

UNIVERSIDADE DE SÃO PAULO  
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA  
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

**Jogo Estilo Puzzle Facilitador  
de Aprendizado dos Conceitos  
Básicos de Programação**

Mateus Agostinho dos Anjos

MONOGRAFIA FINAL

MAC 0499 — TRABALHO DE  
FORMATURA SUPERVISIONADO

Supervisor: Prof. Dr. Marco Dimas Gubitoso

São Paulo  
28 de outubro de 2019



# Agradecimentos

Eu poderia agradecer algumas pessoas aqui.



# Resumo

Mateus Agostinho dos Anjos. **Jogo Estilo Puzzle Facilitador de Aprendizado dos Conceitos Básicos de Programação**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2019.

O resumo está sendo produzido.

**Palavras-chave:** Palavra-chave1. Palavra-chave2. Palavra-chave3.



## Lista de Abreviaturas

IME Instituto de Matemática e Estatística  
USP Universidade de São Paulo

## Lista de Símbolos

# Lista de Figuras

1.1	Jogo não conectado . . . . .	2
1.2	Jogo conectado . . . . .	3
1.3	Exemplo de comandos disponíveis . . . . .	3
1.4	Jogador preenchendo os argumentos . . . . .	4
1.5	Sistema pronto para execução . . . . .	4
1.6	Início da execução do sistema . . . . .	5
1.7	Valor do <i>Input</i> antes da operação . . . . .	5
1.8	Valor do <i>Input</i> após a operação . . . . .	6
2.1	Exemplo de árvore . . . . .	7
2.2	Sinais pré programados de um nó do tipo <i>Button</i> . . . . .	12
2.3	Sinal <i>input_output_defined</i> criado por código . . . . .	12
2.4	Sinal <i>input_output_defined</i> conectado via interface . . . . .	12
2.5	Sinal <i>variable_changed</i> conectado via código (linha 12.) . . . . .	12



## **Lista de Tabelas**

## **Lista de Programas**



# Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Motivação e Objetivos . . . . .	1
1.2	Organização do Projeto . . . . .	2
1.3	Noção básica sobre o jogo . . . . .	2
<b>2</b>	<b>Conceitos Básicos</b>	<b>7</b>
2.1	O Conceito de Árvore . . . . .	7
2.2	O que é <i>Game Engine</i> ? . . . . .	8
2.3	Entendendo sobre a <i>Godot Engine</i> . . . . .	8
2.3.1	Nós . . . . .	8
2.3.2	Cenas . . . . .	9
2.3.3	Instâncias . . . . .	9
2.3.4	<i>SceneTree</i> . . . . .	9
2.3.5	Singleton . . . . .	11
2.3.6	Sinais . . . . .	11
2.3.7	<i>GDScripts</i> . . . . .	13
<b>3</b>	<b>Bibliografia</b>	<b>15</b>

## Apêndices

## Anexos

<b>Índice Remissivo</b>	<b>17</b>
-------------------------	-----------



# Capítulo 1

## Introdução

### 1.1 Motivação e Objetivos

Com a crescente ascensão da tecnologia nos dias de hoje o conhecimento sobre programação tem se tornado cada vez mais importante, não só pelas inúmeras aplicações que existem, mas também por ser um facilitador, tanto na vida pessoal quanto na vida profissional.

Devido a esse fato, houve um grande aumento no número de interessados pelo conhecimento da programação e, consequentemente, o ensino de tal área tem se difundido cada vez mais. Entretanto, muitos dos interessados por tais técnicas não dispõem do tempo necessário ou da paciência e concentração para o aprendizado tradicional, ou seja, leituras extensas sobre os temas e longas sessões práticas para a aplicação das técnicas aprendidas.

Neste momento os jogos ganham força como disseminadores do conhecimento para os que buscam o primeiro contato com esta área, pois são uma forma divertida e rápida de se adquirir experiência básica sobre algo. Por ser uma forma simples e dinâmica de aprendizado o indivíduo encontra mais facilidade para encaixar o jogo em sua agenda do que ler um livro teórico sobre algo. Por isso que o jogo desenvolvido tenta *gamificar* uma plataforma de ensino.

