

**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS**  
**ESCOLA POLITÉCNICA**  
**GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO**



**FRAMEWORK PARA GERENCIAMENTO COMPLETO E PADRONIZAÇÃO  
DE PROJETOS NA GODOT ENGINE**

**LUCAS SILVA QUEIROZ**

**GOIÂNIA**  
**2024**

**LUCAS SILVA QUEIROZ**

**FRAMEWORK PARA GERENCIAMENTO COMPLETO E PADRONIZAÇÃO  
DE PROJETOS NA GODOT ENGINE**

Trabalho de Conclusão de Curso apresentado à Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção de título de Bacharel em Engenharia de Computação.

Orientador:

Prof. Me. Fernando Gonçalves Abadia

Banca examinadora:

Profa. Dra. Carmen Cecília Centeno

Prof. Dr. Anibal Santos Jukemura

**GOIÂNIA  
2024**

**LUCAS SILVA QUEIROZ**

**FRAMEWORK PARA GERENCIAMENTO COMPLETO E PADRONIZAÇÃO  
DE PROJETOS NA GODOT ENGINE**

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Engenharia de Computação, em \_\_\_\_/\_\_\_\_/\_\_\_\_\_.

---

Orientador: Prof. Me. Fernando Gonçalves Abadia

---

Profa. Dra. Carmen Cecília Centeno

---

Prof. Me. Anibal Santos Jukemura

**GOIÂNIA  
2024**

## **DEDICATÓRIA**

Dedico este trabalho aos dois seres que mais impactaram a minha existência: ao meu eterno amigo e companheiro de quatro patas, Pirulei (†), que foi a razão de minha dedicação tão intensa à programação quando se foi e deixou um vazio insuprível; e ao meu irmão Cauan, a pessoa que eu mais amo neste mundo, e que para mim será eternamente meu mamãozinho.

## AGRADECIMENTO

Este trabalho é fruto da contribuição de diversas pessoas, que durante os anos da minha vida, contribuíram com alguma lição valiosa. A cada experiência vivida, aprendi um pouco mais sobre a vida e o mundo, e fiz o possível para transmitir isso por onde passei. Agradeço a cada uma dessas pessoas, e em especial a estas:

Primeiramente ao meu melhor amigo, Alexandre Mascarenhas, por estar ao meu lado durante os momentos bons e ruins, me apoiando com coisas que foram desde ficar em chamada na madrugada durante minhas crises de ansiedade, até me ajudar a arrumar a casa nos dias que eu não tive ânimo para nada.

Ao meu irmão Cauan, por trazer muita alegria à minha vida desde que nasceu, e me permitir continuar tendo esperanças de um mundo melhor, além de ser a pessoa que eu mais amo nesse mundo todo, e aos meus pais Nilda e André, por estarem ao meu lado até mesmo nas ideias mais loucas que eu já tive, sempre acreditando em mim, me inspirando a ir além dos meus limites e por serem exemplo na minha vida e os pais mais incríveis que eu poderia ter.

À minha madrinha Fabiana, aos meus avós Francisca, Fernando, Norma e Manoel, e às minhas tias Nilva e Nilza por me amarem incondicionalmente, me encherem de carinho desde o dia que eu nasci, e me ensinarem lições marcantes para a vida.

Aos meus amigos Raul Navia e Fabiola Delgado, por me proporcionarem uma amizade tão verdadeira que nem o tempo nem a distância foram capazes de diminuir.

Aos meus amigos Giovanna Sampaio e Gabriel Carnelutti, por terem me acompanhado e apoiado em diversos momentos que eu precisei, sem medir esforços.

Aos meus amigos de infância Mariana Menezes e Felipe Delfino, por terem me permitido viver uma amizade tão sincera que literalmente tem durado a vida toda.

À minha primeira professora de programação, Karina Alvarez, por ter me motivado a me desafiar e descobrir minhas habilidades na computação.

Aos meus professores de parkour, Azaf Moreira, Thiago Monteiro, Pedro Romani, Thais Lelis e João Junior, por me ensinarem uma nova arte e me permitir enxergar o mundo e as pessoas por uma nova perspectiva.

Aos meus vizinhos Graça, Marilú, Angélica e Aníbal, por se importarem comigo até mesmo nos dias em que eu estava mais ocupado, mesmo que fosse para dar bom dia.

A Alexandre Longo, Elis Regina, Lhasa De Sela, Stuart A. Staples, Nikolai Rimsky-Korsakov, Camille Saint-Saëns e Ludwig van Beethoven por comporem e executarem as obras musicais mais fantásticas que foram a trilha sonora da minha jornada.

À 42 São Paulo por me permitir conhecer pessoas incríveis (que pretendo levar para o resto da vida) e viver experiências inesquecíveis que me tornaram mais forte psicologicamente e mais humano, além de ter contribuído com uma parte essencial quanto a conhecimentos técnicos.

Por fim, aos professores da PUC que tiveram grande influência durante minha graduação: Fernando Gonçalves, Carmen Centeno, Marco Antônio Figueiredo, Mirian Gusmão, Cláudio Martins Garcia, Cristian Nova, Antônio Marcos, Antônio Newton, Solange Da Silva, Lucília Ribeiro, Raffael Figueiredo, Bercholina Honorato, Joel Ferreira e Polliana Alves. Agradeço especialmente ao professor Talles pelos conselhos de vida.

## EPÍGRAFE

*“O sucesso é a soma de múltiplos fracassos acumulados.”*

(Lucas S. Queiroz)

## RESUMO

Algumas técnicas como *frameworks* ou *design patterns* estão ligadas diretamente ao reuso de alguma característica de projetos. Quando aplicadas adequadamente, possibilitam melhorias na qualidade de software, reduzem a quantidade de esforços e agilizam os processos de produção e manutenção. Este trabalho visa documentar e explicar o processo de desenvolvimento de um *framework* utilizando a Godot Engine, aplicando estruturas de projeto e fazendo uso de classes especializadas, com a finalidade de acelerar o processo de desenvolvimento de jogos e permitir uma fácil manutenção, ao abstrair componentes reutilizáveis que podem ser aplicados em diferentes projetos, todavia mantendo um modelo de estrutura e arquitetura padronizado. O modelo de arquitetura desenvolvida nesta pesquisa foi organizado em camadas com propósitos específicos, adaptados às necessidades e recursos da Godot Engine e do projeto de pesquisa. Além do desenvolvimento, foram realizadas demonstrações e testes utilizando dois projetos, tendo somente um deles utilizado o *framework* produzido nesta pesquisa. Com os testes, foram medidos e comparados 38 critérios, agrupados nas categorias de critérios quantitativos de implementação, de desempenho de execução e de uso hardware, e posteriormente comparados utilizando os valores do projeto sem o *framework* como referência.

Palavras-chave: Godot-engine, Game-engine, Framework, Padronização, Modularização.

## **ABSTRACT**

Certain methodologies, such as frameworks and design patterns, are intrinsically linked to the reuse of project features. When properly applied, they enhance software quality, reduce effort, and expedite production and maintenance processes. This study aims to document and elucidate the development process of a framework using the Godot Engine, incorporating design structures and leveraging specialized classes to accelerate game development and facilitate maintenance by abstracting reusable components applicable across various projects, while maintaining a standardized structural and architectural model. The architectural model developed in this research was systematically organized into layers with specific purposes, tailored to the requirements and resources of the Godot Engine and the research project. In addition to development, demonstrations and testing were conducted on two projects, with only one utilizing the framework developed in this research. The testing involved measuring and comparing 38 criteria, categorized into quantitative implementation metrics, execution performance, and hardware usage, and subsequently compared, using the values from the project without the framework as the reference baseline.

**Keywords:** Godot-engine, Game-engine, Framework, Standardization, Modularization.



## LISTA DE FIGURAS

Figura 1 - Estrutura de nós de uma cena na Godot Engine .....	21
Figura 2 – Conexão do sinal de um nó a um método por meio de interface gráfica .....	24
Figura 3 - Exibição de <i>annotations</i> na aba <i>Inspector</i> .....	26
Figura 4 - Visualização da aba <i>AssetLib</i> .....	26
Figura 5 - Estrutura de repositórios e ramificações do Git.....	27
Figura 6 - Tela inicial do jogo <i>Masterpiece Madness</i> .....	30
Figura 7 - Estrutura da cena principal de um projeto utilizando o <i>framework</i> <i>FDCore</i> .	36
Figura 8 - Representação do sistema de ativação de ações do <i>framework</i> <i>FDCore</i> .....	37
Figura 9 - Métodos públicos da classe <i>FDCore</i> , organizados em seções .....	38
Figura 10 - Atributos públicos da classe <i>FDCore</i> .....	39
Figura 11 - Métodos e atributos públicos da classe <i>Transition</i> .....	40
Figura 12 - Métodos e atributos públicos da classe <i>LogoIntro</i> .....	42
Figura 13 - Variáveis públicas da classe <i>LogoIntro</i> .....	42
Figura 14 - Customização de animações de um nó do tipo <i>ActionHUD</i> .....	43
Figura 15 - Variáveis públicas da classe <i>ActionHUD</i> .....	44
Figura 16 - Atributos e métodos públicos da classe <i>ActionHUD</i> .....	45
Figura 17 - Definindo o nó raiz de interface gráfica em um nó do tipo <i>ActionButton</i> ..	45
Figura 18 - Possíveis ações customizáveis dos eventos da classe <i>ActionButton</i> .....	46
Figura 19 - Conexão de um sinal a uma ação, utilizando o nó <i>SignalActionTrigger</i> .....	46
Figura 20 - Organização da estrutura do repositório do <i>framework</i> no <i>GitHub</i> .....	47
Figura 21 - Utilizando o repositório do <i>framework</i> no <i>GitHub</i> como <i>template</i> .....	48
Figura 22 - Tela de criação de repositório no <i>GitHub</i> , utilizando <i>template</i> .....	49
Figura 23 - Criação de novo script gerenciador de projeto .....	50
Figura 24 – Especificando caminho do script gerenciador de projeto .....	50
Figura 25 - Exemplo de estrutura de cena de tela.....	50
Figura 26 - Configuração de nó do tipo <i>ActionButton</i> .....	51
Figura 27 - Configuração de ação ativada por <i>ActionButton</i> .....	52
Figura 28 - Opções de personalização de um <i>ActionHUD</i> .....	52
Figura 29 - Cenas resultantes do projeto de exemplo.....	53
Figura 30 - Fluxo de execução para testes de desempenho de execução .....	55

## **LISTA DE TABELAS**

Tabela 1 - Resultados dos testes com e sem o framework .....	57
Tabela 2 - Comparativos percentuais dos resultados obtidos após com os testes .....	58
Tabela 3 - Percentuais de aproveitamento por critério .....	59

## LISTA DE ALGORITMOS

Algoritmo 1 - Exemplo de sobrescrita de método de um script .....	22
Algoritmo 2 - Conexão do sinal de um nó a um método por meio de código.....	24
Algoritmo 3 - Exemplo do uso de <i>annotations</i> .....	25
Algoritmo 4 - Código de emissão de ações de clique no jogo Masterpiece Madness....	32
Algoritmo 5 - Chamada de uma função de <i>singleton</i> , no projeto Evolution.....	32
Algoritmo 6 - Código da função <i>join_match</i> em um <i>singleton</i> , no projeto Evolution...	33
Algoritmo 7 - Sobrescrita do método <i>_init</i> no gerenciador de projeto .....	54
Algoritmo 8 - Sobrescrita do método <i>_on_action_triggered</i> no gerenciador de projeto	54

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO</b>	<b>14</b>
1.1	Justificativa	15
1.2	Objetivos	15
1.2.1	<i>Objetivo geral</i>	15
1.2.2	<i>Objetivos específicos</i>	15
1.3	Resultados esperados	16
1.4	Metodologia	16
1.5	Organização do projeto	17
<b>2</b>	<b>REFERENCIAL TEÓRICO</b>	<b>18</b>
2.1	Frameworks	18
2.1.1	<i>Características de um framework</i>	18
2.2	Game engines	19
2.3	Godot Engine	20
2.3.1	<i>GScript</i>	21
2.3.2	<i>Signals</i>	23
2.3.3	<i>Annotations</i>	25
2.3.4	<i>AssetLib</i>	26
2.4	GitHub	27
<b>3</b>	<b>MATERIAIS E MÉTODOS</b>	<b>29</b>
<b>4</b>	<b>DESENVOLVIMENTO DO PROJETO</b>	<b>30</b>
4.1	Identificando os Requisitos do Projeto	30
4.1.1	<i>Requisitos funcionais</i>	34
4.1.2	<i>Requisitos não funcionais</i>	35
4.2	Arquitetura do projeto	35
4.3	Apresentação do framework	37
4.3.1	<i>FDCore</i>	38
4.3.2	<i>FDProjectManager</i>	39
4.3.3	<i>Transition</i>	40
4.3.4	<i>LogoIntro</i>	41
4.3.5	<i>ActionHUD</i>	43
4.3.6	<i>ActionButton</i>	45
4.3.7	<i>SignalActionTrigger</i>	46

<b>4.4</b>	<b>Repositório do framework.....</b>	<b>47</b>
<b>4.5</b>	<b>Configuração de novo projeto utilizando o framework .....</b>	<b>49</b>
<b>5</b>	<b>RESULTADOS .....</b>	<b>55</b>
<b>5.1</b>	<b>Critérios quantitativos relativos à implementação .....</b>	<b>55</b>
<b>5.2</b>	<b>Critérios de desempenho de execução .....</b>	<b>55</b>
<b>5.3</b>	<b>Critérios de uso de recursos de hardware .....</b>	<b>56</b>
<b>5.4</b>	<b>Projetos de teste e comparativos.....</b>	<b>56</b>
<b>5.5</b>	<b>Análise dos resultados obtidos .....</b>	<b>59</b>
<b>5.5.1</b>	<i>Quanto aos critérios de implementação.....</i>	<i>59</i>
<b>5.5.2</b>	<i>Quanto aos critérios de desempenho de execução .....</i>	<i>60</i>
<b>5.5.3</b>	<i>Quanto aos critérios de uso de hardware .....</i>	<i>60</i>
<b>6</b>	<b>CONSIDERAÇÕES FINAIS .....</b>	<b>61</b>
	<b>REFERÊNCIAS BIBLIOGRÁFICAS .....</b>	<b>63</b>
	<b>APÊNDICE A - Diagrama de classe - FDCore .....</b>	<b>71</b>
	<b>APÊNDICE B - Diagrama de classe – FDProjectManager .....</b>	<b>72</b>
	<b>APÊNDICE C - Diagrama de classe – ActionButton .....</b>	<b>72</b>
	<b>APÊNDICE D - Diagrama de classe – SignalActionTrigger .....</b>	<b>72</b>
	<b>APÊNDICE E - Diagrama de classe – Transition .....</b>	<b>73</b>
	<b>APÊNDICE F - Diagrama de classe – LogoIntro .....</b>	<b>74</b>
	<b>APÊNDICE G - Diagrama de classe – ActionHUD.....</b>	<b>75</b>

## 1 INTRODUÇÃO

Desenvolver um projeto de software muitas vezes tende a ser um desafio. Mesmo que cada projeto tenha suas particularidades, existem obstáculos em comum que podem levar à falha. Por isso é essencial que haja um balanceamento na velocidade de desenvolvimento (FORBES, 2024), podendo-se utilizar de recursos como *frameworks*, a fim de reduzir a quantidade de esforços (JOHNSON, 1997), agilizar os processos de produção e posteriormente de manutenção (MURILO e BITTENCOURT, 2021).

Um *framework* consiste em uma técnica ligada ao reuso de algum sistema, módulo ou até mesmo esqueleto de uma aplicação (JOHNSON, 1997). O uso de *frameworks* pode ser adotado por programadores de diferentes níveis (MYLLÄRNIEMI et al., 2018), possibilitando o aumento de qualidade de software, redução de custos (FAYAD, 1997) e estabelecendo melhores práticas de programação com uso adequado de *design patterns*.

Por sua vez, os *design patterns* são soluções reutilizáveis a problemas de design que podem ocorrer em diferentes situações, comumente apresentando soluções orientadas a objetos, nas quais se mostram as relações entre classes e objetos (EDWIN, 2014). Isso proporciona uma certa vantagem ao se reutilizar um modelo de design em vez de reutilizar códigos, tendo em vista que poderá ser aplicado a mais contextos (JOHNSON, 1997). Um exemplo são as *game engines* no contexto de desenvolvimento de jogos digitais, consideradas para alguns como *frameworks* (POLITOWSKY et al., 2020).

Para Thorn (2010), uma *game engine* é definida como o núcleo de um jogo, que contém a maior parte dos componentes generalizáveis em projetos de jogos. No entanto existem também algumas partes que são não-generalizáveis, como conteúdo do jogo (gráficos, áudios, malhas 3D, ajustes etc.) e ferramentas do jogo (editores de mapa, arquivos de configurações).

Para esta pesquisa foi utilizada a Godot Engine, uma *game engine* de código aberto para projetos 2D e 3D, desenvolvida inicialmente em 2014 e atualmente (2024) mantida por uma comunidade de mais de 2400 contribuidores (sendo quase todos voluntários) (GITHUB, 2024h; LINIETSKY e MANZUR et al., 2024).

Considerando que a Godot Engine utiliza uma estrutura hierárquica com componentes chamados “nós”, e permite a criação de classes personalizadas (GODOT, 2024m), apresenta-se a seguinte questão de pesquisa: como pode ser definida a estrutura, arquitetura e componentes de um framework que permita a criação e manutenção de

novos projetos, realizando o gerenciamento centralizado de fluxo de troca de telas e comunicação facilitada entre as telas, adaptados à Godot Engine, de forma que se reduza o tempo de codificação e a quantidade de linhas de código necessárias por projeto?

Atentando-se à questão de pesquisa, este trabalho visa documentar e explicar o processo de desenvolvimento do *framework* FDCore, utilizado inicialmente pelo estúdio de jogos Fire-Droid Game Studio. O código fonte do *framework* está disponível através da página “<https://github.com/FireDroidGameStudios/godot-project-template>”.

## **1.1 Justificativa**

Justifica-se estudar este tema devido à necessidade de um modelo padronizado para criação e manutenção de projetos futuros, redução no tempo de desenvolvimento e uma melhoria na curva de aprendizado para novos colaboradores, dado que, uma vez aprendido o padrão de projetos, o colaborador poderá atuar com mais facilidade em outros projetos já existentes, por não haver necessidade de aprender um padrão completamente novo.

## **1.2 Objetivos**

Nesta seção serão apresentados os objetivos geral e específicos do projeto.

### **1.2.1 Objetivo geral**

Desenvolver um framework para a Godot Engine, que possua componentes pré-implementados que permitam uma fácil organização e estruturação de projetos, reduzindo a quantidade de códigos e facilitando a comunicação e a interação entre as cenas do projeto.

### **1.2.2 Objetivos específicos**

- Aprofundar conhecimentos em Git, suas formas de estruturação de projetos e projetos de código aberto;
- Aprimorar conhecimentos sobre boas práticas de projetos na Godot Engine e linguagem de programação GDScript;
- Identificar os requisitos do projeto;
- Projetar a arquitetura do *framework*;
- Desenhar a organização e a estrutura dos projetos a serem desenvolvidos utilizando o *framework*;

- Planejar as classes e componentes a serem implementados;
- Desenvolver as classes e componentes do *framework*;
- Documentar o *framework*, incluindo suas classes e componentes;
- Desenvolver dois projetos de teste, um utilizando o *framework* e o outro não;
- Comparar os dois projetos de teste, em implementação, desempenho de execução e uso de hardware, e analisar os resultados.

