo de negócio de forma ludificada, permite aos jogadores compreender e aprender seu funcionamento de forma divertida e engajante e desenvolver reflexões em relação à sua necessidade, sua prática, seus valores, seus desafios e limitações de execução. (CLASSE e ARAUJO).

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.

## **1.2 Motivação do Projeto**

## **1.3 Objetivo deste documento**

## **1.4 Público-alvo deste documento**

# **2. BPM Game Engine**

BPM Game Engine é um motor de jogo multiplataforma com capacidade de renderização em tempo real de imagens bidimensionais e tridimensionais. Com o motor de jogo é possível criar um jogo 2D ou 3D completo do início até o final. Além disso, ele fornece diversas funcionalidades para criar um jogo do gênero baseado em modelos de processos de negócio como um parser de arquivos BPMN (Business Process Model Notation) e um mapeamento de elementos de negócio em elementos de jogos.

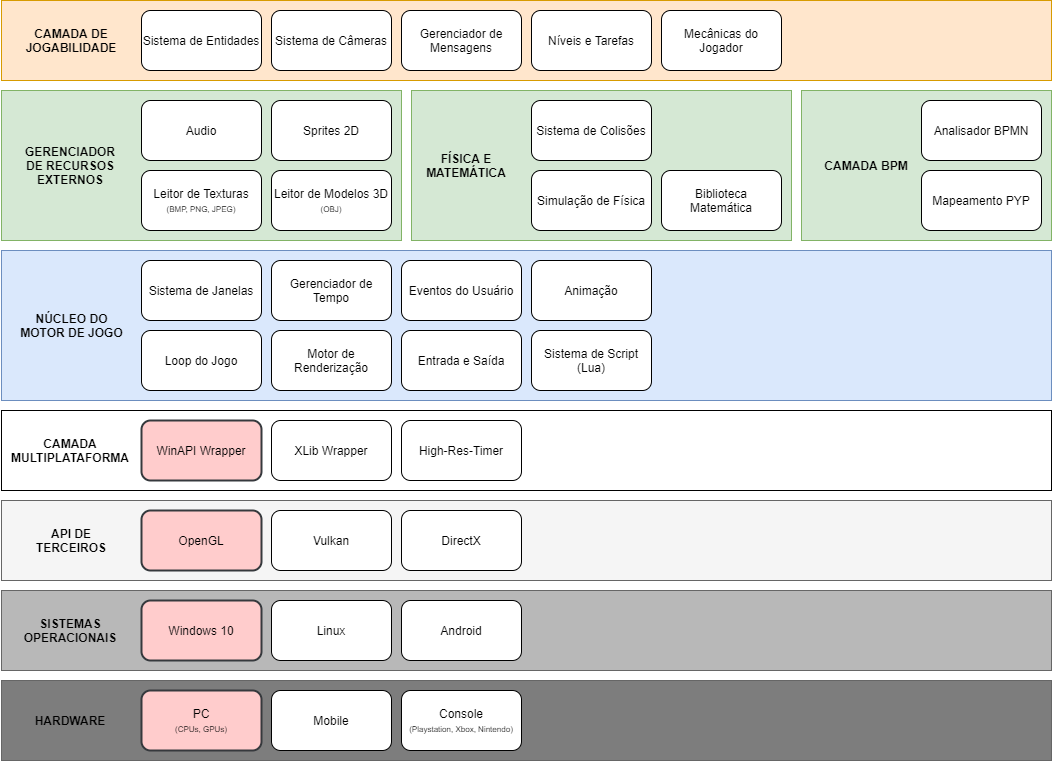
## **2.1 Diferença entre outros motores de jogos**

[Descrever aqui os diferentes tipos de motor de jogo e explicar em qual modelo BPM Game Engine se encaixa. Por exemplo, middleware, framework, etc]

## **2.2 Arquitetura Geral de BPM Game Engine**

Assim como a maioria dos motores de jogos, BPM Game Engine é composto por diversos componentes e subsistemas que devem trabalhar em conjunto para formar, de fato, um motor de jogo como um todo.