Desta forma, visando proporcionar um ambiente facilitador do aprendizado dos conceitos de programação para indivíduos iniciantes ou com pouca experiência foi desenvolvido o jogo Phoenix Rising. Além disso a estrutura do código foi pensada de modo a facilitar a inserção de novas características ao jogo pelos indivíduos que têm certa experiência em programação, fazendo com que o projeto desenvolvido sirva para uma grande parte dos interessados em aprofundar o conhecimento.

## 1.2 Organização do Projeto

O projeto foi desenvolvido utilizando Godot na versão 3.1.1 stable, uma *game engine* que facilita a produção de jogos e possui uma linguagem própria chamada GDScript. Todo o código do jogo está mantido no GitHub, portanto o projeto é open source, o que facilita a contribuição pela comunidade.

Como um dos objetivos do projeto é disponibilizar o código fonte para melhorias serem implementadas, o código e comentários estão em inglês, seguindo as boas práticas de programação. Vale salientar também que a eficiência não foi principal ponto do projeto mas sim a legibilidade e a flexibilidade do código, portanto em algumas partes preferiu-se utilizar um pouco mais de memória e/ou processamento, embora tais escolhas não tenham grande impacto na jogabilidade.

## 1.3 Noção básica sobre o jogo

Para conseguir completar o objetivo o jogo *Phoenix Rising* funciona da seguinte forma:

O jogador deve resolver o quebra cabeças conectando os blocos da forma correta até que a Entrada esteja conectada com a Saída.