### 1.3 Resultados esperados

Espera-se que como resultado desta pesquisa, se obtenha um *framework* capaz de reduzir a quantidade de esforços de implementação e manutenção em projetos desenvolvidos na Godot Engine, aplicando estruturas padronizadas e componentes modulares. Espera-se também que o desempenho de execução e de uso de hardware dos projetos que utilizem esse *framework*, seja próximo ao de projetos desenvolvidos sem sua utilização.

### 1.4 Metodologia

Esta pesquisa, segundo sua natureza é um trabalho original, pois a partir de observações e teorias, busca apresentar conhecimento novo (WAZLAWICK, 2014).

Segundo seus objetivos é exploratória e descritiva. A pesquisa exploratória busca tornar o problema mais explícito ou a construir hipóteses, por meio de uma maior familiarização do problema (Gil, 2017). A pesquisa descritiva busca realizar o levantamento de dados por meio de entrevistas, questionários ou outros meios, e posteriormente descrever os fatos como são (WAZLAWICK, 2014). Para esta pesquisa, o objetivo descritivo se relaciona aos testes de validação que serão realizados após o término das implementações.

Quanto aos procedimentos técnicos esta pesquisa é bibliográfica, documental e experimental.

Os procedimentos bibliográficos consistem na elaboração em materiais já publicados, sejam eles impressos, digitais ou armazenados em outros meios físicos. Suas etapas conformam a escolha do tema, elaboração do plano de trabalho abrangendo o primeiro semestre de 2024, identificação de materiais científicos já existentes sobre o assunto, compilação em fichamento de excertos relevantes dos materiais selecionados, análise textual e escrita da monografia (GIL, 2017; LAKATOS e MARCONI, 2017).



Os procedimentos de uma pesquisa documental consistem em analisar dados e documentos não sistematizados ou publicados, iniciando com a formulação do problema a ser resolvido, seguido da identificação das fontes de pesquisa, análise e interpretação dos dados, para que ao final possa contribuir para a escrita do texto.

Os procedimentos experimentais são caracterizados por implicar variáveis experimentais (controladas pelo pesquisador) e observadas (valores práticos), com a finalidade de alcançar conclusões de possíveis dependências entre tais variáveis. Seu planejamento implica na elaboração da questão de pesquisa, definição de um plano experimental (descrito no capítulo 5), determinação do ambiente de manipulação das variáveis (especificado no capítulo 3), coleta de dados e realização dos testes (demonstrado no capítulo 5), análise dos resultados obtidos e escrita do texto (GIL, 2017; WAZLAWICK, 2014).

### **1.5 Organização do projeto**

Quanto à organização do projeto, está composto em seis capítulos: Introdução, que aborda de forma geral sobre a ideia da pesquisa, seus objetivos e metodologias; Referencial Teórico, que aborda sobre os temas relacionados à pesquisa, provendo um contexto inicial ao leitor; Materiais e Métodos, que descreve as ferramentas utilizadas durante a realização da pesquisa; Desenvolvimento do Projeto, que documenta os processos realizados durante a pesquisa, e detalha aspectos sobre o funcionamento do projeto; Resultados, que apresenta as análises realizadas em base aos resultados obtidos com a pesquisa; e Considerações Finais, que reúne todas as informações dos capítulos anteriores e as sintetiza em um texto breve e resumido, apresentando também as conclusões obtidas e sugestões de trabalhos futuros.

## 2 REFERENCIAL TEÓRICO

Este capítulo provê um contexto inicial sobre os assuntos que serão abordados no decorrer do trabalho. Os temas abordados são *Frameworks*, *Game Engines*, *Godot Engine* e *GitHub*.

### 2.1 Frameworks

Johnson (1997) aventa que as definições de *frameworks* variam, e descreve sua estrutura como “um design reutilizável de tudo ou parte de um sistema representado por um conjunto de classes abstratas e a forma como suas instâncias interagem” (JOHNSON, 1997, p. 39); indica também que o propósito de um *framework* é de funcionar como “o esqueleto de uma aplicação que pode ser customizada pelo desenvolvedor da aplicação” (JOHNSON, 1997, p. 39).

A essência de um *framework* está no reuso adequado, consistindo em uma aplicação “semi-completa” com uma especialização definida, a fim de criar aplicações personalizadas. Contudo, isso só é possível graças aos componentes providos pela tecnologia de reuso, que permitem uma fácil integração durante a criação de um novo sistema. Esses componentes são encapsulados de forma que o desenvolvedor não necessita de conhecimento sobre sua implementação, somente de como usá-los. Isso facilita o aprendizado e resulta em sistemas mais eficientes, confiáveis e de fácil manutenção (CODENIE et al., 1997; EDWIN, 2014; JOHNSON, 1997; POLITOWSKY et al., 2020).

#### 2.1.1 Características de um *framework*

Existem algumas características que diferem um *framework* de uma biblioteca, sendo elas (EDWIN, 2014):

- **Comportamento padrão:** Antes de realizar um comportamento customizado, um *framework* segue um comportamento específico em relação à interação do usuário (EDWIN, 2014). Existem pontos chamados *hot spots*, que definem em quais partes o *framework* pode ser personalizado (CODENIE et al., 1997), definidos a partir da generalização de funcionalidades de uma aplicação (SCHMID, 1997);

- **Inversão de controle:** Diferente de bibliotecas, o fluxo de controle de um framework não é definido por quem faz a chamada dos métodos específicos, mas sim o próprio framework. Isso permite tratar os eventos e chamadas de forma personalizada (EDWIN, 2014; FAYAD, 1997);
- **Extensibilidade:** Frameworks devem possibilitar ao usuário estender as funcionalidades dos componentes modulares, seja por uso de interfaces, templates ou outra técnica. Combinar esta característica à de inversão de controle, permite que o framework possua métodos que podem ser sobrescritos e posteriormente chamados por funções fixas e pré-definidas do framework (EDWIN, 2014; FAYAD, 1997; JOHNSON, 1997). A característica de extensibilidade pode ser comparada ao Open-Closed dos princípios SOLID, que consiste em manter uma classe aberta para extensão, mas fechada para modificação, a fim de prevenir erros após a implementação de novas funcionalidades (MARTIN, 2014);
- **Código não-modificável do framework:** Os componentes providos pelo framework devem ser suficientes para que o desenvolvedor personalize a aplicação de acordo com os requisitos, de forma simples. O encapsulamento colabora com o aumento da qualidade do software e melhora sua manutenção, ao permitir que o programador utilize os componentes sem que necessariamente entenda sua implementação interna (CODENIE et al., 1997; EDWIN, 2014; FAYAD, 1997; JOHNSON, 1997; SCHMID, 1997).

## 2.2 Game engines

Thorn (2010) descreve uma *game engine* como sendo o “coração” de um jogo, indicando que é uma parte essencial do desenvolvimento. Contudo, é importante considerar que a *game engine* não deve ser o jogo completo, mas sim uma peça importante que permite que o jogo seja desenvolvido, fornecendo componentes e ferramentas aos desenvolvedores (GREGORY, 2018; JOHNSON, 1997; THORN, 2010).

Dado que as *game engines* compartilham similaridades com os frameworks, algumas características em comum podem ser transformadas em componentes, para serem reutilizadas e abarcar diferentes projetos com diferentes propósitos. Conteúdos

como sistema de renderização gráfica, de áudio e de malhas e objetos 3d são exemplos de características providas por *game engines* que podem ser utilizadas em diferentes gêneros de jogos (CODENIE et al., 1997; GREGORY, 2018; POLITOWSKY et al., 2020; SCHMID, 1997; THORN, 2010).

Apesar de existirem *game engines* desenvolvidas por empresas privadas, como a Unity (UNITY, 2024) e a Unreal Engine (UNREAL, 2024), existem outras que são desenvolvidas por comunidades *open-source* (cujo código-fonte é público), como a Godot Engine (GODOT, 2024f) e a Defold (DEFOLD, 2024).

### 2.3 Godot Engine

O *framework* FDCore (documentado no presente trabalho) foi desenvolvido em linguagem GDScript, utilizando a Godot Engine.

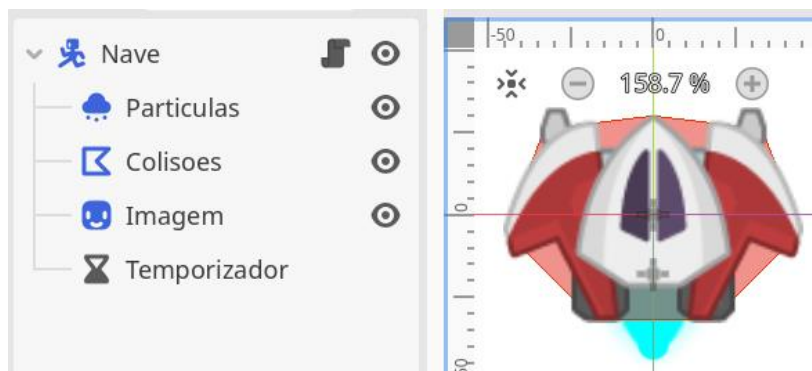
A Godot Engine é uma *game engine* multiplataforma *open-source*, que proporciona ferramentas para a criação de projetos 2D e 3D de todo tipo. Apesar do foco principal ser o desenvolvimento de jogos, a Godot também suporta o desenvolvimento de aplicativos. Os projetos podem ser exportados para as plataformas desktop (Windows, Linux e macOS), *mobile* (Android e iOS), web e consoles (GODOT, 2024i, 2024k).

No presente ano de 2024, o repositório da Godot Engine na plataforma GitHub conta com mais de 2400 contribuidores (sendo a maioria voluntários) que colaboram em diferentes áreas da *engine*, como desenvolvimento, documentação, manutenção de fóruns e gerência de projeto (GITHUB, 2024h; LINIETSKY et al., 2024). Além de usuários, também existem empresas colaboradoras, como por exemplo as empresas Google e The Forge, que contribuíram em otimizações de performance muito úteis para o suporte a gráficos em sistemas mobile (JOHN, 2023), e a empresa Meta, que realizou doações financeiras para contribuir em melhorias dos sistemas de realidade estendida (realidade virtual e aumentada) (VERSCHELDE, 2021).

Tanto os projetos 2D quanto 3D desenvolvidos com a Godot Engine possuem uma estrutura hierárquica em árvore, formada por componentes chamados de “nós” (ou nodes). Os nós são componentes com características específicas e que podem ser expandidos a outras funcionalidades. Os nós também podem ser agrupados em estruturas para formar cenas, que são componentes reutilizáveis. Uma cena pode ser um personagem, um objeto ou até mesmo um cenário completo (GODOT, 2024m). Na Figura 1, é possível notar os seguintes nós e seus respectivos tipos: Nave - CharacterBody2D;

Partículas - GPUParticles2D; Colisoes - CollisionPolygon2D; Imagem - Sprite2D e Temporizador - Timer.

Figura 1 - Estrutura de nós de uma cena na Godot Engine



Fonte: Autoria própria, 2024.

Todos os nós estendem as características do nó base “Node”, porém existem três nós principais que derivam nos demais, sendo eles classificados de acordo ao seu propósito e especialização: os nós especializados em elementos de interface gráfica, como botões, caixas de texto, listas selecionáveis etc., estendem do nó “Control”; os nós relacionados às demais funcionalidades 2D estendem do nó Node2D; e por último, os nós relacionados às funcionalidades 3D estendem do nó Node3D (GODOT, 2024c, 2024j, 2024l, 2024o).

### 2.3.1 GDScript

É possível vincular um script a um nó para estender suas funcionalidades e seu comportamento. Ao fazer isso, o script receberá (por meio de herança) todas as funções e propriedades do nó ao qual está vinculado. Nativamente a Godot oferece suporte às linguagens GDScript e C#, porém a tecnologia GDExtension permite utilizar linguagens externas como por exemplo C, C++, D, Haxe, Rust ou Swift, sem a necessidade de recompilar a *engine*, isso é, gerar o executável da Godot por meio do código fonte e adicionando bibliotecas (GODOT, 2024n, 2024r).

A Godot permite também mesclar linguagens em um mesmo projeto, permitindo uma maior versatilidade e maximizando a otimização ao utilizar por exemplo, GDScript (cuja sintaxe é mais simples) para códigos no geral e C# (cuja performance é melhor) para códigos específicos que possuam uma maior complexidade (GODOT, 2024n).

A estrutura de um código em linguagem GDScript enquadra-se como simples. Similar a linguagens como Python, os escopos são definidos por meio de indentação

(quantidade de tabulações no início da linha). Os identificadores (como nomes de variáveis ou de funções) podem conter somente caracteres alfanuméricos não acentuados maiúsculos ou minúsculos, e o caractere *underscore* ('\_'). Palavras reservadas, como nomes de funções, ou constantes embutidas, não podem ser usadas como identificadores (GODOT, 2024d).

A linguagem GDScript também oferece suporte à sobrescrita de métodos, que por sua vez permite ao programador alterar o comportamento de um nó em um determinado estágio da execução, seja durante a inicialização, inserção do nó na cena ou até mesmo nos loops padrões ou de processamento de físicas. Para sobrescrever um método basta criar uma definição idêntica à anterior, inserindo o comportamento desejado, como exemplificado no Algoritmo 1 (GODOT, 2024d, 2024g).

Algoritmo 1 - Exemplo de sobrescrita de método de um script

```
# Script inimigo.gd

extends CharacterBody2D

v func _process(delta: float) -> void:
  >| var distancia_ao_player: float = global_position.distance_to(Level.player)
  >| if distancia_ao_player < 400: # Atira quando o player estiver perto
  >| >| _atirar() # Chamada do método _atirar

v func _atirar() -> void: # Declaração inicial de _atirar (pode ser sobrescrito)
  >| pass

# Script alien.gd (estende o script inimigo.gd)

extends "res://cenas/inimigo.gd"

const CenaLaser = preload("res://cenas/Laser.tscn")

@onready var arma = get_node("Arma")

v func _atirar() -> void: # Sobreescrever o método _atirar de inimigo.gd
  >| var laser = CenaLaser.instantiate()
  >| laser.direcao = arma.direcao
  >| laser.global_position = arma.global_position
  >| Level.adicionar_laser(laser)
```

Fonte: Autoria própria, 2024.

Alguns métodos são conhecidos como *built-in* e consistem em métodos pré-definidos pela linguagem. Eles podem ser sobrescritos para que seu comportamento seja

alterado, e suas chamadas são realizadas pela *engine* em diferentes momentos da execução:

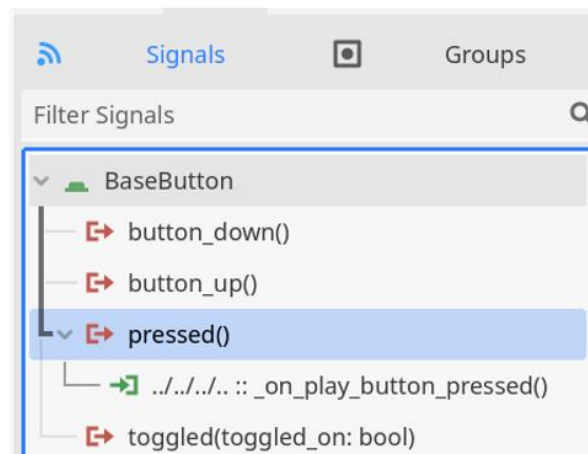
- *\_init*: durante a criação do nó como uma instância, e antes de ser incluído na cena;
- *\_enter\_tree*: durante a inserção do nó na árvore de cena, porém sem aguardar que seus nós filhos sejam inseridos também;
- *\_ready*: quando o nó e todos seus filhos (consequentemente todos seus descendentes) foram inseridos na cena;
- *\_exit\_tree*: durante a remoção do nó da árvore de cena;
- *\_process*: durante o loop de cada frame, e é executado a maior quantidade de vezes possível (depende da capacidade da máquina);
- *\_physics\_process*: é chamada de maneira fixa, normalmente 60 vezes por segundo, de forma independente à taxa de atualização do jogo, permitindo um controle mais suavizado das físicas (GODOT, 2024d, 2024g, 2024h, 2024p, 2024l).

### 2.3.2 *Signals*

Além dos métodos *built-in*, que são chamados em diferentes momentos da execução, existem também aqueles que são executados ao receberem um sinal, de forma assíncrona. Esses sinais são conectados diretamente ao método, e funcionam como gatilhos de ativação. São chamados de *signals*, e estão presentes tanto de forma embutida nos nós, como de forma customizada, mediante a declaração e emissão de um sinal personalizado. Alguns exemplos são um clique de botão, detecção de colisões ou controle de entrada de mouse em uma área visível (GODOT, 2024q).

A Figura 2 e o Algoritmo 3 demonstram, respectivamente, o processo de vinculação e emissão de um sinal, de forma gráfica (por meio da interface gráfica) e por meio de código.

Figura 2 – Conexão do sinal de um nó a um método por meio de interface gráfica



```

v func _on_play_button_pressed() -> void:
  >| get_tree().change_scene_to_file("res://scenes/level.tscn")

```

Fonte: A autoria própria, 2024.

Algoritmo 2 - Conexão do sinal de um nó a um método por meio de código

```

# Código do Player

signal atirou # Criando o sinal 'atirou'

v func _physics_process(delta: float) -> void:
  >| var player_atirou: bool = Input.is_action_just_pressed("atirar")
v >| if player_atirou:
  >| >| atirou.emit() # Emitindo o sinal 'atirou'

# Código do Level

const CenaLaser = preload("res://scenes/lasers/laser_player.tscn")

@onready var player = get_node("Player")
@onready var lasers = get_node("Lasers")

v func _ready() -> void:
  >| >| # Conectando o sinal 'atirou' à função '_criar_laser'
  >| player.atirou.connect(_criar_laser)

v func _criar_laser(posicao: Vector2) -> void:
  >| var laser = CenaLaser.instantiate()
  >| laser.position = posicao
  >| lasers.add_node(laser)

```

Fonte: A autoria própria, 2024.



### 2.3.3 Annotations

Outro recurso útil da Godot Engine são as *annotations*. Elas servem para especificar algum comportamento específico do próprio script, de alguma variável ou de outro recurso inserido no código. Sua notação consiste em inserir o caractere arroba (@) seguido do nome da *annotation* antes do nome do recurso que se deseja afetar, como mostrado no Algoritmo 3 (GODOT, 2024a, 2024d).