Estes componentes e subsistemas podem ser divididos em diversas camadas para representar os relacionamentos e a dependência entre eles. Segue a imagem com todos os componentes:



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

### 2.2.1 Camada de Hardware

Este projeto nasceu com a ideia de possibilitar a utilização deste motor de jogo 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).

### 2.2.2 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.

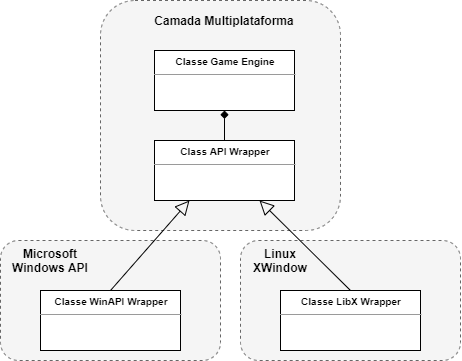
### 2.2.3 Camada API de Terceiros

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.

### 2.2.4 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.

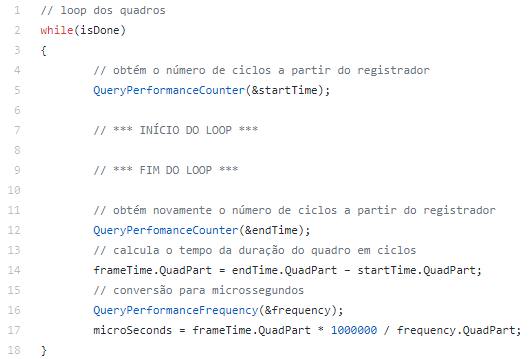


*Diagrama de classes para representar os componentes multiplataforma.*

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 do Jogo

### 2.5.2 Gerenciador de Tempo

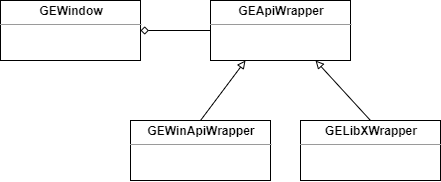
TimeHandler

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. Uma janela de aplicação é criada utilizando esta classe, 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*

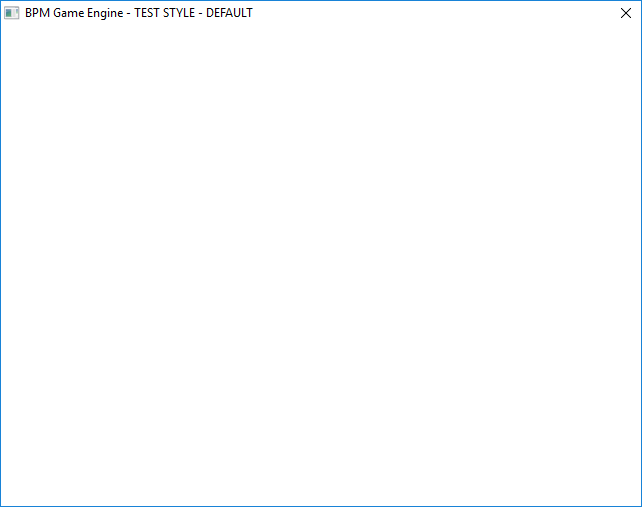
**Nota:** De fato, é possível criar uma janela de aplicação utilizando apenas a classe GEApiWrapper ou uma das classes herdadas. Isso evita overhead, porém perde muito em questões de reutilização e organização e manutenibilidade.