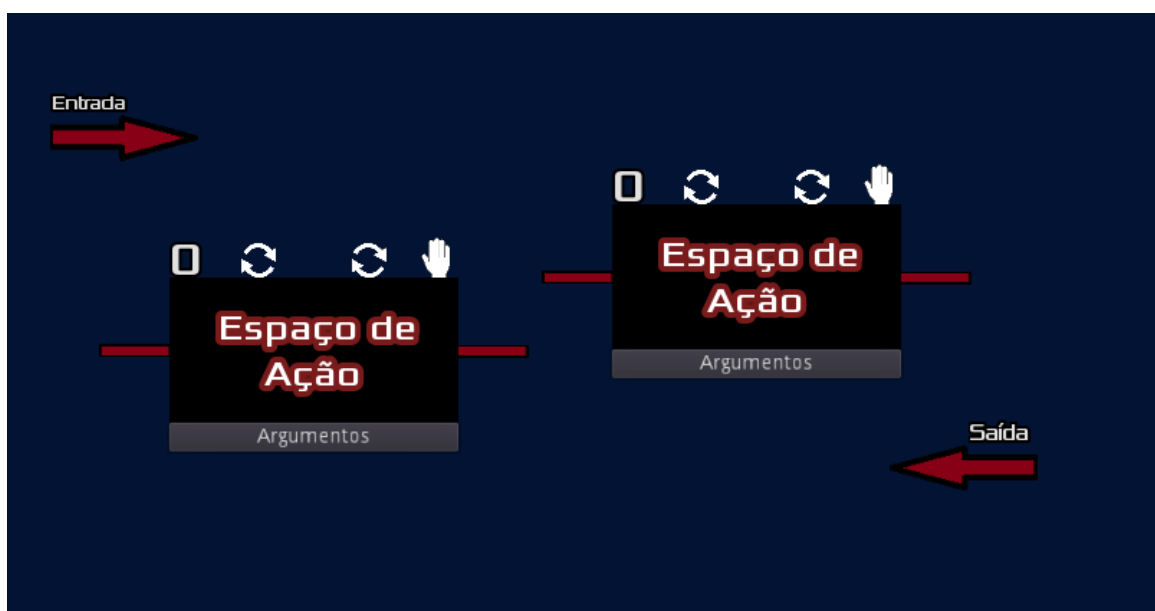


Figura 1.1: Jogo não conectado



**Figura 1.2:** *Jogo conectado*

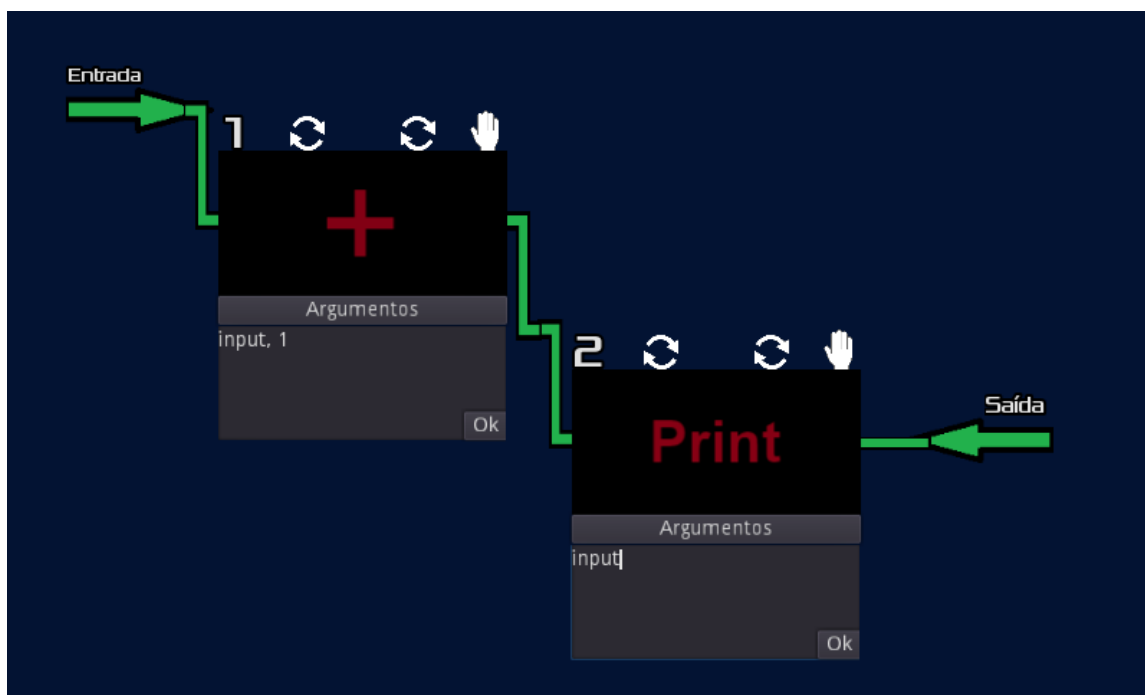
Desta forma o jogador deve compreender que, para criar um programa, é necessário pensar sobre a estrutura que o código terá antes de começar a utilizar os comandos, pois tentar criar um código apenas inserindo comandos sem pensar previamente em uma estrutura base leva a códigos confusos e que muitas vezes não funcionam corretamente. É claro que para sistemas maiores as reestruturações do modelo ocorrem com certa frequência, porém o objetivo deste jogo é apenas introduzir os conceitos básicos de programação.

Após ter o sistema conectado, o jogador deve utilizar os comandos que são disponibilizados no inventário, posicionados no canto inferior esquerdo da tela de jogo.