Algoritmo 3 - Exemplo do uso de *annotations*

```
@tool # Permite que o nó seja executado no editor
extends Node2D

>| # Permite alterar o valor diretamente pelo editor
@export var vidas: int = 3

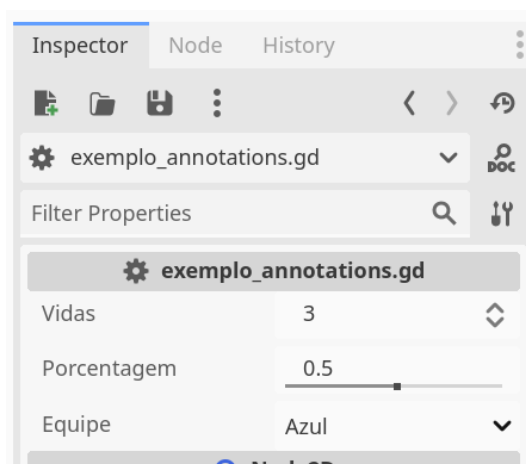
v >| # Permite alterar o valor diretamente pelo editor, limitando o valor
>| # a um intervalo personalizado
@export_range(0.0, 1.0, 0.1) var porcentagem: float = 0.5

v >| # Permite alterar o valor diretamente pelo editor, limitando o valor
>| # a uma lista de valores pré-definidos
@export_enum("Azul", "Vermelho", "Amarelo") var equipe: int = 0

v >| # Atribui o valor somente quando o nó e seus filhos foram adicionados
>| # à cena
@onready var player = get_node("Player")
```

Fonte: Autoria própria, 2024.

Os *annotations* que derivam do *@export*, como *@export\_range*, *@export\_enum*, *@export\_group* e outros, permitem que as variáveis do nó instanciado possam ser editadas diretamente pelo editor, sem a necessidade da criação de um novo script ou modificação do script inicial. Essas edições podem ser feitas por meio da aba Inspector, no editor da Godot Engine, conforme a Figura 3 (GODOT, 2024a, 2024d).

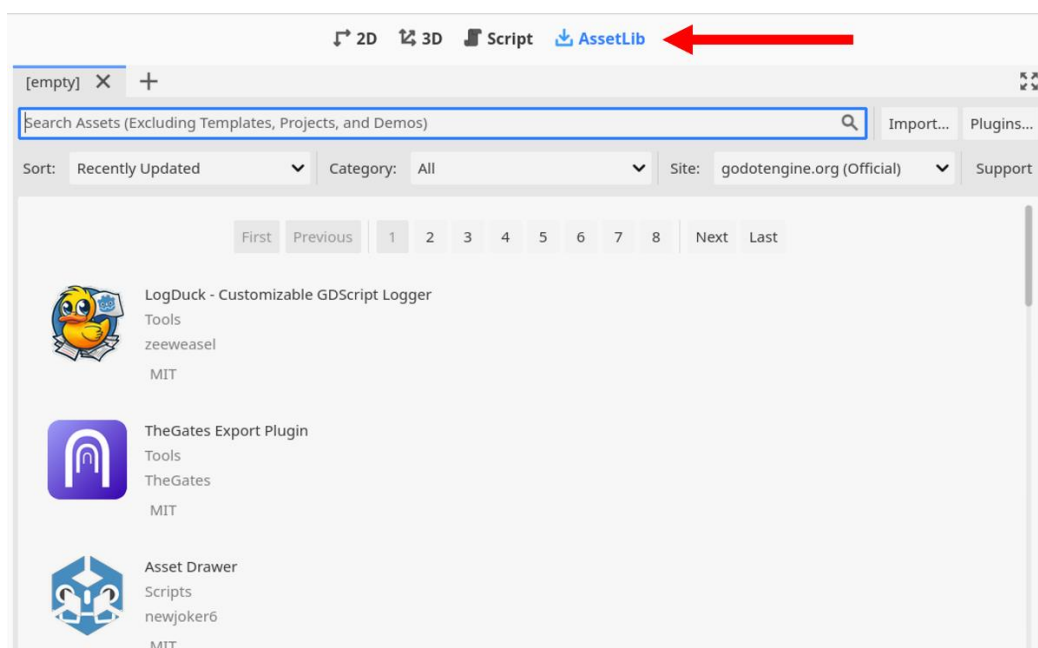
Figura 3 - Exibição de *annotations* na aba *Inspector*

Fonte: Autoria própria, 2024.

### 2.3.4 *AssetLib*

Além de prover os recursos anteriormente citados, a Godot também possui uma biblioteca que permite que a comunidade compartilhe *assets*, que são recursos para o jogo, como scripts, plugins, templates ou recursos multimídia (malhas, áudios, componentes de interface etc.). Os usuários podem fazer download desses recursos diretamente pelo editor, acessando a aba superior de AssetLib (demonstrado na Figura 4) (GODOT, 2024b).

Figura 4 - Visualização da aba AssetLib



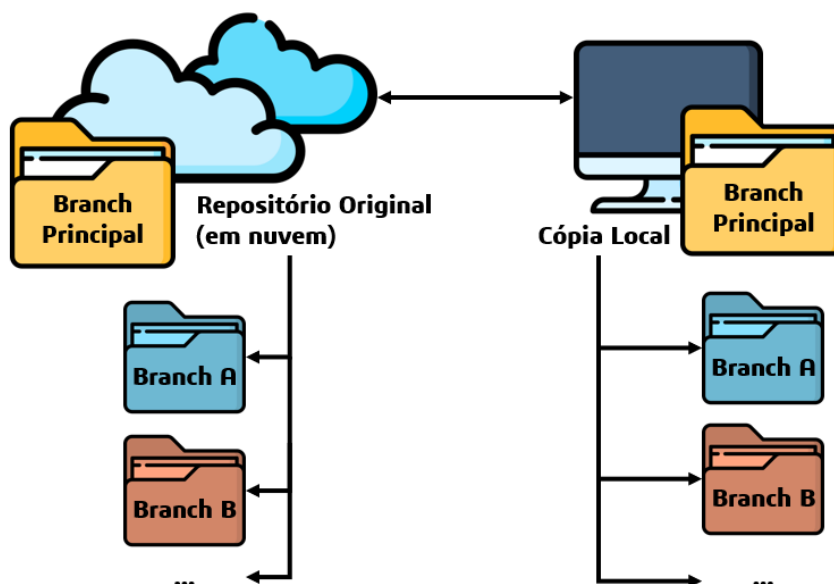
Fonte: Autoria própria, 2024.

## 2.4 GitHub

Além da Godot Engine, existem outros projetos *open source* (código aberto). Diversos desses projetos possuem seu código fonte armazenado em repositórios na plataforma GitHub. De forma simples, o GitHub é uma plataforma baseada na nuvem, que permite aos usuários trabalharem em conjunto, armazenando e compartilhando códigos e arquivos em repositórios Git (GITHUB, 2024m).

Git consiste em um sistema de controle de versionamento, que surgiu inicialmente em 2005, como um sistema de gerenciamento de revisões para o desenvolvimento do *kernel* Linux. O sistema Git permite rastrear e manter registro de todas as mudanças feitas nos arquivos de um repositório, realizar edições individuais e de forma segura aos arquivos, criar cópias de repositórios e ramificações de um repositório, e mesclar várias alterações por diferentes pessoas, sem impactar alterações em andamento (GITHUB, 2024m; SPINELLIS, 2012). A Figura 5 ilustra de forma simplificada a estrutura de repositórios do Git e suas possíveis ramificações, chamadas de *branches*.

Figura 5 - Estrutura de repositórios e ramificações do Git



Fonte: Autoria própria, 2024.

As operações básicas providas pelo sistema Git incluem criação e deleção de repositórios, criação de cópias locais de repositórios já existentes (clonar), criação e deleção de *branches*, atualização do estado de arquivos presentes em uma *branch* local

(*commit*), alterações em uma *branch* remota/em nuvem (*push*), sugestão de modificações em *branches* colaborativas (*pull request*), revisões de sugestões de modificação (*reviews*), mesclagem de alterações em uma *branch* (*merge*), resolução de conflitos entre versões e reversão de alterações baseada no histórico de versões (GITHUB, 2024a, 2024c, 2024d, 2024e, 2024f, 2024i, 2024k; SPINELLIS, 2012).

### 3 MATERIAIS E MÉTODOS

Durante o desenvolvimento desta pesquisa, foram utilizadas as seguintes ferramentas, com suas respectivas versões: Godot Engine v4.1.2 (versão inicial do projeto) e v4.2.1 para a implementação do *framework*, dos projetos de teste e das medições de valores de teste; GitHub Desktop v3.3.14 x64 para operações de versionamento no repositório do projeto; GitHub Web para criação e manutenção do repositório do projeto, incluindo também a disponibilização do projeto como *template* para outros projetos; interface de desenvolvimento Visual Studio Code v1.89.1 e linguagem Python 3.10.12, para a implementação dos scripts utilizados no tratamento dos valores obtidos com os testes; e o software NVIDIA GeForce Experience v3.28.0.412 para obtenção dos valores de uso de hardware durante os testes.

Quanto ao hardware utilizado, foi utilizado um notebook modelo Nitro 5 AN515-57, com sistema operacional Windows 11 Home Single Language (licença embutida no notebook), processador Intel Core i5-11400H de 11ª geração, 24 GB de memória RAM, placa gráfica NVIDIA GeForce RTX 3050 Laptop GPU e armazenamento com 2 SSDs de 512 GB cada. Os softwares utilizados não possuem uma especificação de requisitos mínimos de hardware, contudo, os recursos utilizados foram suficientes para uma execução sem travamentos ou interrupções.

## 4 DESENVOLVIMENTO DO PROJETO

O desenvolvimento deste projeto iniciou com a ideia de um *framework* de comportamentos para nós da Godot Engine, porém evoluiu para um *framework* de gerenciamento completo de projetos, com comunicação facilitada entre as diferentes cenas, e posteriormente incluindo também uma arquitetura específica e uma organização de projetos. A primeira etapa no desenvolvimento foi a identificação dos requisitos, e posteriormente foi realizada a implementação.

### 4.1 Identificando os Requisitos do Projeto

A abordagem inicial do projeto utilizou uma estratégia *bottom-up*, isto é, foram realizadas implementações iniciais sem requisitos definidos explicitamente. Isso permitiu uma melhor compreensão inicial de quais requisitos o projeto necessitaria posteriormente. As primeiras ideias que levaram aos requisitos explícitos do projeto surgiram durante a competição Go Godot Jam 4, com o jogo Masterpiece Madness (código fonte disponível em: <https://github.com/Firemanarg/go-godot-jam-4>) apresentado na Figura 6.

Figura 6 - Tela inicial do jogo Masterpiece Madness



Fonte: Autoria própria, 2024.

Em determinado momento da implementação do jogo, a transição de telas mostrou-se complexa devido à quantidade de scripts que utilizava (um por tela), além da comunicação entre as telas, que era feita somente no momento da troca, com informações armazenadas por um nó mediador visível o tempo todo. Ao inserir transições às trocas de telas, a complexidade aumentou ainda mais, dificultando a manutenção posterior e

levando ao abandono temporário do projeto. Isso permitiu identificar os primeiros requisitos:

- Padronização da estrutura do projeto, para que a troca de telas seja feita por meio de uma estrutura pré-definida;
- Padronização na comunicação entre as diferentes telas por meio de parâmetros explícitos e já esperados;
- Visualização global das informações comunicadas entre as telas, permitindo que outras classes tenham acesso às mensagens para interagir individualmente.

Além dos requisitos mencionados, surgiu também a necessidade de inclusão de transições personalizadas entre telas, portanto um novo requisito:

- Uso de transições durante a troca de telas, por meio de chamadas simples de função e com possibilidade de personalização.

Durante as últimas implementações, foi alcançado um resultado que se mostrou eficiente e permitiu uma melhoria considerável na organização do projeto: organizar a estrutura em camadas, conforme mostrado na Figura 7. Assim, surgiu mais um requisito que posteriormente levaria à arquitetura do projeto:

- Divisão da estrutura do projeto em camadas com propósitos específicos.

Com a estrutura do projeto realizada, foram realizados testes com a nova implementação, e os resultados foram satisfatórios. Contudo, surgiu também uma nova necessidade: a comunicação era feita conforme o Algoritmo 4 e havia comunicação padronizada entre as telas, porém era necessário um prefixo em toda mensagem enviada, para especificar sua origem. Com isso, um novo requisito se apresentou:

- Personalização de mensagens durante a comunicação entre as diferentes telas, porém incluindo também o contexto da mensagem, isto é, sua origem.

Algoritmo 4 - Código de emissão de ações de clique no jogo Masterpiece Madness

```
# Arquivo main_screen.gd

# Sinal de botão pressionado, passando o nome de uma ação
signal button_pressed(action_name: String)

▼ func _on_button_play_pressed():
  >| button_pressed.emit("play") # Emitindo a ação "play"

▼ func _on_button_settings_pressed():
  >| button_pressed.emit("settings") # Emitindo a ação "settings"

▼ func _on_button_credits_pressed():
  >| button_pressed.emit("credits") # Emitindo a ação "credits"
```

Fonte: Autoria própria, 2024.

Posteriormente, durante a competição Game Off 2023, o projeto Evolution (código fonte disponível em: <https://github.com/Firemanarg/game-off-2023>) utilizou recursos da API Nakama para implementar a conexão multijogador e um sistema de salas com código único. Devido à quantidade de funções necessárias para realizar as conexões, foram desenvolvidas funções abstratas que reduziram a quantidade de código. Essas funções foram agrupadas em uma única classe, seguindo um padrão de design conhecido como *Singleton*, que consiste em uma classe que possui somente uma instância em todo o projeto, porém visível no escopo global (acessível em qualquer código do projeto).

Os Algoritmos 5 e 6 mostram a chamada da função *join\_match* e sua respectiva implementação no código de um *singleton*. É notável a diferença na quantidade de linhas.

Algoritmo 5 - Chamada de uma função de *singleton*, no projeto Evolution

```
# Arquivo test_screen.gd

# Código para se juntar a uma partida já existente, passando um código
▼ func _on_button_join_match_pressed() -> void:
  >| var match_code: String = %LineEditMatchCode.text
  >| Online.debug_print("button_pressed", "Pressed 'join_match'")
  >| await OnlineMatch.join_match(match_code) # Chamada da função do singleton OnlineMatch
```

Fonte: Autoria própria, 2024.



Algoritmo 6 - Código da função *join\_match* em um *singleton*, no projeto Evolution

```

# Arquivo online_match.gd

# Método do singleton OnlineMatch
▼ func join_match(match_code: String) -> void:
  >| var payload: Dictionary = {"match_code" : match_code}
  >| var response = await Online.call_rpc_func("get_match_id_by_code", payload)
  ▼ >| if response.is_exception():
    >| >| Online.debug_print("join_match", "Error: " + response.get_exception().message)
    >| >| match_ = null
    >| >| match_join_failed.emit()
    >| >| return
    >| var json = JSON.parse_string(response.payload)
    >| var match_id: String = json.get("match_id", "")
  ▼ >| if match_id.is_empty():
    >| >| Online.debug_print("join_match", "Error retrieving match_id")
    >| >| match_ = null
    >| >| match_join_failed.emit()
    >| >| return
    >| var joined: bool = await _join_match_by_id(match_id, true)
  ▼ >| if not joined:
    >| >| match_join_failed.emit()
    >| >| match_ = null
    >| >| return
  >| match_joined.emit()

```

Fonte: Autoria própria, 2024.

Sendo assim, foi observado que o uso de *singletons* se adequa à personalização e portanto, foi definido mais um requisito:

- Implementar uma classe gerenciadora global que conterà os métodos mais importantes, como trocas de telas ou gerenciamentos que possam ocorrer durante a execução.

Outro aspecto observado em ambos os projetos mencionados foi a organização dos arquivos, que permitiu separar os arquivos por finalidade (recursos, cenas, *assets* etc.). Isso levou à formulação de mais um requisito:

- Definir uma organização dos arquivos do projeto em diretórios, permitindo agrupá-los por finalidade específica.

E por último, dado que o projeto em questão foi desenvolvido para auxiliar nos projetos de um estúdio específico, surgiu a necessidade de que todo projeto criado tenha por padrão a introdução (*intro*) contendo a logo da engine (Godot Engine) e da empresa (Fire-Droid Game Studios):

- Exibir introduções animadas das logo da Godot Engine e da Fire-Droid Game Studios no início de cada projeto, de forma automática;
- Implementar um modo *debug* que desabilite as introduções animadas para a realização de testes unitários.

Sendo assim, se tem como lista completa de requisitos, baseados nas necessidades levantadas e anteriormente mencionadas adicionadas aos requisitos não funcionais:

#### **4.1.1 *Requisitos funcionais***

1. Padronização da estrutura do projeto, para que a troca de telas seja feita por meio de uma estrutura pré-definida;
2. Padronização na comunicação entre as diferentes telas por meio de parâmetros explícitos e já esperados;
3. Visualização global das informações comunicadas entre as telas, permitindo que outras classes tenham acesso às mensagens para interagir individualmente;
4. Uso de transições durante a troca de telas, por meio de chamadas simples de função e com possibilidade de personalização;
5. Divisão da estrutura do projeto em camadas com propósitos específicos;
6. Personalização de mensagens durante a comunicação entre as diferentes telas, porém incluindo também o contexto da mensagem, isto é, sua origem;
7. Implementar uma classe gerenciadora global que conterà os métodos mais importantes, como trocas de telas ou gerenciamentos que possam ocorrer durante a execução;
8. Definir uma organização dos arquivos do projeto em diretórios, permitindo agrupá-los por finalidade específica;
9. Exibir introduções animadas das logo da Godot Engine e da Fire-Droid Game Studios no início de cada projeto, de forma automática;
10. Implementar um modo *debug* que desabilite as introduções animadas para a realização de testes unitários.

### 4.1.2 *Requisitos não funcionais*

1. Acelerar o processo de desenvolvimento de jogos;
2. Garantir um desempenho próximo ou melhor ao da execução de projetos sem o *framework*;
3. Reduzir a quantidade de etapas para a criação de novos projetos;
4. Permitir um melhor reuso de componentes em projetos futuros;
5. Diminuir a quantidade necessária de scripts por projeto;
6. Permitir que o programador foque o fluxo de execução do projeto em um único script gerenciador;
7. Reduzir a curva de aprendizado de colaboradores para diferentes projetos;
8. Facilitar a manutenção de projetos que utilizem o *framework*;
9. Minimizar os custos e o tempo necessários por projeto, ao padronizar a estrutura e definir uma organização prévia que permita reuso facilitado;

## 4.2 **Arquitetura do projeto**

O projeto utiliza uma arquitetura baseada em camadas especializadas, com propósitos específicos e que são geradas somente durante a execução. Algumas são somente de uso interno do *framework*, e não podem ser acessadas diretamente pelo programador. As camadas presentes na arquitetura são:

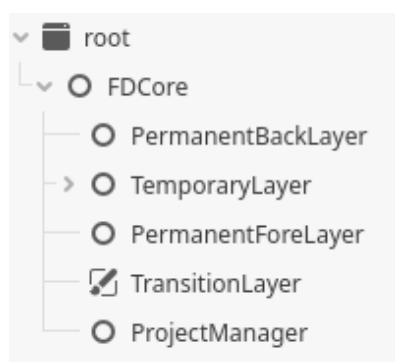
- **PermanentBackLayer**: atua como uma camada permanente. Os nós presentes nesta camada não serão removidos durante as trocas de cena e serão sempre desenhados por trás dos nós das demais camadas;
- **TemporaryLayer**: atua como uma camada temporária. Durante as trocas de cena, os nós presentes nesta camada serão removidos e a nova cena será adicionada à camada;
- **PermanentFrontLayer**: atua de forma similar à camada **PermanentBackLayer**, diferindo somente em que seus nós serão desenhados acima da camada temporária;
- **TransitionLayer**: atua unicamente como camada de transição, exibindo as transições e sobrepondo as demais camadas durante o processo. Esta camada serve somente para uso interno do *framework*, não deve ser acessada diretamente pelo programador.