#### 2.5.3.1 Estilos da janela

Com o sistema de janelas de BPM Game Engine, é possível criar diferentes estilos de janelas. A seguir, segue os estilos que podem ser definidos:

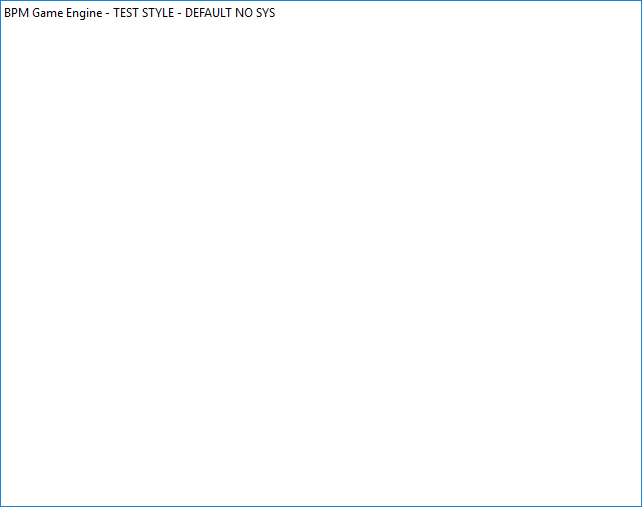
**Estilo padrão (GE\_WIN\_DEFAULT)**

Este estilo define uma janela de aplicação popup, com uma barra de título e ícone, botão de sistema fechar e não redimensionável.



**Estilo padrão sem o botão fechar (GE\_WIN\_DEFAULT\_NO\_SYS)**

Este estilo define uma janela de aplicação popup, com uma barra de título e não redimensionável.



**Estilo splash screen (GE\_WIN\_SPLASH)**

Este estilo define uma janela de aplicação popup, sem barra de título, sem botões de sistema e não redimensionável.



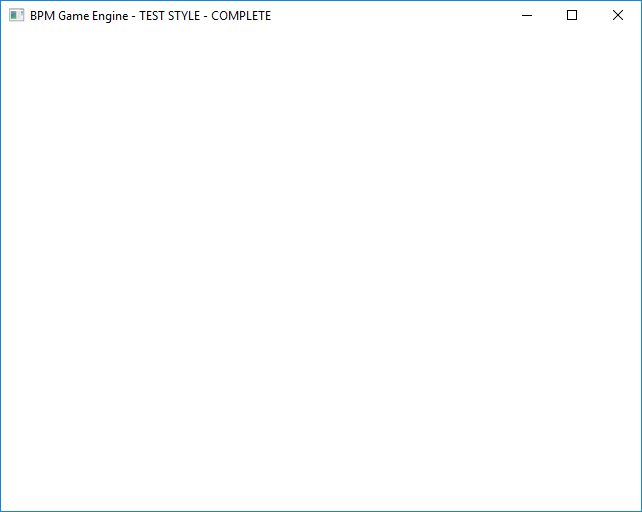
**Estilo tela cheia modo janela (GE\_WIN\_WINDOWED\_FULLSCREEN)**

Este estilo define uma janela de aplicação popup, sem barra de título, sem botões de sistema, maximizada (tela cheia) e não redimensionável.

Nota: Falar sobre as diferenças entre este modo e a tela cheia de fato.

**Estilo janela popup completo (GE\_WIN\_COMPLETE)**

Este estilo define uma janela de aplicação popup, com barra de título, com todos os botões de sistema (minimizar, maximizar e fechar janela) e redimensionável.



Configurações adicionais de estilos de janela

Ainda é possível adicionar alguns estilos adicionais para os estilos citados acima:

* Iniciar a janela maximizada
* Iniciar a janela minimizada
* Adicionar um botão de sistema minimizar
* Adicionar um botão de sistema maximizar
* Adicionar um botão de sistema fechar janela
* Tornar a janela redimensionável

### **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.

### 2.5.5 Entrada e Saída com Teclado, Mouse e Joystick

### 2.5.6 Motor de Renderização

### 2.5.7 Animação

### 2.5.8 Sistema de Scripts

# **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

??