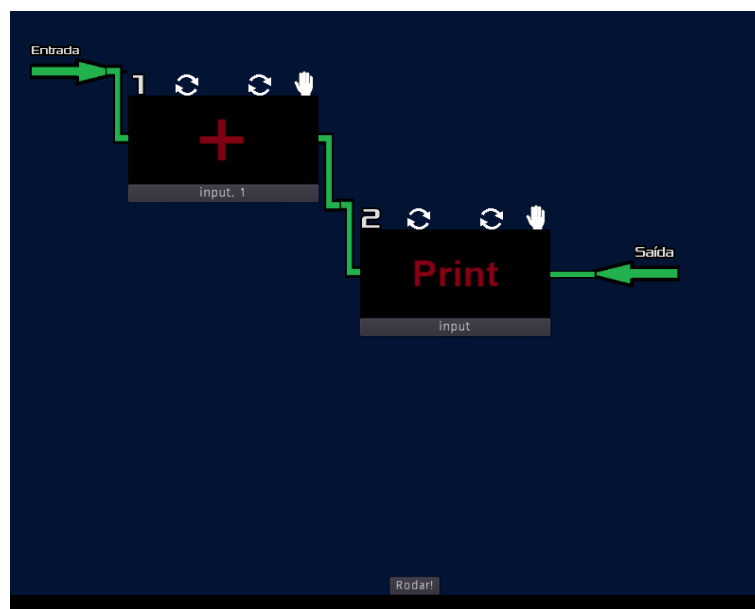
**Figura 1.3:** *Exemplo de comandos disponíveis*

Depois de posicionar os comandos, o jogador deve preencher os argumentos que cada comando recebe e então o sistema estará pronto para ser executado.



**Figura 1.4:** Jogador preenchendo os argumentos

Agora o sistema está pronto para ser executado.



**Figura 1.5:** Sistema pronto para execução

Para iniciar o processamento dos dados de entrada, ou seja, rodar o programa, basta o jogador clicar no botão *rodar!* e ficar atento à animação. No exemplo abaixo a primeira entrada era o número 7 e está sinalizada na animação pelo nome *Input:*.



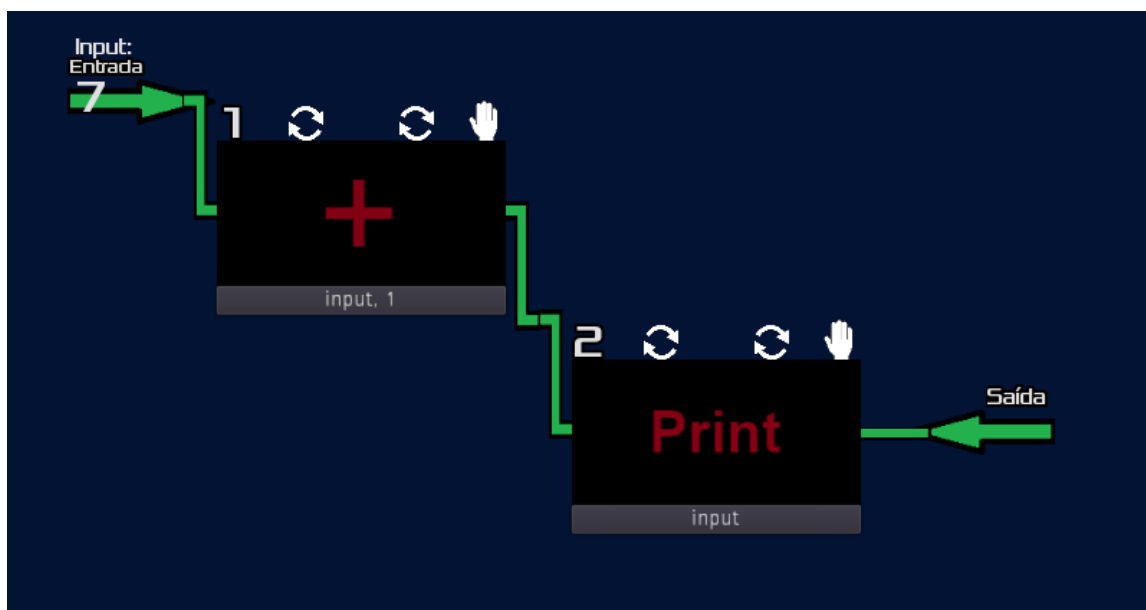


Figura 1.6: Início da execução do sistema

Agora o programa está rodando e o jogador pode acompanhar o que está acontecendo, pois o valor do *Input* será exibido constantemente na tela. Após passar por algum comando o valor de *Input* será modificado de acordo com a operação executada.