Além das camadas citadas, o framework conta também com dois gerenciadores:

- **FDCore**: gerenciador global. É definido como um *singleton* e possui uma visão total do escopo do projeto. Este gerenciador possui as funções essenciais para o uso do *framework*, e atua como um núcleo;
- **ProjectManager**: é o gerenciador de projeto. Seu comportamento é totalmente personalizável e deve ser definido antes da execução do código, herdando da classe `FDProjectManager`.

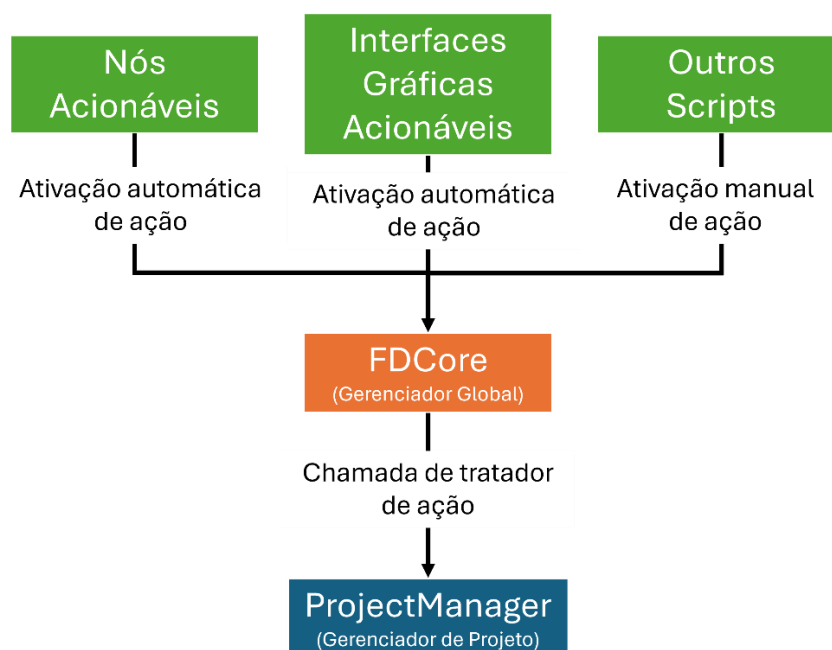
A Figura 7 mostra a estrutura da cena principal durante a execução de um projeto utilizando o *framework*.

Figura 7 - Estrutura da cena principal de um projeto utilizando o *framework* FDCore



Fonte: Autoria própria, 2024.

Para padronizar a comunicação entre diferentes cenas durante a execução, é utilizado um sistema de troca de mensagens por ativadores de ação, chamados de `ActionTriggers`. Esse sistema sinaliza ao gerenciador global sempre que uma ação for ativada, indicando a ação e seu contexto (origem). Em seguida, o gerenciador global sinaliza ao gerenciador de projeto para que alguma decisão seja tomada. Essa tomada de decisão é definida pelo programador dentro do código do gerenciador de projeto. O processo de ativação de ações pode ser visualizado na Figura 8.

Figura 8 - Representação do sistema de ativação de ações do *framework* FDCore

Fonte: Autoria própria, 2024.

### 4.3 Apresentação do framework

Nesta seção será apresentada somente a estrutura das classes, sendo mostrados os métodos e atributos públicos de forma simplificada e ilustrativa. Contudo, em apêndice estão presentes os diagramas de classe detalhados, contendo métodos e atributos públicos e privados. Também é possível acessar a documentação completa pela Godot Engine, abrindo a aba “Ajuda”, selecionando a opção “Pesquisar Ajuda” e selecionando a classe da qual se deseja acessar a documentação.

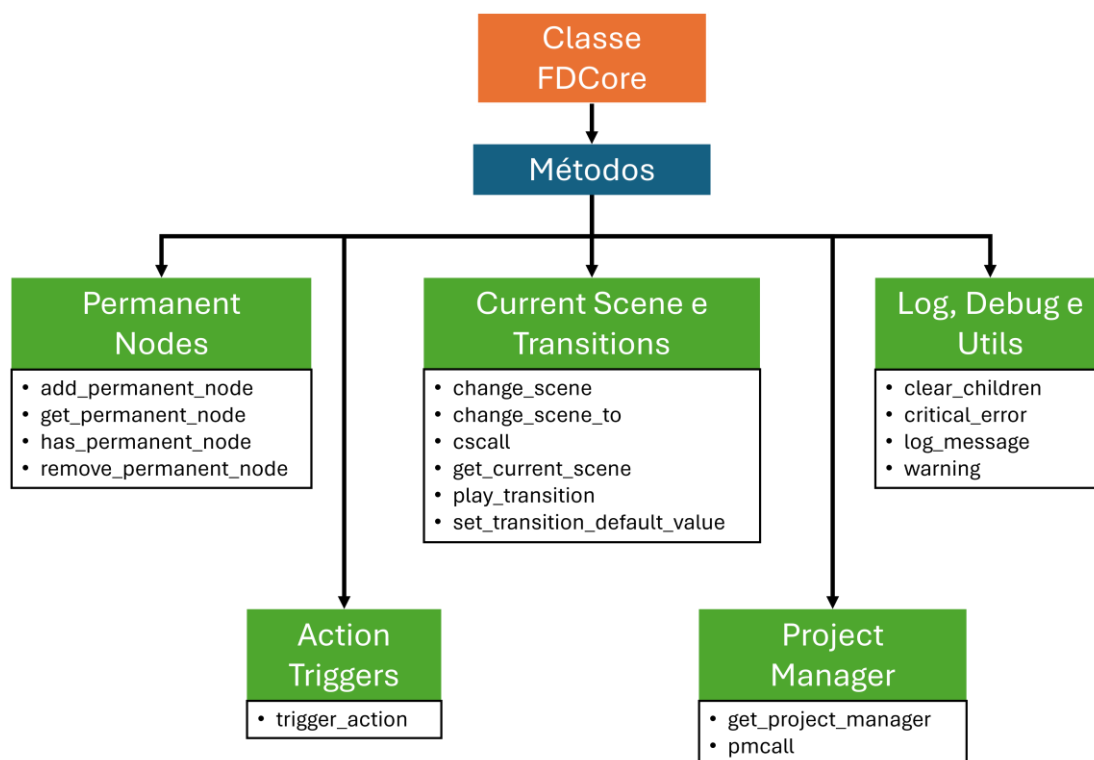
Quanto à estrutura do *framework*, está formada por classes independentes que se relacionam entre si, e se comunicam por meio de uma classe núcleo chamada FDCore. As classes existentes no projeto são:

- FDCore
- FDProjectManager
- Transition
- LogoIntro
- ActionHUD
- ActionButton
- SignalActionTrigger

### 4.3.1 FDCore

A classe FDCore é um *singleton* que possui visão do escopo global e acesso direto ao gerenciador do projeto. Esta classe atua como um núcleo, criando a estrutura completa do projeto (seguindo a arquitetura da Figura 7) no início da execução. Possui também métodos auxiliares para a troca de telas, gerenciamento de nós permanentes e realiza a mediação e o tratamento das mensagens transmitidas durante a ativação de ações. Para uma fácil visualização, os métodos públicos foram agrupados em seções de acordo a sua funcionalidade, conforme mostrado na Figura 9.

Figura 9 - Métodos públicos da classe FDCore, organizados em seções



Fonte: Autoria própria, 2024.

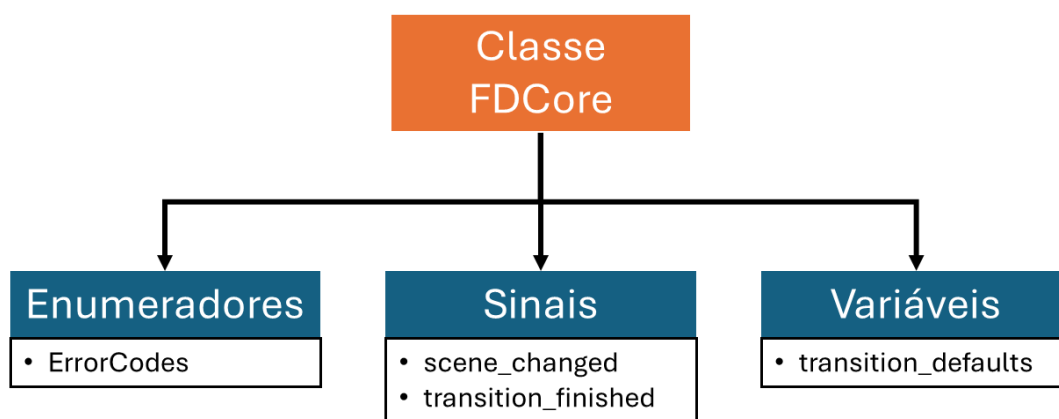
As seções apresentadas na figura 9 podem ser descritas por:

- **Permanent Nodes:** lida com o gerenciamento de nós pertencentes às camadas permanentes (PermanentBackLayer e PermanentFrontLayer), permitindo adicionar, remover, obter e verificar nós;

- **Current Scene e Transitions:** lida diretamente com as trocas de cenas e exibição de transições. Apesar das transições serem realizadas automaticamente ao trocar para uma cena nova, é possível exibir uma transição de forma manual (sem a necessidade de troca de cena);
- **Log, Debug e Utils:** consiste nos métodos utilizados para exibir mensagens no terminal, seja para criar registros quanto para facilitar o *debug* do projeto. Esta seção também contém métodos úteis para o programador, como por exemplo, remover todos os filhos de um nó por meio de uma única chamada de método;
- **Action Triggers:** trata diretamente da parte de ativação de ações. Permite ao programador ativar manualmente uma ação;
- **Project Manager:** os métodos contidos nesta seção estão propriamente relacionados ao gerenciador de projeto, permitindo obter uma referência ao objeto do gerenciador ou fazer chamada direta a algum de seus métodos.

Quanto aos atributos públicos da classe, estão listados na Figura 10 os enumeradores, sinais e variáveis.

Figura 10 - Atributos públicos da classe FDCore



Fonte: Autoria própria, 2024.

#### 4.3.2 *FDProjectManager*

É mandatório que todo projeto utilizando o *framework* possua um gerenciador de projeto herdado da classe *FDProjectManager*. Essa classe possui métodos virtuais que devem ser implementados no gerenciador de projeto, pois são essenciais para o gerenciamento e tratamento de mensagens durante as ativações de ações. Quanto a seus

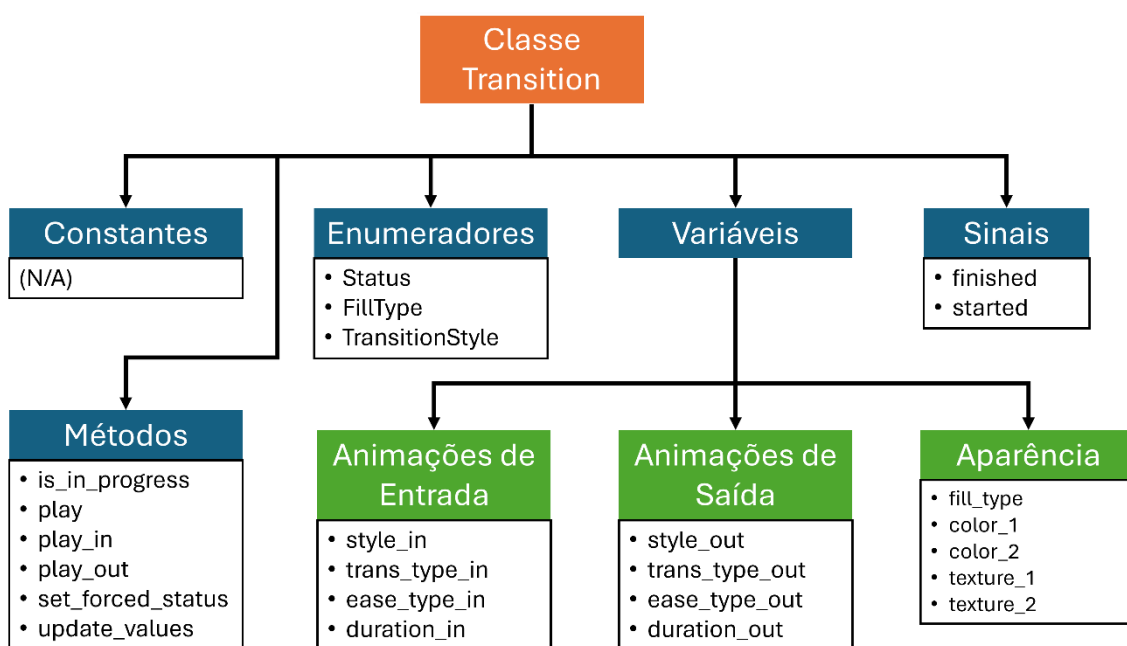
métodos virtuais, existe somente o método `_on_action_triggered`, o qual é chamado sempre que a classe núcleo `FDCore` transmite uma ativação de ação, e quanto a seus atributos, existe somente a variável `initial_scene`, que deve ser definida no método `_init`.

### 4.3.3 Transition

O *framework* possui uma classe chamada `Transition`, que é especializada em criar efeitos de transição com possibilidade de personalização. O programador pode usar essas transições durante a troca de tela (de forma implícita e automática) ou de forma manual, para que seja possível proporcionar ao jogador uma melhor experiência visual durante as mudanças do que está sendo exibido na tela (mudança de cenário, troca de gameplay para animações cinemáticas, abertura de inventário etc.).

Os métodos e atributos públicos presentes na classe estão listados na Figura 11, tendo sido as variáveis agrupadas por seções de acordo a suas respectivas finalidades.

Figura 11 - Métodos e atributos públicos da classe `Transition`



Fonte: Autoria própria, 2024.

As seções nas quais as variáveis foram agrupadas possuem as seguintes finalidades:



- Animações de Entrada: relativa às variáveis que definem o comportamento das animações de início da transição (aparecimento), sendo seu estilo de transição, tipo de curva de transição, tipo de interpolação e duração da animação;
- Animações de Saída: possui praticamente a mesma finalidade da seção Animações de Entrada, com diferença que suas variáveis estão relacionadas à animação de finalização da transição (desaparecimento);
- Aparência: está diretamente relacionada ao aspecto da transição. É possível definir se o preenchimento de uma transição será com uma cor sólida ou uma imagem, e permite personalizar esse preenchimento.

Um detalhe importante é que a classe `FDCore` cria uma instância da classe `Transition` sempre que há uma mudança de tela ou quando o método `play_transition` é chamado. Contudo, caso o programador deseje realizar o processo de transição manualmente utilizando a classe, é possível fazê-lo seguindo os passos descritos na documentação do *framework*.

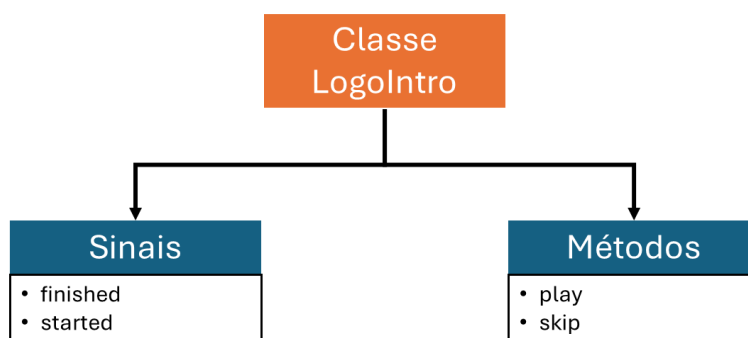
#### **4.3.4 LogoIntro**

No início da execução da classe `FDCore`, é criada a estrutura de projeto e posteriormente são exibidas as telas. No início dessa exibição de telas, primeiramente são exibidas duas introduções com logos animadas, sendo a primeira da Godot Engine e a segunda do estúdio Fire-Droid Game Studios. Ambas as animações herdam da cena `LogoIntro`, que possui como nó raiz a classe de mesmo nome (`LogoIntro`). Ambas dispensam ajustes por parte do programador, e são utilizadas somente no início da execução.

Apesar de não necessitar configuração para funcionar inicialmente, a classe `LogoIntro` permite que o programador crie suas próprias logos animadas e as exiba na tela. Para isso, deve ser criada uma cena herdando do arquivo localizado no caminho “`res://addons/fire_droid_core/scenes/logo_intro/logo_intro.tscn`”. A seguir, devem ser definidos os frames da animação alterando a propriedade `frames` pelo próprio editor da Godot, e caso a animação tenha som, deve ser definido na propriedade `audio_stream`, também pelo editor.

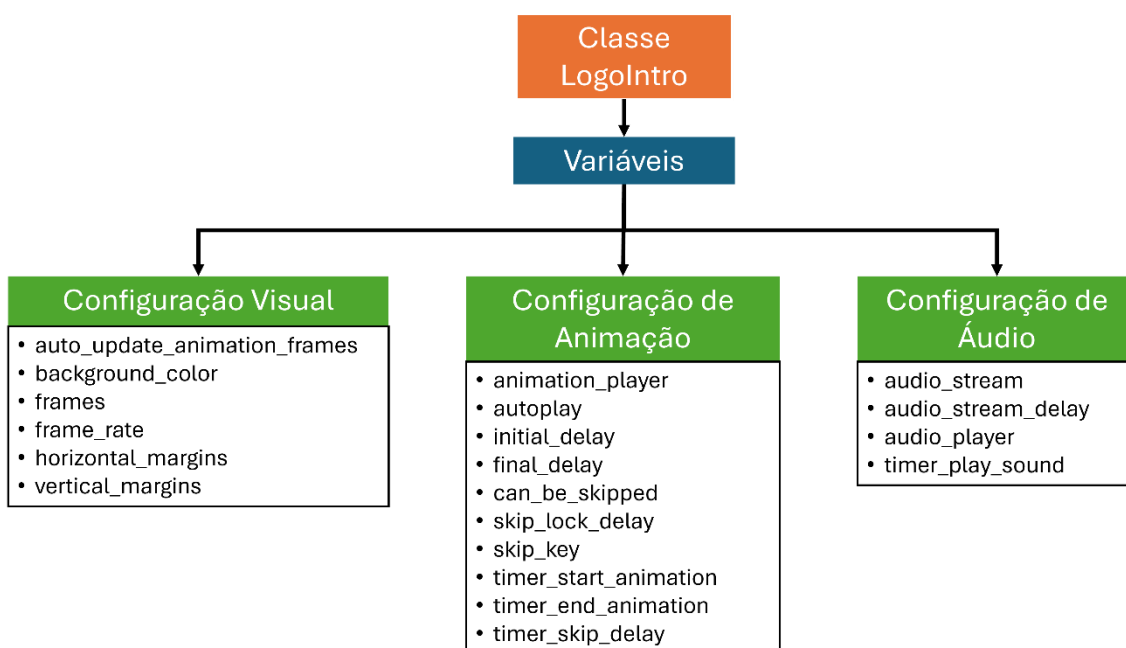
Os métodos e atributos públicos presentes na classe `LogoIntro` estão representados nas Figuras 12 e 13.

Figura 12 - Métodos e atributos públicos da classe LogoIntro



Fonte: Autoria própria, 2024.

Figura 13 - Variáveis públicas da classe LogoIntro



Fonte: Autoria própria, 2024.