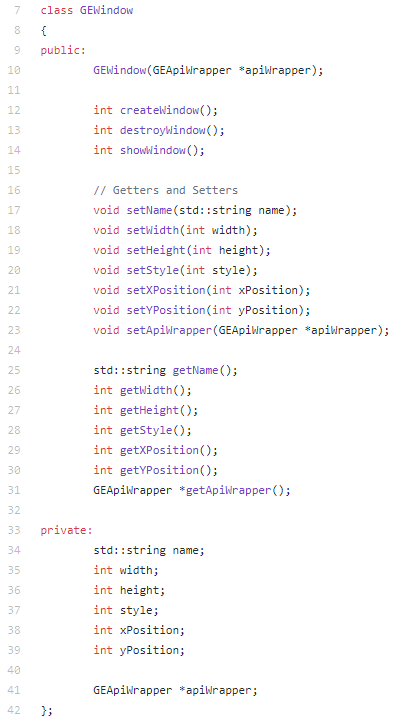
mainLoop

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 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).

|- style

|- dialog box

|- Grupo de Controle

|- Scroll horizontal

|- Scroll vertical

|- Botão de maximizar

|- Botão de minimizar

|- Desativar

|- Bordar

|- Titulo

|- WS\_OVERLAPPED (titulo e borda)

|- WS\_OVERLAPPED\_WINDOW (titulo, borda, icone, botao minimizar, botao maximizar, botao sair)

|- janela redimensionavel

|- WS\_SYSMENU (icon, titulo, borda, botao sair)

|- TABCONTROL

- Barra de título com todos os botões e da mesma cor de fundo da tela

- Janela com várias opções de menu e com a cor da barra de título personalizada

- Janela com fundo transparente

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.

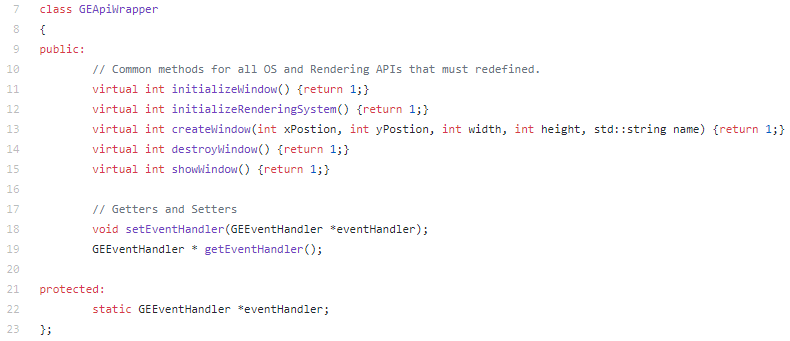
int destroyWindow()

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();

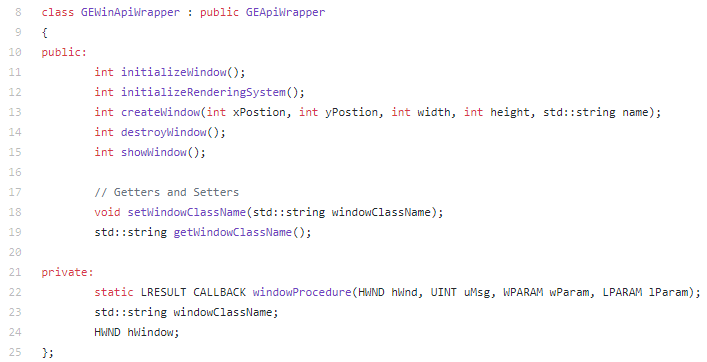
virtual int initializeRenderingSystem();

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()

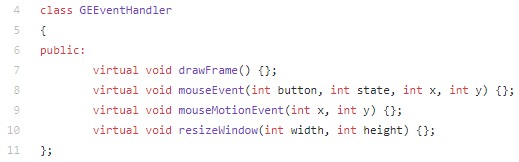
int createWindow(int xPostion, int yPostion, int width, int height, std::string name)

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