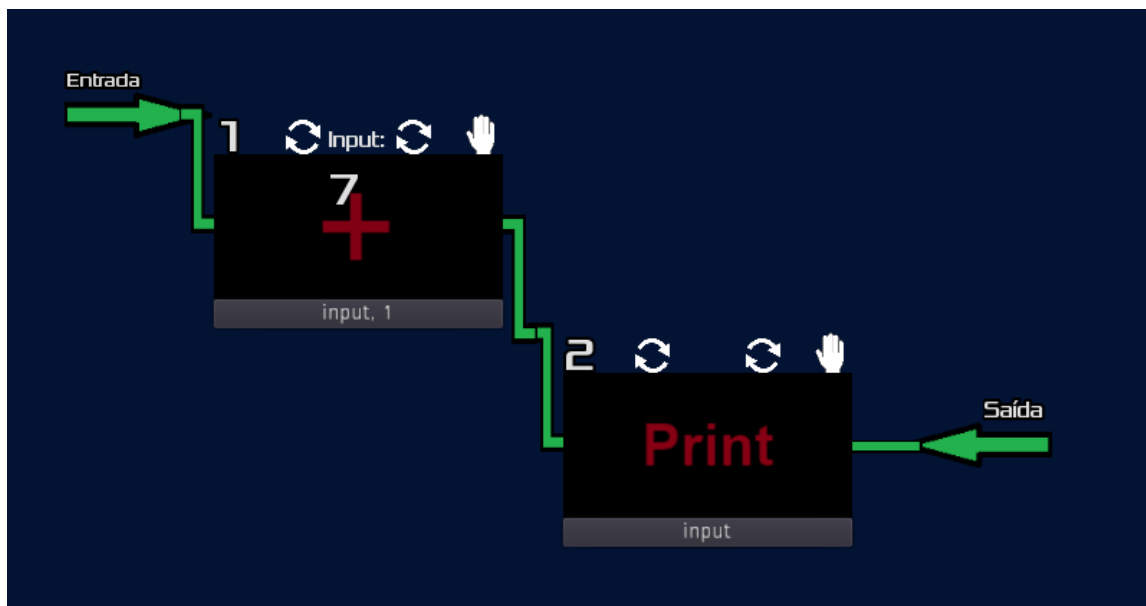
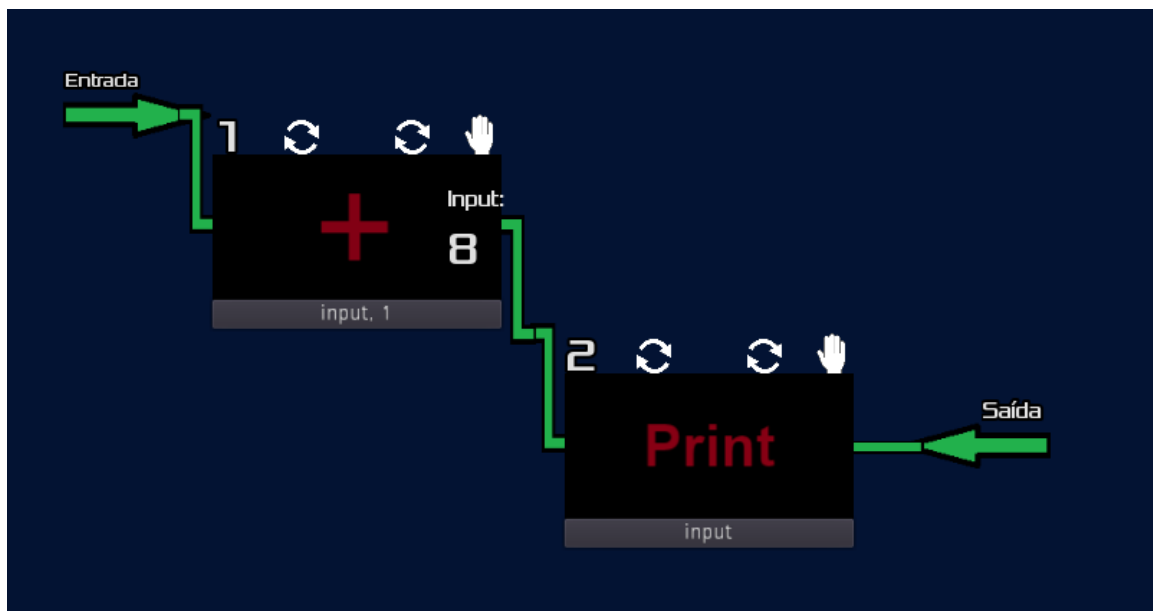