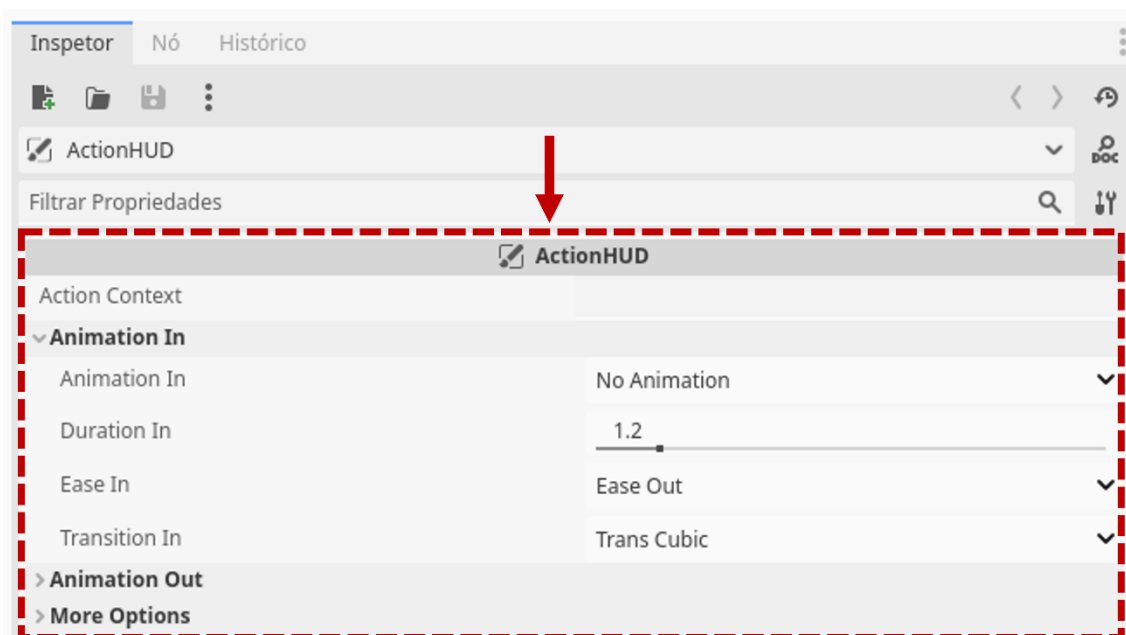
Na Figura 13, as variáveis foram agrupadas em seções, de acordo a seus propósitos:

- Configuração Visual: métodos utilizados para criar as animações de forma automática, ajustar o tamanho da logo na cena e alterar a cor do *background*;
- Configuração de Animação: relativo aos métodos utilizados para definir o comportamento da animação, como atraso inicial ou final, configuração de tecla para pular a animação, entre outros;
- Configuração de Áudio: possui métodos utilizados para configurar o áudio durante a animação, para que esteja sincronizado com os frames.

### 4.3.5 ActionHUD

Esta classe consiste na base utilizada para todas as interfaces que utilizem o sistema de ativação de ações do *framework*. Um ActionHUD permite ao programador definir um contexto, que serve como origem da mensagem enviada durante a ativação de ações. É possível customizar animações da interface diretamente pelo editor, sem a necessidade de código, conforme mostrado na Figura 14.

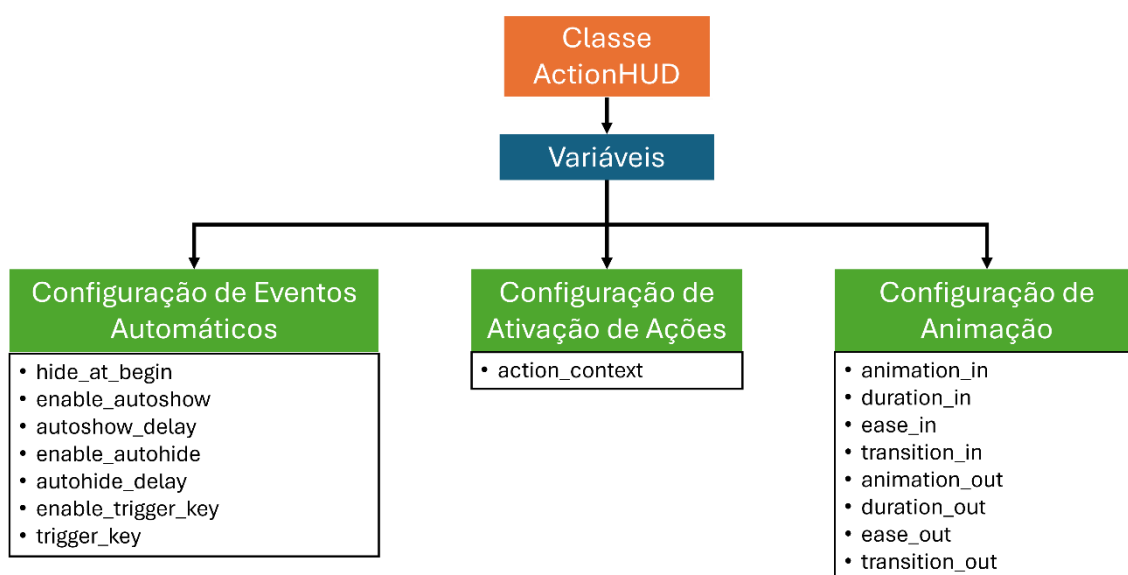
Figura 14 - Customização de animações de um nó do tipo ActionHUD



Fonte: Autoria própria, 2024.

A Figura 15 representa as variáveis públicas da classe ActionHUD, agrupadas em seções de acordo a suas funcionalidades.

Figura 15 - Variáveis públicas da classe ActionHUD



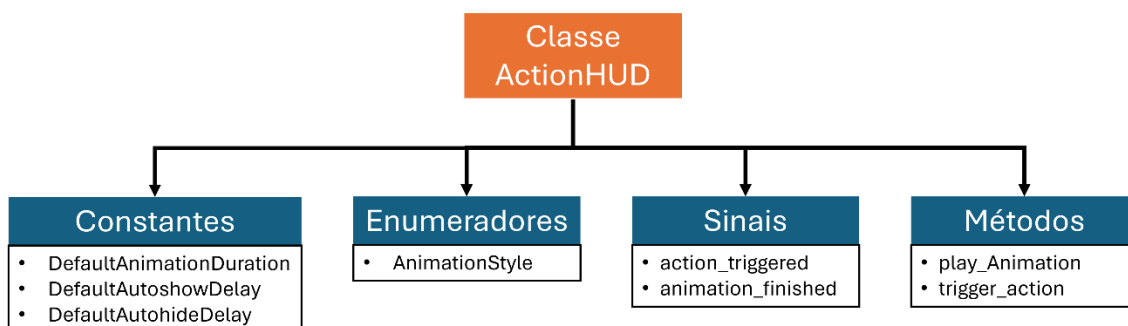
Fonte: Autoria própria, 2024.

As seções da figura 15 estão organizadas da seguinte forma:

- **Configurações de Eventos Automáticos:** possui as variáveis utilizadas para os ajustes de eventos automáticos, como esconder a interface no início, e mostrar ou esconder a interface automaticamente após um tempo ou por meio do pressionamento de alguma tecla;
- **Configurações de Ativação de Ações:** engloba a variável que define o contexto das ações que serão ativadas pela interface;
- **Configuração de Animação:** inclui as variáveis necessárias para controle das animações da interface (mostrar ou esconder), definindo propriedades como tipo da animação, duração, tipo de transição e de interpolação.

Quanto aos métodos e demais atributos públicos da classe, estão descritos na Figura 16.

Figura 16 - Atributos e métodos públicos da classe ActionHUD

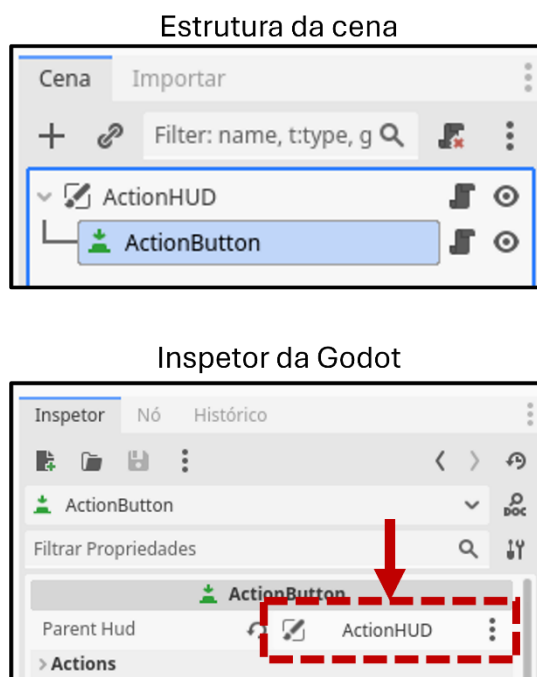


Fonte: Autoria própria, 2024.

#### 4.3.6 ActionButton

A classe ActionButton pode ser utilizada para criar botões em interfaces gráficas que tenham um ActionHUD como nó raiz, dispensando o uso de códigos para integração com ativação de ações. É exigido que esse nó raiz seja definido pelo Inspetor da Godot, conforme mostrado na Figura 17.

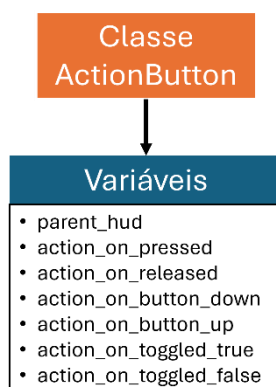
Figura 17 - Definindo o nó raiz de interface gráfica em um nó do tipo ActionButton



Fonte: Autoria própria, 2024.

Além de integração com interfaces gráficas herdadas de ActionHUD, a classe ActionButton também permite personalizar a ação que será ativada em cada evento, sem a necessidade de código. A Figura 18 lista as variáveis para os possíveis eventos de um botão. A classe não possui atributos além das variáveis, tampouco métodos.

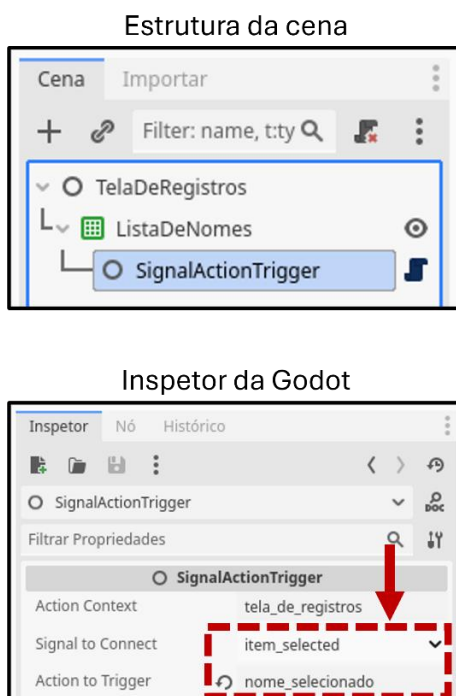
Figura 18 - Possíveis ações customizáveis dos eventos da classe ActionButton



Fonte: A autoria própria, 2024.

#### 4.3.7 *SignalActionTrigger*

Para evitar o uso de código para vincular um sinal a uma ativação de ação, de qualquer que seja o nó, é possível utilizar um nó do tipo *SignalActionTrigger*. Esta classe permite a vinculação diretamente pelo Inspetor, sem a necessidade de incorporar scripts adicionais, conforme demonstrado na Figura 19.

Figura 19 - Conexão de um sinal a uma ação, utilizando o nó *SignalActionTrigger*

Fonte: A autoria própria, 2024.

A classe *SignalActionTrigger* possui somente três variáveis públicas, sendo elas *action\_context* (contexto da ação que será ativada), *signal\_to\_connect* (sinal do nó que

será conectado à ação) e *action\_to\_trigger* (ação que será ativada). A classe não possui métodos ou outros atributos.

#### 4.4 Repositório do framework

O *framework* desenvolvido possui código aberto. Isso significa que seu código-fonte está disponível publicamente, permitindo que qualquer usuário o utilize e faça suas próprias modificações de acordo à necessidade. Os arquivos que conformam o *framework* estão disponíveis no repositório do GitHub, com acesso mediante o link <https://github.com/FireDroidGameStudios/godot-project-template>.

Ao acessar a página do repositório, será exibida a estrutura de arquivos que conformam o projeto. Essa estrutura está organizada conforme a Figura 20.

Figura 20 - Organização da estrutura do repositório do framework no GitHub

```
godot-project-template/  
├── addons/  
│   ├── fire_droid_core/  
├── assets/  
│   ├── audios/  
│   │   ├── musics/  
│   │   └── sounds/  
│   ├── fonts/  
│   ├── images/  
│   │   ├── common_textures/  
│   │   ├── icons/  
│   │   ├── sprites/  
│   │   └── tiles/  
│   └── meshes/  
├── prefabs/  
├── resources/  
│   ├── materials/  
│   ├── shaders/  
│   └── styles/  
└── scenes/
```

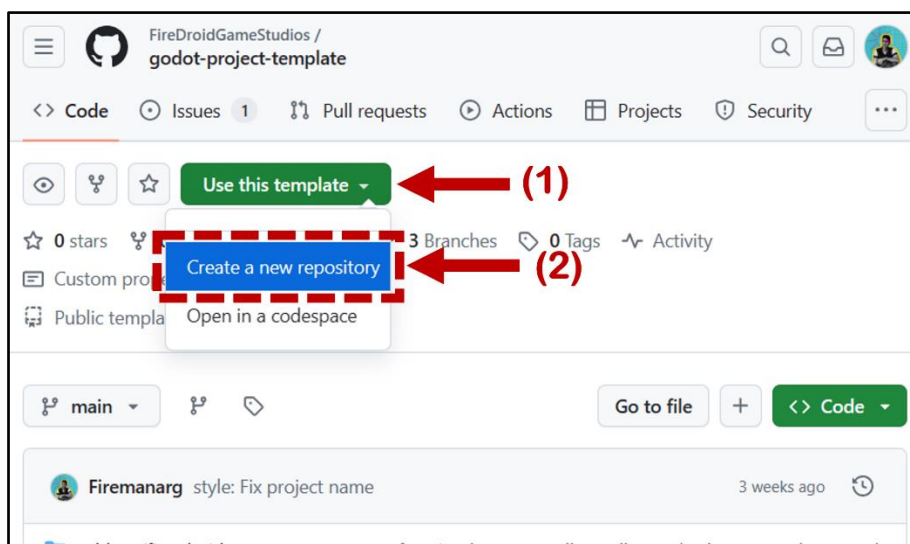
Fonte: Autoria própria, 2024.

Apesar do repositório do *framework* estar organizado como na Figura 20, a organização é somente uma sugestão de padronização, sendo necessário somente o diretório “/addons/fire\_droid\_core” e seus arquivos para o correto funcionamento do *framework*. Os diretórios listados podem ser descritos por:

- *addons*: possui os plugins instalados no projeto, normalmente baixados da biblioteca AssetLib da Godot Engine;
- *assets*: contém os ativos que serão utilizados no jogo, como áudios, fontes, ícones, texturas, malhas etc.;
- *prefabs*: inclui cenas pré-fabricadas que serão utilizadas posteriormente como modelos para que instâncias sejam criadas;
- *resources*: abrange os recursos em formatos oferecidos pela Godot, como materiais (2D e/ou 3D), scripts de efeitos visuais, estilos de nós etc.
- *scenes*: está composto pelas demais cenas utilizadas no projeto, como interfaces, níveis, menus, gerenciadores abstratos etc.

O repositório do *framework* foi configurado como um *template*, permitindo que qualquer usuário do GitHub possa criar seus próprios projetos seguindo a organização da Figura 20 diretamente pela versão web da plataforma. Para isso, será necessário acessar a página do repositório e criar um repositório utilizando o *template*, como mostrado na Figura 21.

Figura 21 - Utilizando o repositório do framework no GitHub como template

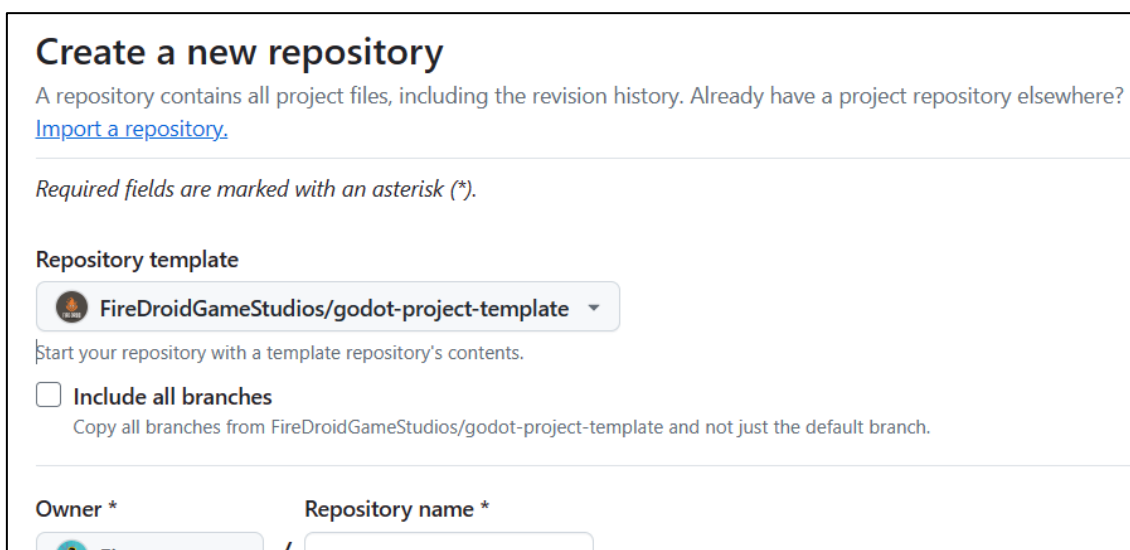


Fonte: Autoria própria, 2024.

Ao criar um novo repositório, será exibida a tela de configurações da Figura 22. Após configurado e criado, o repositório será criado com a organização do *template*.



Figura 22 - Tela de criação de repositório no GitHub, utilizando template




**Create a new repository**

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (\*).

**Repository template**

 FireDroidGameStudios/godot-project-template ▾

Start your repository with a template repository's contents.

**Include all branches**  
Copy all branches from FireDroidGameStudios/godot-project-template and not just the default branch.

**Owner \*** **Repository name \***

Fonte: Autoria própria, 2024.

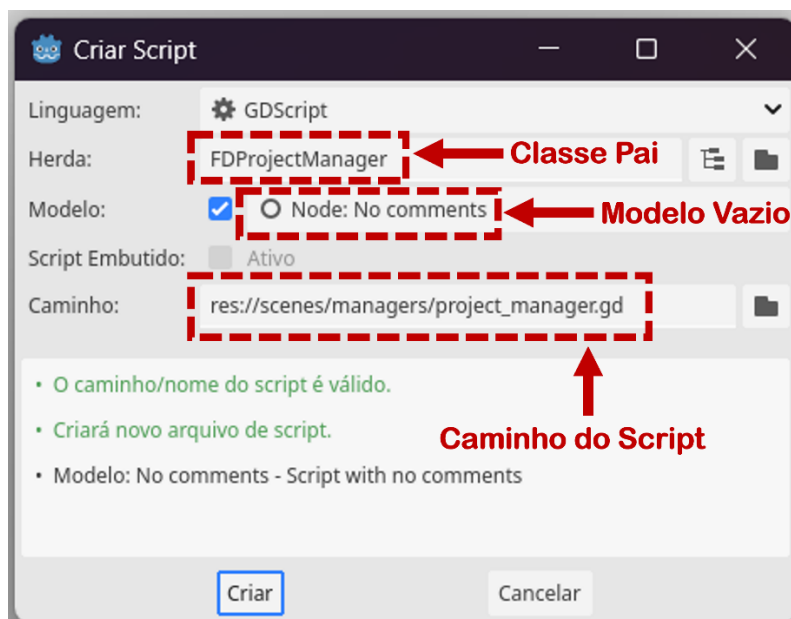
Uma vez criado o repositório, deve-se cloná-lo localmente e então abri-lo na Godot. Caso o usuário opte por não seguir a arquitetura sugerida, deverá remover todos os diretórios com exceção do diretório “*addons*”.