Figura 1.7: Valor do Input antes da operação

Note que após passar pelo comando de soma, o valor de *Input* será incrementado em 1, pois foi passado como argumento "input, 1", fazendo com que seja somado 1 ao valor corrente do *Input*.



**Figura 1.8:** Valor do Input após a operação

Esta maneira de conseguir acompanhar o que está acontecendo com os valores do programa enquanto é executada cada ação permite que o jogador entenda realmente como cada comando funciona, facilitando o aprendizado principalmente das instruções que controlam o fluxo de operação e loops.

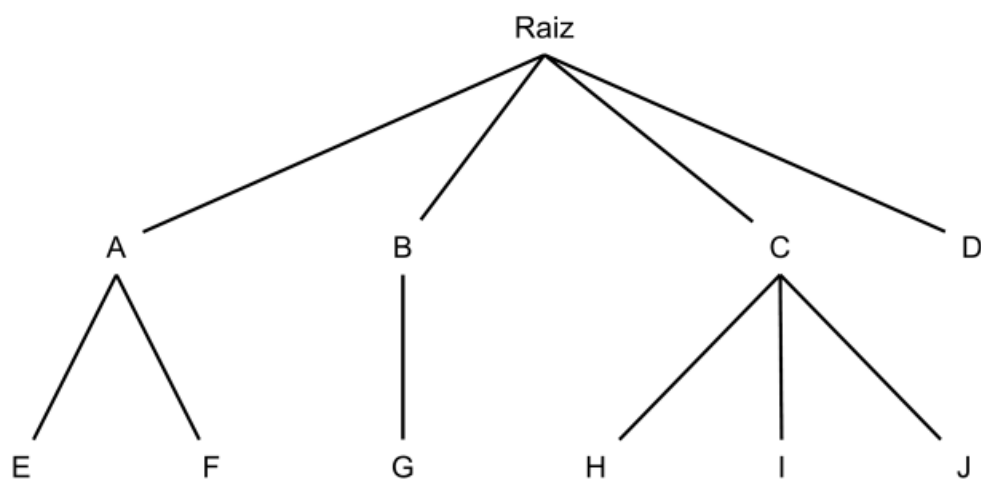
## Capítulo 2

# Conceitos Básicos

### 2.1 O Conceito de Árvore

Para facilitar o entendimento, deve-se entender um pouco sobre o que é uma árvore no escopo da programação, pois tal conceito aparecerá muitas vezes neste trabalho, entretanto a definição informal, passando apenas a ideia do funcionamento, bastará para entender este projeto.

Árvore refere-se a uma forma de estruturar os dados de um programa, informalmente pode ser definido como um conjunto de elementos que armazenam informações, por sua vez são os chamados nós. Toda árvore possui o elemento chamado raiz, que é primeiro nó, de onde a árvore começa, e que possui ligações para outros elementos denominados filhos, por sua vez também são nós.



**Figura 2.1:** *Exemplo de árvore*

Perceba que a árvore cresce para baixo, sendo que a raiz dá origem a tudo. Os nós A, B,

C, D são filhos da Raiz. Os nós E, F são filhos do nó A. O nó D é filho da Raiz e não tem filhos.

Como a estrutura dos projetos criados utilizando a *Godot Engine* é baseada em árvores, já é possível entender parte de como o jogo desenvolvido foi estruturado. Entretanto ainda é necessário explicar o que é uma *Game Engine*.

## 2.2 O que é *Game Engine*?

Uma *game engine* é um programa para computador com um conjunto de bibliotecas capaz de juntar e construir, em tempo real, todos os elementos de um jogo. Ela inclui motor gráfico para renderizar gráficos em 2D ou 3D, motor de física para detectar colisões e fazer animações, além de suporte para sons, inteligência artificial, gerenciamento de arquivos, programação, entre outros. Por conta dessas facilidades, a partir do uso de uma *game engine*, é possível criar um jogo do zero de maneira mais simples e replicar vários estilos jogos com mais facilidade.

Escolher a Godot para este projeto teve como motivação o grupo de extensão USPGameDev, além do aprendizado ser relativamente simples e de ser um *software open source* sob a licença MIT, desenvolvido de forma independente pela comunidade.

Como foi estabelecido o conhecimento sobre alguns termos gerais, agora é possível entender o básico de como funciona a *Godot Engine*.

## 2.3 Entendendo sobre a *Godot Engine*

A seguir estão as explicações dos conceitos básicos.

### 2.3.1 Nós

Nós são blocos de construção fundamentais para a criação de um jogo. Um nó pode executar uma variedade de funções especializadas. No entanto, qualquer nó fornecido sempre possui os seguintes atributos:

- Possui um nome.
- Possui propriedades editáveis.
- Ele pode receber um retorno de chamada (*callback*) para processar todos os quadros (*frames*).
- Pode ser estendido (para ter mais funções).
- Pode ser adicionado a outro nó como filho.

Perceba que o último atributo é muito importante, pois quando nós tem outros nós como filhos o conjunto se torna uma árvore, como foi explicado anteriormente. Em Godot, a capacidade de organizar nós dessa maneira cria uma ferramenta poderosa para organizar projetos. Como nós diferentes têm funções diferentes, combiná-los permite a criação de funções mais complexas, a partir disso *Phoenix Rising* foi criado.

### 2.3.2 Cenas

Uma cena é composta por um grupo de nós organizados hierarquicamente (em forma de árvore). Além disso, uma cena:

- Sempre tem um nó raiz.
- Pode ser salvo no disco e carregado de volta.
- Pode ser instanciado (Explicado adiante).

Executar um jogo significa executar uma cena. Um projeto pode conter várias cenas, mas para o jogo começar, uma delas deve ser selecionada como a cena principal.

Basicamente, o editor Godot é um editor de cenas. Possui muitas ferramentas para editar cenas 2D e 3D, bem como interfaces com o usuário, mas o editor é baseado no conceito de edição de uma cena e nos nós que a compõem.

### 2.3.3 Instâncias

Criar uma única cena e adicionar nós a ela pode funcionar para pequenos projetos, mas à medida que o projeto aumenta em tamanho e complexidade, o número de nós pode se tornar rapidamente incontável. Para resolver isso, Godot permite que um projeto seja separado em qualquer número de cenas. Isso fornece uma ferramenta poderosa que ajuda a organizar os diferentes componentes do seu jogo.

Em Cenas e nós, você aprendeu que uma cena é uma coleção de nós organizados em uma estrutura de árvore, com um único nó como raiz da árvore. Você pode criar quantas cenas quiser e salvá-las em disco. As cenas salvas dessa maneira são chamadas de "Cenas compactadas" (*packed scenes*) e têm uma extensão de nome de arquivo ".tscn".

A instanciação é muito utilizada em *Phoenix Rising*, portanto esta parte deve ficar mais clara conforme adentrarmos nos detalhes da estrutura e de implementação mais adiante.

### 2.3.4 SceneTree

Para entender melhor o que é uma *SceneTree* deve-se entender um pouco sobre o modo como *Godot* trabalha internamente.

Primeiro, a única instância que é executada no início pertence à classe *OS*. Depois, todos os drivers, servidores, linguagens de script, sistema de cenas e outros recursos são carregados.

Quando a inicialização estiver concluída, o sistema operacional precisará receber um *MainLoop* para executar. Até o momento, tudo isso funciona internamente (você pode verificar o arquivo "main.cpp" no código-fonte se estiver interessado em ver como isso funciona internamente).