#### 4.5 Configuração de novo projeto utilizando o framework

Para facilitar a compreensão acerca do uso do *framework*, nesta seção será demonstrada a criação de um projeto de exemplo. Para isso, será considerando que o projeto já foi criado previamente utilizando o *template* disponível no GitHub, e foi aberto no editor da Godot Engine.

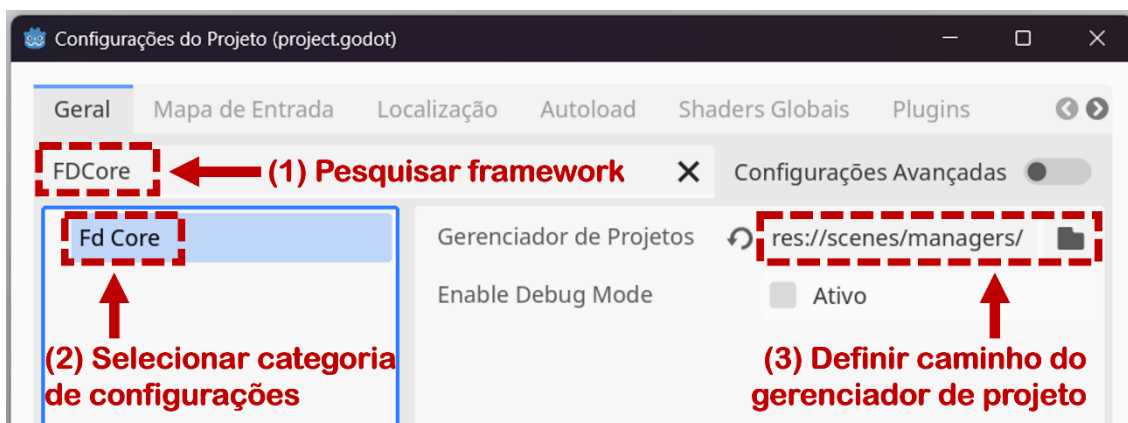
A recomendação é de que inicialmente se configure o gerenciador de projeto. Para isso, deverá ser criado um script que herde da classe `FDProjectManager`, conforme a Figura 23, e então referenciar sua localização na janela de Configurações de Projeto, de acordo com a figura 24. Esse script pode ser implementado pelo programador para definir o fluxo de tratamento das ações, sendo necessário sobrescrever o método `_on_action_triggered`.

Figura 23 - Criação de novo script gerenciador de projeto



Fonte: Autoria própria, 2024.

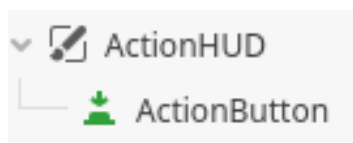
Figura 24 – Especificando caminho do script gerenciador de projeto



Fonte: Autoria própria, 2024.

Após ter criado e configurado o gerenciador de projeto, serão criadas duas cenas para exemplificar a troca de telas, sendo elas a Tela Principal e a Tela de Créditos. Ambas possuindo o nó raiz do tipo ActionHUD, tal qual exibido na Figura 25.

Figura 25 - Exemplo de estrutura de cena de tela

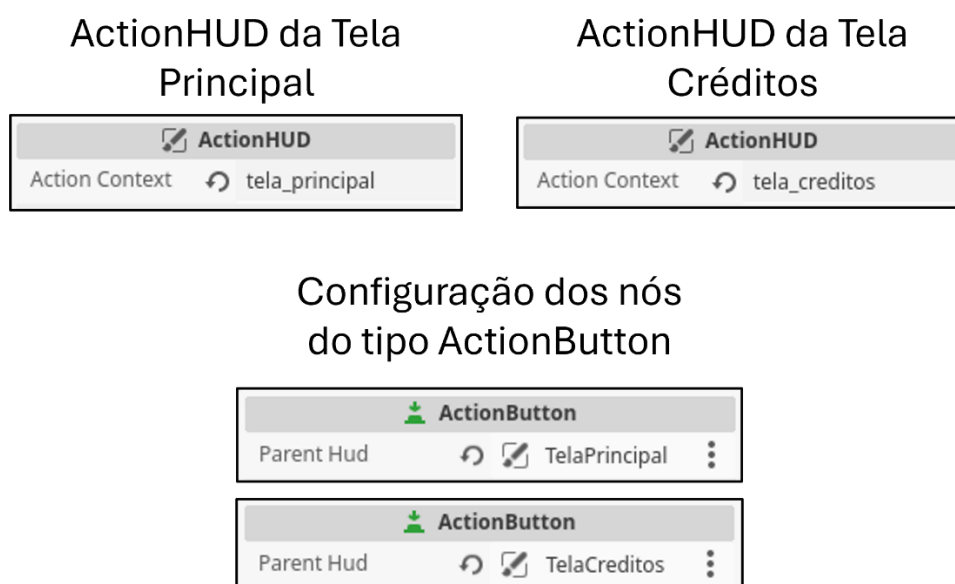


Fonte: Autoria própria, 2024.

As cenas de tela possuirão dois botões cada, sendo “Créditos” e “Sair” pertencentes à Tela Principal, e “Voltar” e “Sair” pertencentes à Tela Créditos. Para que os botões emitam uma ação ao serem clicados, serão criados como `ActionButton`, simplificando assim a integração com o *framework*.

Ao criar um botão do tipo `ActionButton`, é necessário que seja definido um nó raiz do tipo `ActionHUD`, que atuará como um transmissor de mensagens para o ativador global de ações. Os nós raiz de cada cena também devem indicar qual o seu contexto, para que seja especificada a origem das ações que esses nós ativarem. A Figura 26 ilustra a configuração dos contextos dos nós de tipo `ActionHUD`, e das referências às respectivas raízes de cena dentro dos botões.

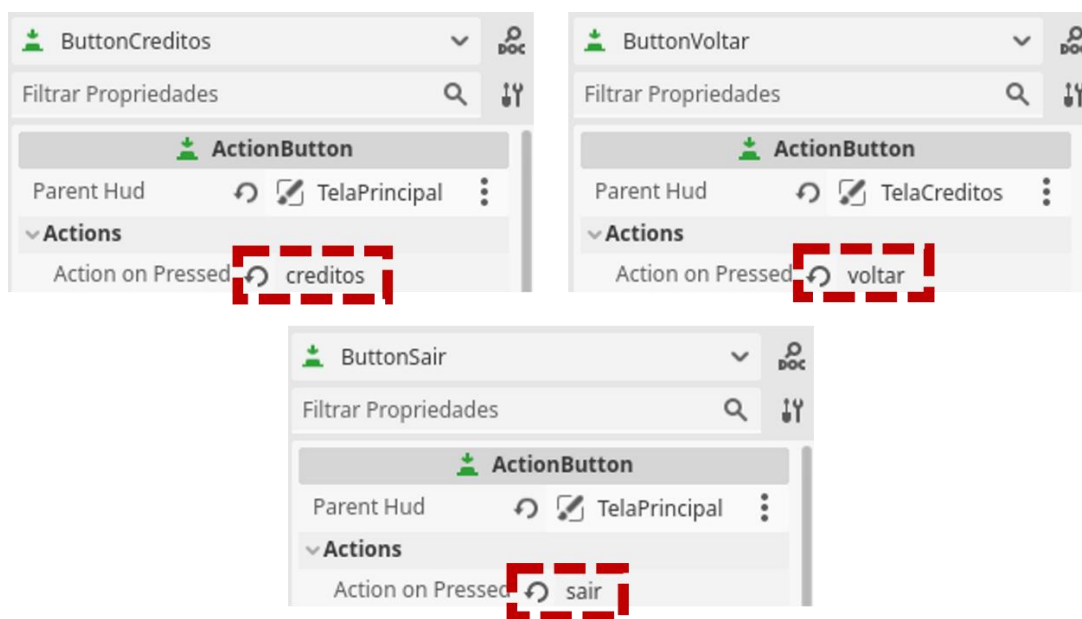
Figura 26 - Configuração de nó do tipo `ActionButton`



Fonte: A autoria própria, 2024.

Além da referência à raiz de cena, os botões do tipo `ActionButton` permitem ao programador definir quais ações serão emitidas a cada tipo específico de clique. Conforme mostrado na Figura 27, foram definidas as ações “creditos”, “voltar” e “sair” para quando os botões forem pressionados. Reiterando que para definir as ações de um `ActionButton`, não é necessária nenhuma alteração em código, exigindo somente a modificação dos valores no painel Inspector.

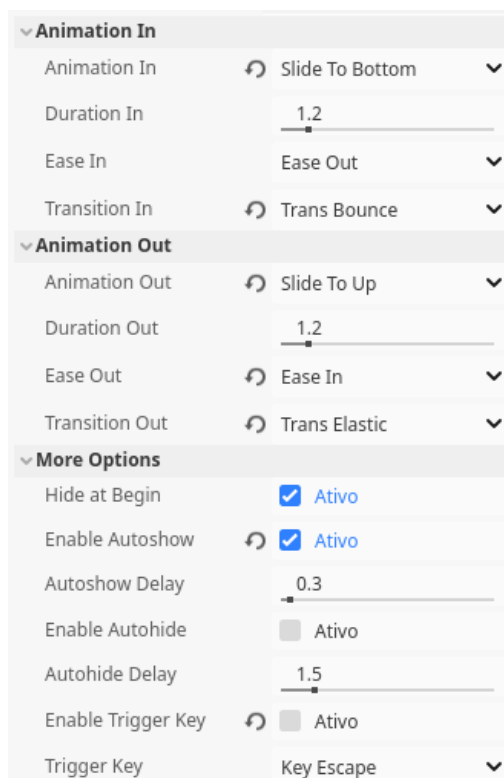
Figura 27 - Configuração de ação ativada por ActionButton



Fonte: Autoria própria, 2024.

Os nós raiz do tipo ActionHUD também permitem customização sem a necessidade de alterações em código. Na Figura 28 é possível perceber a variedade de personalizações disponíveis, totalmente acessíveis pelo painel do editor da Godot.

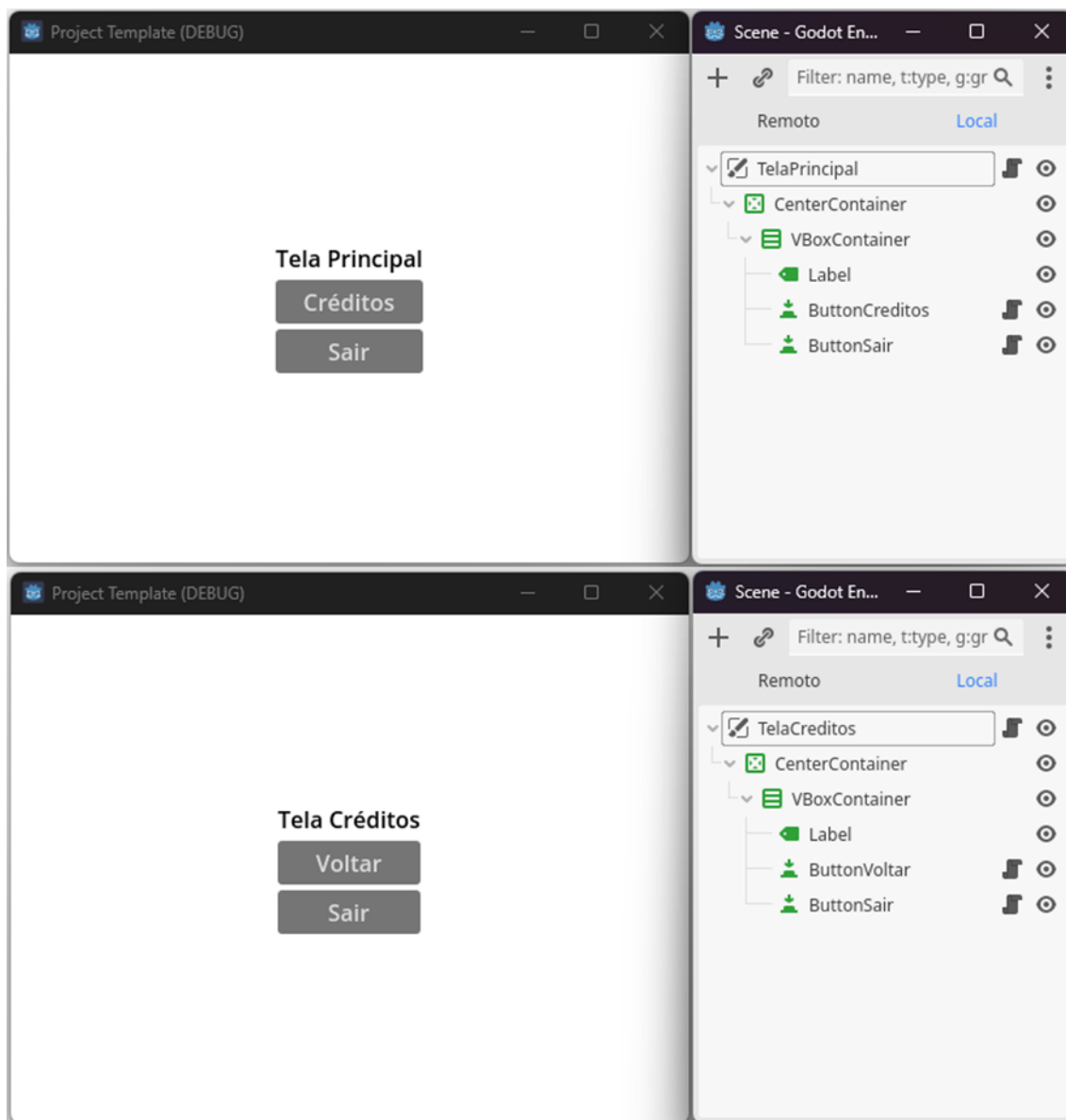
Figura 28 - Opções de personalização de um ActionHUD



Fonte: Autoria própria, 2024.

Com as duas cenas finalizadas, foram obtidas as estruturas da Figura 29.

Figura 29 - Cenas resultantes do projeto de exemplo



Fonte: Autoria própria, 2024.

Como última etapa para tornar a transição de telas funcional, o script do gerenciador de projeto deve ser modificado. Inicialmente deve-se sobrescrever o método *built-in* `_init`, que será utilizado para definir a cena inicial, conforme mostrado no Algoritmo 7.

Algoritmo 7 - Sobrescrita do método `_init` no gerenciador de projeto

```

4  ▾ func _init() -> void:
5  ▾ >|  initial_scene = ( # Definir a cena inicial do projeto
6  >| >|  preload("res://scenes/tela_principal.tscn")
7  >| )

```

Fonte: Autoria própria, 2024.

Ademais, o método `_on_action_triggered` deve ser sobrescrito para que as ações ativadas possam ser tratadas de acordo à necessidade do projeto. No Algoritmo 8, o método é sobrescrito para tratar dois possíveis contextos, sendo eles “tela\_principal” e “tela\_credits”, nas linhas 11 e 19 respectivamente. O primeiro contexto realiza a verificação de duas ações possíveis, “credits” e “sair” (linhas 13 e 17), enquanto o segundo verifica as ações “voltar” e “sair” (linhas 21 e 25). As ações “créditos” e “voltar” realizam trocas de tela utilizando o método `change_scene_to` do *singleton* `FDCore` (linhas 14 e 22).

Algoritmo 8 - Sobrescrita do método `_on_action_triggered` no gerenciador de projeto

```

10 ▾ func _on_action_triggered(action: String, context: String = "") -> void:
11 ▾ >|  if context == "tela_principal": # Verificar o contexto (origem) da ação
12 ▾ >| >|  match action: # Alternativa à estrutura utilizando if, elif e else
13 ▾ >| >| >|  "credits": # Verificar o nome da ação
14 ▾ >| >| >| >|  FDCore.change_scene_to( # Trocar para Tela Créditos
15 >| >| >| >| >|  preload("res://scenes/tela_credits.tscn").instantiate()
16 >| >| >| >| )
17 ▾ >| >| >|  "sair": # Verificar o nome da ação
18 >| >| >| >|  get_tree().quit() # Sair do programa em execução
19 ▾ >|  elif context == "tela_credits": # Verificar o contexto (origem) da ação
20 ▾ >| >|  match action: # Alternativa à estrutura utilizando if, elif e else
21 ▾ >| >| >|  "voltar": # Verificar o nome da ação
22 ▾ >| >| >| >|  FDCore.change_scene_to( # Trocar para Tela Principal
23 >| >| >| >| >|  preload("res://scenes/tela_principal.tscn").instantiate()
24 >| >| >| >| )
25 ▾ >| >| >|  "sair": # Verificar o nome da ação
26 >| >| >| >|  get_tree().quit() # Sair do programa em execução
27

```

Fonte: Autoria própria, 2024.

Sendo assim, o projeto resultante de exemplo é capaz de realizar a troca entre duas telas, exibindo logos animadas no início, animações para as interfaces criadas e transições a cada troca de tela. Para tais resultados, foram necessários somente dois arquivos de cena (Tela Principal e Tela Credits) e um script de 26 linhas.

## 5 RESULTADOS

Para validar os resultados obtidos, foram implementados os projetos A e B, ambos com o mesmo produto, porém somente o primeiro tendo utilizado o *framework* durante seu desenvolvimento. As análises dos dois projetos se realizaram seguindo os mesmos critérios, e posteriormente foram comparadas. Os critérios utilizados foram agrupados em três categorias: critérios quantitativos relativos à implementação, critérios de desempenho de execução e critérios de uso de recursos de hardware.

### 5.1 Critérios quantitativos relativos à implementação

Nesta categoria foram medidos aspectos relacionados diretamente ao desenvolvimento dos projetos, abrangendo somente os arquivos criados com extensão `.tscn` para cenas e `.gd` para *scripts*. Foi medido também a quantidade de linhas de cada *script*, desconsiderando as linhas vazias ou que contenham somente comentários. As implementações foram realizadas seguindo as práticas recomendadas pela documentação oficial da Godot Engine (GODOT, 2024e). Não foi possível realizar a medição da duração total do desenvolvimento dos projetos, dado que em ambos a implementação não foi realizada de forma contínua e contou com diversos intervalos, inviabilizando a medição.

### 5.2 Critérios de desempenho de execução

Para medir o desempenho de execução, foram considerados os aspectos relacionados ao tempo de execução em diferentes situações, todos seguindo o fluxo de execução descrito na Figura 30.

Figura 30 - Fluxo de execução para testes de desempenho de execução



Fonte: Autoria própria, 2024.

Os resultados foram obtidos por meio do próprio editor da Godot Engine, mediante as opções embutidas de depuração. Os critérios medidos nesta categoria incluem a média de taxa de quadros durante a execução (medido em FPS ou quadros por segundo), média de tempo de processamento ocioso (*idle processing*) e de físicas (*physics processing*), tempo de inicialização (tempo entre abertura do executável e o início da primeira *intro*), quantidade de objetos, nós e recursos criados durante a execução, e média da quantidade de nós órfãos (nós removidos sem deleção imediata). Para todos os critérios de média citados, foram registrados também os valores de pico (máximo e mínimo).

### **5.3 Critérios de uso de recursos de hardware**

Para esta categoria, foram medidos os aspectos relacionados diretamente ao uso de recursos de memória, CPU e GPU. Quanto à memória, foi medido a média e os picos de uso percentuais, além da quantidade em MiB. Quanto ao uso de CPU e GPU, foi medido somente o uso percentual, incluindo a média e os picos de uso. Foi medido também o tamanho do arquivo gerado após a exportação, considerando que o projeto foi exportado para a plataforma Windows 11, com recursos embutidos no arquivo executável.

### **5.4 Projetos de teste e comparativos**

O produto dos projetos de teste consistiu em um jogo de plataforma, contendo a tela principal, tela de vitória, tela de derrota e outras 3 telas representando as fases do jogo, totalizando seis telas. Outra funcionalidade presente nos projetos de teste foi um sistema de transições a cada troca de telas, e animação de interface para os botões dos menus. O código fonte dos projetos está disponível através do link [https://github.com/Firemanarg/testes\\_tcc](https://github.com/Firemanarg/testes_tcc).

Para a obtenção dos resultados, foi implementado um plugin de reporte de valores monitorados pelo depurador da Godot. Alguns valores de hardware foram obtidos por meio de reportes do software NVIDIA GeForce Experience. Posteriormente, os dados obtidos foram tratados utilizando um script em Python, que possibilitou a realização dos cálculos de média aritmética e dos valores de pico (máximo e mínimo). Tais valores calculados foram incluídos na Tabela 1.



Tabela 1 - Resultados dos testes com e sem o framework

		Critério utilizado	UDM	Valores medidos	
				Projeto A	Projeto B
1	CQRI	Quantidade de cenas criadas	qtd	9	13
2		Quantidade de <i>scripts</i> criados	qtd	2	9
3		Quantidade de linhas implementadas	qtd	132	308
4	CDE	Média de taxa de quadros durante a execução	FPS	129	133
5		Pico máximo de quadros durante a execução	FPS	145	145
6		Pico mínimo de quadros durante a execução	FPS	1	1
7		Média de tempo de processamento ocioso ( <i>idle</i> )	ms	94,471	69,455
8		Pico máximo de tempo de processamento ocioso ( <i>idle</i> )	ms	512,377	508,971
9		Pico mínimo de tempo de processamento ocioso ( <i>idle</i> )	ms	0,0	0,0
10		Média de tempo de processamento de físicas ( <i>physics</i> )	ms	0,511	0,316
11		Pico máximo de tempo de processamento de físicas ( <i>physics</i> )	ms	60,857	50,154
12		Pico mínimo de tempo de processamento de físicas ( <i>physics</i> )	ms	0,0	0,0
13		Tempo de inicialização	s	8,452	7,534
14		Média da quantidade de objetos criados durante a execução	qtd	2350	2341
15		Pico máximo da quantidade de objetos criados durante a execução	qtd	2450	2429
16	Pico mínimo da quantidade de objetos criados durante a execução	qtd	2298	2295	
17	Média da quantidade de nós criados durante a execução	qtd	28	26	
18	Pico máximo da quantidade de nós criados durante a execução	qtd	59	56	
19	Pico mínimo da quantidade de nós criados durante a execução	qtd	17	15	
20	Média da quantidade de recursos criados durante a execução	qtd	792	791	
21	Pico máximo da quantidade de recursos criados durante a execução	qtd	792	791	
22	Pico mínimo da quantidade de recursos criados durante a execução	qtd	792	791	
23	Média de quantidade de nós órfãos	qtd	2	1	
24	Pico máximo da quantidade de nós órfãos	qtd	24	1	
25	Pico mínimo da quantidade de nós órfãos	qtd	0	0	
26	CURH	Média percentual de uso de memória durante a execução	%	0,383	0,381
27		Pico máximo do percentual de uso de memória	%	0,387	0,383
28		Pico mínimo do percentual de uso de memória	%	0,377	0,374
29		Média quantitativa do uso de memória durante a execução	MiB	97,943	97,258
30		Pico máximo quantitativo do uso de memória durante a execução	MiB	98,828	97,986
31		Pico mínimo quantitativo do uso de memória durante a execução	MiB	96,298	95,534
32		Média percentual de uso de CPU durante a execução	%	11,741	11,253
33		Pico máximo do percentual de uso de CPU	%	22,0	16,0
34		Pico mínimo do percentual de uso de CPU	%	8,0	7,0
35		Média percentual de uso de GPU durante a execução	%	24,422	29,086
36		Pico máximo do percentual de uso de GPU	%	78,0	92,0
37		Pico mínimo do percentual de uso de GPU	%	0,0	0,0
38	Tamanho do arquivo executável (Windows 11 e recursos embutidos)	MB	134,270	134,176	
Legenda:	CQRI → Critérios quantitativos relativos à implementação				
	CDE → Critérios de desempenho de execução				
	CURH → Critérios de uso de recursos de hardware				
	UDM → Unidade de medida				
	qtd → Quantidade				

Fonte: Autoria própria, 2024.

Para os comparativos, foi calculada a diferença absoluta e relativa dos valores obtidos. A diferença absoluta (DA) foi calculada pela subtração entre os valores do projeto A ( $V_A$ ) e do projeto B ( $V_B$ ), enquanto diferença percentual ( $DP(\%)$ ) foi calculada dividindo DA por  $V_B$ , conforme a eq. (1). Os resultados podem ser visualizados na Tabela 2.

$$DP(\%) = \frac{(V_A - V_B)}{V_B} \times 100 \quad (1)$$

Tabela 2 - Comparativos percentuais dos resultados obtidos após com os testes

	Critério utilizado	UDM	DA	DP (%)	RE
1	Quantidade de cenas criadas	qtd	-4	-30,769	< 0
2	Quantidade de <i>scripts</i> criados	qtd	-7	-77,778	< 0
3	Quantidade de linhas implementadas	qtd	-176	-57,143	< 0
4	Média de taxa de quadros durante a execução	FPS	-4	-3,008	> 0
5	Pico máximo de quadros durante a execução	FPS	0	0	> 0
6	Pico mínimo de quadros durante a execução	FPS	0	0	< 0
7	Média de tempo de processamento ocioso ( <i>idle</i> )	ms	25,016	36,018	< 0
8	Pico máximo de tempo de processamento ocioso ( <i>idle</i> )	ms	3,406	0,669	< 0
9	Pico mínimo de tempo de processamento ocioso ( <i>idle</i> )	ms	0	0	< 0
10	Média de tempo de processamento de físicas ( <i>physics</i> )	ms	0,195	61,709	< 0
11	Pico máximo de tempo de processamento de físicas ( <i>physics</i> )	ms	10,703	21,340	< 0
12	Pico mínimo de tempo de processamento de físicas ( <i>physics</i> )	ms	0	0	< 0
13	Tempo de inicialização	s	0,918	12,185	< 0
14	Média da quantidade de objetos criados durante a execução	qtd	9	0,384	< 0
15	Pico máximo da quantidade de objetos criados durante a execução	qtd	21	0,865	< 0
16	Pico mínimo da quantidade de objetos criados durante a execução	qtd	3	0,131	< 0
17	Média da quantidade de nós criados durante a execução	qtd	2	7,692	< 0
18	Pico máximo da quantidade de nós criados durante a execução	qtd	3	5,357	< 0
19	Pico mínimo da quantidade de nós criados durante a execução	qtd	2	13,333	< 0
20	Média da quantidade de recursos criados durante a execução	qtd	1	0,126	< 0
21	Pico máximo da quantidade de recursos criados durante a execução	qtd	1	0,126	< 0
22	Pico mínimo da quantidade de recursos criados durante a execução	qtd	1	0,126	< 0
23	Média de quantidade de nós órfãos	qtd	1	100	< 0
24	Pico máximo da quantidade de nós órfãos	qtd	23	2300	< 0
25	Pico mínimo da quantidade de nós órfãos	qtd	0	0	< 0
26	Média percentual de uso de memória durante a execução	%	0,002	0,525	< 0
27	Pico máximo do percentual de uso de memória	%	0,004	1,044	< 0
28	Pico mínimo do percentual de uso de memória	%	0,003	0,802	< 0
29	Média quantitativa do uso de memória durante a execução	MiB	0,685	0,704	< 0
30	Pico máximo quantitativo do uso de memória durante a execução	MiB	0,842	0,859	< 0
31	Pico mínimo quantitativo do uso de memória durante a execução	MiB	0,764	0,8	< 0
32	Média percentual de uso de CPU durante a execução	%	0,488	4,337	< 0
33	Pico máximo do percentual de uso de CPU	%	6	37,5	< 0
34	Pico mínimo do percentual de uso de CPU	%	1	14,286	< 0
35	Média percentual de uso de GPU durante a execução	%	-4,664	-16,035	< 0
36	Pico máximo do percentual de uso de GPU	%	-14	-15,217	< 0
37	Pico mínimo do percentual de uso de GPU	%	0	0	< 0
38	Tamanho do arquivo executável (Windows 11 e recursos embutidos)	MB	0,094	0,07	< 0
Legenda:	CQRI → Critérios quantitativos relativos à implementação				
	CDE → Critérios de desempenho de execução				
	CURH → Critérios de uso de recursos de hardware				
	DP → Diferença percentual				
	RE → Resultado esperado				

Fonte: Autoria própria, 2024.

Com base nas diferenças percentuais da Tabela 2, foram calculados os percentuais de aproveitamento (PA) descritos na Tabela 3, desconsiderando os valores de pico (máximos e mínimos). Para tais cálculos, foram considerados os valores dos

resultados esperados para definir os sinais (positivos ou negativos) dos valores calculados. Um PA positivo indica vantagem utilizando o *framework*. Espera-se que os PAs sejam positivos ou próximos de 0%. Para facilitar a visualização, os critérios foram agrupados em categorias, segundo seus contextos no desenvolvimento. Foram calculadas também as médias aritméticas dos PAs de cada categoria.

Tabela 3 - Percentuais de aproveitamento por critério

Percentuais de Aproveitamento por Critério	
Critério de Implementação	Percentual de Aproveitamento
Quantidade de cenas criadas	30,77%
Quantidade de scripts criados	77,78%
Quantidade de linhas implementadas	57,14%
Média de Aproveitamento	55,23%
Critério de Desempenho de Execução	Percentual de Aproveitamento
Média de taxa de quadros durante a execução	-3,01%
Média de tempo de processamento ocioso (idle)	-36,02%
Média de tempo de processamento de físicas (physics)	-61,71%
Tempo de inicialização	-12,18%
Média da quantidade de objetos criados durante a execução	-0,38%
Média da quantidade de nós criados durante a execução	-7,69%
Média da quantidade de recursos criados durante a execução	-0,13%
Média de quantidade de nós órfãos	-100,00%
Média de Aproveitamento	-27,64%
Critério de Uso de Hardware	Percentual de Aproveitamento
Média percentual de uso de memória durante a execução	-0,52%
Média quantitativa do uso de memória durante a execução	-0,70%
Média percentual de uso de CPU durante a execução	-4,34%
Média percentual de uso de GPU durante a execução	16,04%
Tamanho do arquivo executável (Windows 11 e recursos embutidos)	-0,07%
Média de Aproveitamento	2,08%

Fonte: Autoria própria, 2024.

A Tabela 3 apresenta os critérios agrupados em três categorias, e as respectivas médias aritméticas de cada uma delas.

## 5.5 Análise dos resultados obtidos

Quanto aos resultados presentes nas tabelas 1, 2 e 3, agrupados em categorias, e considerando que o projeto A utiliza o *framework* e o projeto B não utiliza o *framework*, infere-se que:

### 5.5.1 Quanto aos critérios de implementação

Houve um aproveitamento significativo, indicando uma média de 55,23% de vantagem ao utilizar o *framework*. A quantidade de linhas de código necessárias para

alcançar o mesmo produto foi de menos da metade se comparado ao projeto B, e a quantidade de scripts necessários alcançou 77,78% a menos no projeto A;

### **5.5.2 *Quanto aos critérios de desempenho de execução***

Houve um aproveitamento médio negativo de 27,64%, indicando desvantagem quanto ao uso do *framework*. Tais resultados inferiores possivelmente se devem à possibilidade de customização, existentes somente no projeto A. Contudo, ao analisar as diferenças absolutas dos valores médios com unidade de medida de tempo (ms e s), percebeu-se que os valores alcançaram resultados próximos entre os projetos, com diferenças absolutas de 25,016ms, 0,195ms e 0,918s para os critérios de média de processamento ocioso, média de processamento de físicas e tempo de inicialização. A média da quantidade de nós órfãos também indicou resultados próximos, com apenas 1 de diferença (apesar do percentual negativo de 100%).

Um indicativo de que houve um bom aproveitamento na prática foi a baixa diferença na média de taxa de quadros por segundo, com somente 4 FPS a menos no projeto A, indicando uma diferença percentual de -3,01% quando comparado ao projeto B;

### **5.5.3 *Quanto aos critérios de uso de hardware***

Houve um aproveitamento médio de 2,08% ao utilizar o *framework*. Apesar do aproveitamento positivo, a maior parte dos critérios apresentaram valores negativos, porém bastante próximos de 0% (média de 1,4% considerando somente valores negativos). O critério que mais se diferiu foi o de uso de GPU, indicando um uso de 16,04% a menos quando utilizado o *framework*. Quanto ao uso de CPU e de memória, o projeto A alcançou valores próximos dos obtidos no projeto B. Quanto ao tamanho dos arquivos executáveis, houve uma diferença de apenas 94 KB.

## 6 CONSIDERAÇÕES FINAIS

Em síntese, esta pesquisa consistiu na documentação da arquitetura, estrutura, desenvolvimento, funcionamento e testes de um *framework* implementado na Godot Engine. Os propósitos do produto da presente pesquisa foram de padronizar a estrutura de projetos e proporcionar componentes capazes de reduzir a quantidade de código necessário durante a etapa de implementação, possivelmente acelerando os processos de desenvolvimento e manutenção de novos projetos.

A análise de requisitos foi realizada seguindo a estratégia *bottom-up*, recomendada quando os requisitos não estão muito bem definidos no início do projeto. Durante essa etapa, foram analisados projetos anteriores que apresentaram problemas durante o desenvolvimento, porém que posteriormente tiveram implementações de soluções satisfatórias.

Para atender à necessidade de padronização, foi idealizada uma arquitetura baseada em camadas especializadas, classificadas por seus respectivos propósitos e adaptadas especificamente para a Godot Engine. Estabeleceu-se que a comunicação entre cenas e camadas seria dada por ativadores de ação, os quais possuem um indicativo de mensagem (nome da ação) e sua origem (contexto). Todo o fluxo de projetos seguindo essa arquitetura deve ser conduzido por um gerenciador de projeto, que lida com os ativadores de forma centralizada.

Com a finalidade de contribuir com a comunidade de código aberto, todo o código desenvolvido nesta pesquisa foi disponibilizado em um repositório público na plataforma GitHub, disponível através do link <https://github.com/FireDroidGameStudios/godot-project-template>. Espera-se que futuramente o projeto venha a ser utilizado e aprimorado pela comunidade.

Ao final do desenvolvimento do *framework*, foram desenvolvidos dois projetos de teste, tendo o primeiro utilizado o *framework* e o segundo não. Ambos os projetos obtiveram o mesmo produto: um jogo de plataforma com sistema de troca de telas, transições animadas e gerenciamento de níveis (fases). Os projetos de teste possibilitaram a realização de medições de implementação, desempenho de execução e uso de *hardware*. Os valores obtidos durante as medições foram comparados posteriormente para uma análise detalhada de aproveitamento quanto ao uso do *framework*.

Os testes utilizando o *framework* indicaram aproveitamento de 55,23% positivo quanto às implementações, 27,64% negativo quanto ao desempenho de execução, e

2,08% positivo quanto ao uso de hardware. Tais valores indicam uma vantagem significativa quanto aos esforços de implementação (quantidade de código e de arquivos de código necessários) utilizando o *framework*, e demonstram um baixo impacto no desempenho de execução, dado que a diferença na média da taxa de quadros foi de apenas 4 FPS a menos se comparado ao projeto que não utiliza o *framework*. Quanto ao uso de hardware, o *framework* utilizou 16,04% a menos de GPU, enquanto os valores de CPU e memória permaneceram bastante próximos.

Em conclusão, os resultados da pesquisa mostraram-se satisfatórios quanto ao que se esperava inicialmente, além de possibilitar novos estudos na área. Algumas sugestões para pesquisas futuras são:

- Implementação de sistema de salvamento facilitado de recursos, com integração ao *framework*;
- Implementação de sistema de carregamento de recursos (*loading*) integrado ao *framework*, com possibilidade de personalização de telas de carregamento e acesso simplificado aos recursos carregados;
- Implementação de recursos para desenvolvimento de projetos incluindo multijogador online, com compatibilidade total ou parcial com o *framework*;
- Testes quantitativos de ganho de tempo de desenvolvimento ao utilizar o *framework*;
- Elaboração de um manual (*wiki*) no repositório do GitHub para facilitar o entendimento quanto ao uso do *framework* a novos usuários.

## REFERÊNCIAS BIBLIOGRÁFICAS

CODENIE, Wim; HONDT, Koen D.; STEYAERT, Patrick; VERCAMMEN, Arlette. **From Custom Applications to Domain-Specific Frameworks**. Communications of the ACM. [S. l], v. 40, n. 10, p. 71-77, out. 1997. DOI: <https://doi.org/10.1145/262793.262807>.

DEFOLD. **Defold – Official Homepage – Cross platform game engine**. Defold, 2024. Disponível em: <https://defold.com>. Acesso em: 04 de abril de 2024.

EDWIN, Njeru M. **Software Frameworks, Architectural and Design Patterns**. Journals of Software Engineering and Applications. Nairobi, v. 7, n. 8, p. 670-678, jul. 2014. DOI: <http://dx.doi.org/10.4236/jsea.2014.78061>.

FAYAD, Mohamed E.; SCHMIDT, Douglas C. **Object-Oriented Application Frameworks**. Communications of the ACM. [S. l], v. 40, n. 10, p. 32-38, out. 1997. DOI: <https://doi.org/10.1145/262793.262798>.

FORBES. **16 Obstacles To A Successful Software Project (And How To Avoid Them)**. Forbes Technology Council, jun. 2022. Disponível em: <https://www.forbes.com/sites/forbestechcouncil/2022/06/21/16-obstacles-to-a-successful-software-project-and-how-to-avoid-them>. Acesso em: 15 de mar. de 2024.

GIL, Antônio Carlos. **Como Elaborar Projetos de Pesquisa**. 6a ed. São Paulo: Editora Atlas Ltda., 2017.

GITHUB. **About commits - GitHub Docs**. GitHub Docs, 2024a. Disponível em: <https://docs.github.com/en/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/about-commits>. Acesso em: 03 de maio de 2024.

GITHUB. **About merge conflicts - GitHub Docs**. GitHub Docs, 2024b. Disponível em: <https://docs.github.com/en/pull-requests/collaborating-with-pull->

requests/addressing-merge-conflicts/about-merge-conflicts. Acesso em: 03 de maio de 2024.

GITHUB. **About pull request reviews - GitHub Docs**. GitHub Docs, 2024c. Disponível em: <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/reviewing-changes-in-pull-requests/about-pull-request-reviews>. Acesso em: 03 de maio de 2024.

GITHUB. **Cloning a repository - GitHub Docs**. GitHub Docs, 2024d. Disponível em: <https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-repository>. Acesso em: 02 de maio de 2024.

GITHUB. **Creating a pull request - GitHub Docs**. GitHub Docs, 2024e. Disponível em: <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request>. Acesso em: 03 de maio de 2024.

GITHUB. **Creating and deleting branches within your repository - GitHub Docs**. GitHub Docs, 2024f. Disponível em: <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-and-deleting-branches-within-your-repository>. Acesso em: 02 de maio de 2024.

GITHUB. **Deleting and restoring branches in a pull request - GitHub Docs**. GitHub Docs, 2024g. Disponível em: <https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-branches-in-your-repository/deleting-and-restoring-branches-in-a-pull-request>. Acesso em: 03 de maio de 2024.

GITHUB. **GitHub - godotengine/godot: Godot Engine – Multi-platform 2D and 3D game engine**. Github, 2024h. Disponível em: <https://github.com/godotengine/godot>. Acesso 23 de março de 2024.



GITHUB. **GitHub flow - GitHub Docs**. GitHub Docs, 2024i. Disponível em: <https://docs.github.com/en/get-started/using-github/github-flow>. Acesso em: 02 de maio de 2024.

GITHUB. **Merging a pull request - GitHub Docs**. GitHub Docs, 2024j. Disponível em: <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/incorporating-changes-from-a-pull-request/merging-a-pull-request>. Acesso em: 03 de maio de 2024.

GITHUB. **Quickstart for repositories - GitHub Docs**. GitHub Docs, 2024k. Disponível em: <https://docs.github.com/en/repositories/creating-and-managing-repositories/quickstart-for-repositories>. Acesso em: 02 de maio de 2024.

GITHUB. **Quickstart for repositories - GitHub Docs**. GitHub Docs, 2024k. Disponível em: <https://docs.github.com/en/repositories/creating-and-managing-repositories/quickstart-for-repositories>. Acesso em: 02 de maio de 2024.

GITHUB. **Resolving a merge conflict on GitHub - GitHub Docs**. GitHub Docs, 2024l. Disponível em: <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-on-github>. Acesso em: 03 de maio de 2024.

GITHUB. **Sobre o GitHub e o Git - GitHub Docs**. GitHub Docs, 2024m. Disponível em: <https://docs.github.com/pt/get-started/start-your-journey/about-github-and-git>. Acesso em: 30 de abril de 2024.

GODOT. **@GDScript — Godot Engine (stable) documentation in English**. Godot Docs, 2024a. Disponível em: [https://docs.godotengine.org/en/stable/classes/class\\_%40gdscript.html](https://docs.godotengine.org/en/stable/classes/class_%40gdscript.html). Acesso em: 29 de abril de 2024.

GODOT. **About the Asset Library — Godot Engine (stable) documentation in English**. Godot Docs, 2024b. Disponível em:

[https://docs.godotengine.org/en/stable/community/asset\\_library/what\\_is\\_assetlib.html](https://docs.godotengine.org/en/stable/community/asset_library/what_is_assetlib.html).  
Acesso em: 29 de abril de 2024.

GODOT. **Canvas layers — Godot Engine (stable) documentation in English**. Godot Docs, 2024c. Disponível em:  
[https://docs.godotengine.org/en/stable/tutorials/2d/canvas\\_layers.html](https://docs.godotengine.org/en/stable/tutorials/2d/canvas_layers.html). Acesso em: 22 de abril de 2024.

GODOT. **GScript reference — Godot Engine (stable) documentation in English**. Godot Docs, 2024d. Disponível em:  
[https://docs.godotengine.org/en/stable/tutorials/scripting/gscript/gscript\\_basics.html](https://docs.godotengine.org/en/stable/tutorials/scripting/gscript/gscript_basics.html).  
Acesso em: 29 de abril de 2024.

GODOT. **GScript style guide — Godot Engine (stable) documentation in English**. Godot Docs, 2024f. Disponível em:  
[https://docs.godotengine.org/en/stable/tutorials/scripting/gscript/gscript\\_styleguide.html](https://docs.godotengine.org/en/stable/tutorials/scripting/gscript/gscript_styleguide.html). Acesso em: 23 de maio de 2024.

GODOT. **Godot Engine – Free and open source 2D and 3D game engine**. Godot, 2024g. Disponível em: <https://godotengine.org>. Acesso em: 04 de abril de 2024.

GODOT. **Godot notifications — Godot Engine (stable) documentation in English**. Godot Docs, 2024h. Disponível em:  
[https://docs.godotengine.org/en/stable/tutorials/best\\_practices/godot\\_notifications.html](https://docs.godotengine.org/en/stable/tutorials/best_practices/godot_notifications.html).  
Acesso em: 29 de abril de 2024.

GODOT. **Idle and Physics Processing — Godot Engine (stable) documentation in English**. Godot Docs, 2024i. Disponível em:  
[https://docs.godotengine.org/en/stable/tutorials/scripting/idle\\_and\\_physics\\_processing.html](https://docs.godotengine.org/en/stable/tutorials/scripting/idle_and_physics_processing.html). Acesso em: 29 de abril de 2024.

GODOT. **Introduction - Godot Engine (stable) documentation in English**. Godot Docs, 2024j. Disponível em:

<https://docs.godotengine.org/en/stable/about/introduction.html>. Acesso em: 04 de abril de 2024.

**GODOT. Introduction to 3D — Godot Engine (stable) documentation in English.**

Godot Docs, 2024k. Disponível em:

[https://docs.godotengine.org/en/stable/tutorials/3d/introduction\\_to\\_3d.html](https://docs.godotengine.org/en/stable/tutorials/3d/introduction_to_3d.html). Acesso em: 23 de abril de 2024.

**GODOT. Introduction to Godot - Godot Engine (stable) documentation in English.**

Godot Docs, 2024l. Disponível em:

[https://docs.godotengine.org/en/stable/getting\\_started/introduction/introduction\\_to\\_godot.html](https://docs.godotengine.org/en/stable/getting_started/introduction/introduction_to_godot.html). Acesso em: 04 de abril de 2024.

**GODOT. Node — Godot Engine (stable) documentation in English.**

Godot Docs, 2024m. Disponível em: [https://docs.godotengine.org/en/stable/classes/class\\_node.html](https://docs.godotengine.org/en/stable/classes/class_node.html).

Acesso em: 29 de abril de 2024.

**GODOT. Overview of Godot's key concepts — Godot Engine (stable)**

**documentation in English.** Godot Docs, 2024n. Disponível em:

[https://docs.godotengine.org/en/stable/getting\\_started/introduction/key\\_concepts\\_overview.html](https://docs.godotengine.org/en/stable/getting_started/introduction/key_concepts_overview.html). Acesso em: 23 de abril de 2024.

**GODOT. Scripting languages — Godot Engine (stable) documentation in English.**

Godot Docs, 2024o. Disponível em:

[https://docs.godotengine.org/en/stable/getting\\_started/step\\_by\\_step/scripting\\_languages.html](https://docs.godotengine.org/en/stable/getting_started/step_by_step/scripting_languages.html). Acesso em: 29 de abril de 2024.

**GODOT. User interface (UI) — Godot Engine (stable) documentation in English.**

Godot Docs, 2024p. Disponível em:

<https://docs.godotengine.org/en/stable/tutorials/ui/index.html>. Acesso em: 23 de abril de 2024.

GODOT. **Using SceneTree — Godot Engine (stable) documentation in English.**

Godot Docs, 2024q. Disponível em:

[https://docs.godotengine.org/en/stable/tutorials/scripting/scene\\_tree.html](https://docs.godotengine.org/en/stable/tutorials/scripting/scene_tree.html). Acesso em: 29 de abril de 2024.

GODOT. **Using signals — Godot Engine (stable) documentation in English.** Godot

Docs, 2024r. Disponível em:

[https://docs.godotengine.org/en/stable/getting\\_started/step\\_by\\_step/signals.html](https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html).

Acesso em: 25 de abril de 2024.

GODOT. **What is GDExtension? — Godot Engine (stable) documentation in English.** Godot Docs, 2024s. Disponível em:

[https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/what\\_is\\_gdextension.html](https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/what_is_gdextension.html). Acesso em: 29 de abril de 2024.

GREGORY, Jason. **Game Engine Architecture**. 3a ed. Nova Iorque: CRC Press, 2018.

JOHN, Clay. **Announcing a collaboration with Google and The Forge**. Godot, dec.

2023. Disponível em: <https://godotengine.org/article/collaboration-with-google-forge-2023>. Acesso em: 22 de abril de 2024.

JOHNSON, Ralph E. **Frameworks = (Components + Patterns)**. Communications of the ACM. [S. l], v. 40, n. 10, p. 39-42, out. 1997. DOI:

<https://doi.org/10.1145/262793.262799>.

LAKATOS, Eva M.; MARCONI, Marina A. **Fundamentos de Metodologia Científica**. 8a ed. São Paulo: Editora Atlas Ltda., 2017.

LINIETSKY, Juan, MANZUR, Ariel et al. **Introduction – Godot Engine (stable) documentation in English**. Godot Docs, 2024. Disponível em:

<https://docs.godotengine.org/en/stable/about/introduction.html>. Acesso 23 de março de 2024.

MARTIN, Robert C. **Agile Software Development: Principles, Patterns, and Practices**. Harlow: Pearson, 2014.

MURILO, Cássio; BITTENCOURT, Jeniffer. **Framework: o que é e pra que serve essa ferramenta?**. Alura, nov. 2021. Disponível em:  
<https://www.alura.com.br/artigos/framework-o-que-e-pra-que-serve-essa-ferramenta>. Acesso em 15 de março de 2024.

MYLLÄRNIEMI, Varvana; KUJALA, Sari; RAATIKAINEN, Mikko; SEVÓN, Piia. **Development as a journey: factors supporting the adoption and use of software frameworks**. Journal of Software Engineering Research and Development. [S. l], v. 6, n. 6, jun. 2018. DOI: <https://doi.org/10.1186/s40411-018-0050-8>. Disponível em:  
<https://jserd.springeropen.com/articles/10.1186/s40411-018-0050-8>. Acesso em: 20 de março de 2024.

POLITOWSKY, Cristiano; PETRILLO, Fabio; MONTANDON, João E.; VALENTE, Marco T.; GUÉHÉNEUC, Yann-Gaël. **Are Game Engines Software Frameworks? A Three-perspective Study**. Journal of Systems and Software. [S. l], v. 171, jan. 2020. DOI: <https://doi.org/10.1016/j.jss.2020.110846>.

SCHMID, Hans A. **Systematic Framework Design by Generalization**. Communications of the ACM. [S. l], v. 40, n. 10, p. 48-51, out. 1997. DOI: <https://doi.org/10.1145/262793.262803>.

SPINELLIS, Diomidis. **Git**. IEEE Software. [S. l], v. 29, p. 100-101, mai. 2012. DOI: <https://doi.org/10.1109/MS.2012.61>.

THORN, Alan. **Game Engine Design and Implementation**. Ontário: Jones & Bartlett Learning, 2010.

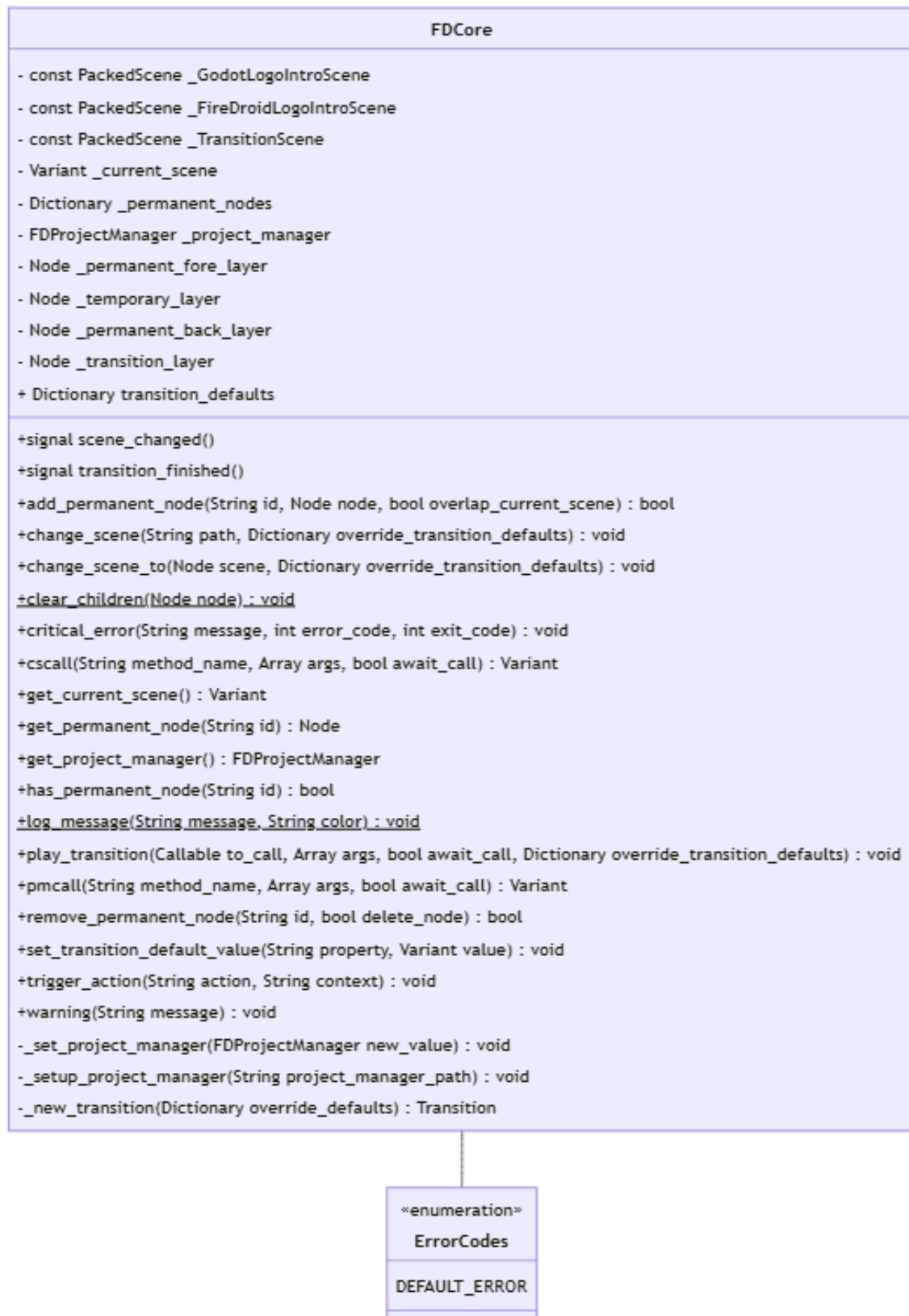
UNITY. **Plataforma e editor de desenvolvimento em 3d em tempo real | Unity**. Unity, 2024. Disponível em: <https://unity.com/pt/products/unity-engine>. Acesso em: 04 de abril de 2024.

UNREAL. **A ferramenta de criação 3D em tempo real mais eficiente – Unreal Engine**. Unreal Engine, 2024. Disponível em: <https://www.unrealengine.com/pt-BR>. Acesso em: 04 de abril de 2024.

VERSCHELDE, Rémi. **Godot Engine receiving a new grant from Meta's Reality Labs**. Godot, dec. 2021. Disponível em: <https://godotengine.org/article/godot-engine-receiving-new-grant-meta-reality-labs>. Acesso em: 22 de abril de 2024.

WAZLAWICK, Raul S. **Metodologia da Pesquisa para Ciência da Computação**. 2a ed. [S.l.]: Campus, 2014.

## APÊNDICE A - Diagrama de classe - FDCore



### APÊNDICE B - Diagrama de classe – FDProjectManager

<b>FDProjectManager</b>
- PackedScene initial_scene
+on_action_triggered(String action, String context) : void
<u>- on action triggered(String action, String context) : void</u>

### APÊNDICE C - Diagrama de classe – ActionButton

<b>ActionButton</b>
+ const float DefaultAnimationDuration
+ const float DefaultAutoshowDelay
+ const float DefaultAutohideDelay
- _AnimationState _animation_state
- Tween _tween
+ @export ActionHUD parent_hud
+ @export String action_on_pressed
+ @export String action_on_release
+ @export String action_on_button_down
+ @export String action_on_button_up
+ @export String action_on_toggled_true
+ @export String action_on_toggled_false
-_connect_signals() : void
-_trigger_action_on_toggled(bool toggled_on) : void

### APÊNDICE D - Diagrama de classe – SignalActionTrigger

<b>SignalActionTrigger</b>
+ @export String action_context
+ String signal_to_connect
+ String action_to_trigger
-_get_parent_signals() : PackedStringArray
-_get_signal_arg_count(String signal_name) : int
-_trigger_action() : void



## APÊNDICE E - Diagrama de classe – Transition



## APÊNDICE F - Diagrama de classe – LogoIntro

LogoIntro
<ul style="list-style-type: none"> <li>- const Dictionary _Shaders</li> <li>+ @export bool autoplay</li> <li>+ @export float horizontal_margins</li> <li>+ @export float vertical_margins</li> <li>+ @export Color background_color</li> <li>+ @export SpriteFrames frames</li> <li>+ @export bool auto_update_animation_frames</li> <li>+ @export float frame_rate</li> <li>+ @export float initial_delay</li> <li>+ @export float final_delay</li> <li>+ @export bool can_be_skipped</li> <li>+ @export float skip_lock_delay</li> <li>+ @export Key skip_key</li> <li>+ @export AudioStream audio_stream</li> <li>+ @export float audio_stream_delay</li> <li>+ @onready Variant animation_player</li> <li>+ @onready Variant timer_start_animation</li> <li>+ @onready Variant timer_end_animation</li> <li>+ @onready Variant timer_play_sound</li> <li>+ @onready Variant timer_skip_delay</li> <li>- bool _can_skip</li> <li>- bool _has_been_skipped</li> </ul>
<ul style="list-style-type: none"> <li>+signal started()</li> <li>+signal finished()</li> <li>+play() : void</li> <li>+skip() : void</li> <li>-_set_background_color(Color new_color) : void</li> <li>-_set_frames(SpriteFrames new_frames) : void</li> <li>-_set_h_margins(float new_margins) : void</li> <li>-_set_v_margins(float new_margins) : void</li> <li>-_on_frames_changed() : void</li> <li>-_update_animation() : void</li> <li>-_on_finished(String _anim_name) : void</li> <li>-_on_timer_start_animation_timeout() : void</li> <li>-_on_timer_end_animation_timeout() : void</li> <li>-_on_timer_play_sound_timeout() : void</li> <li>-_on_timer_skip_delay_timeout() : void</li> </ul>

## APÊNDICE G - Diagrama de classe – ActionHUD