Este *MainLoop* dá início ao programa do usuário, ou jogo. Essa classe possui alguns métodos, para inicialização, *callbacks* e *input*. Novamente, esse é um nível baixo e, ao fazer jogos em Godot, escrever seu próprio *MainLoop* raramente faz sentido.

A partir disso o sistema de cena fornece seu próprio loop principal para o *OS*, chamado de *SceneTree*. Isso é instanciado automaticamente e definido ao executar uma cena, sem a necessidade de fazer nenhum trabalho extra.

Agora que a *SceneTree* foi introduzida é importante saber que ela existe e possui algumas características, como:

- Contém o *Viewport* raiz, ao qual uma cena é adicionada como filha quando é aberta pela primeira vez para se tornar parte da *SceneTree*.
- Contém informações sobre os grupos e possui os meios para chamar todos os nós em um grupo ou obter uma lista deles.
- Contém algumas funcionalidades do estado atual do jogo, como definir o modo de pausa ou término de processos.

Desta forma, quando um nó é conectado, direta ou indiretamente, à *viewport* raiz, ele se torna parte da *SceneTree*. Quando os nós entram na Árvore da cena, eles se tornam ativos. Eles têm acesso a tudo o que precisam para processar, obter informações, exibir imagens em 2D e 3D, receber e enviar notificações, reproduzir sons, entre outros processamentos. Quando são removidos da árvore da cena, perdem essas habilidades, evitando alguns comportamentos indesejados.

A importância de se entender tudo isso, para este projeto, se dá pois a maioria das operações de nó em *Godot*, como desenhar 2D, processar ou obter notificações, são feitas seguindo a ordem que os nós estão na árvore.

O processo de tornar um nó ativo ao entrar na *SceneTree* se dá seguindo os passos:

1. Uma cena é carregada do disco ou criada por script.
2. A raiz dessa cena é adicionada como filha de *Viewport*, ou como filha de qualquer filha de *Viewport*.
3. Cada nó da cena recém-adicionada receberá a notificação "*enter\_tree*" na ordem de cima para baixo, ou seja, o pai é notificado e depois cada um de seus filhos.
4. Uma notificação extra, "*ready*" é fornecida por conveniência, quando um nó e todos os seus filhos estão dentro da cena ativa.

5. Quando uma cena (ou parte dela) é removida, eles recebem a notificação *"exit\_tree"* na ordem de baixo para cima, ou seja, os filhos são notificados e depois o pai.

### 2.3.5 Singleton

O sistema de cenas utilizado em *Godot*, embora poderoso e flexível, tem uma desvantagem: não há método para armazenar informações, por exemplo, pontuação do jogador (inclusive utilizado neste projeto), que é necessário para mais de uma cena.

Existem alternativas de implementação ao se deparar com estes problemas, porém na maioria dos casos o padrão *Singleton* irá consumir menos tempo e memória. Isso deve-se ao fato de *Singleton* ser uma ferramenta útil para resolver o caso de uso comum em que você precisa armazenar informações persistentes entre as cenas. No nosso caso, é possível reutilizar a mesma cena ou classe para vários *Singletons*, desde que eles tenham nomes diferentes.

Resumindo, usando esse conceito, você pode criar objetos que:

- Sempre estejam carregados e prontos para uso, independentemente da cena em execução no momento.
- Pode armazenar variáveis globais, como informações do jogador.
- Pode lidar com alternância de cenas e transições entre cenas.

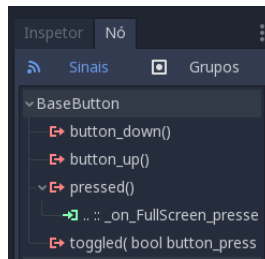
Vale lembrar também que o carregamento automático de nós e scripts pode nos dar essas características ao custo de processamento.

### 2.3.6 Sinais

Sinais permitem que um nó envie uma mensagem que outros nós possam ouvir e responder. Por exemplo, em vez de verificar continuamente um botão para ver se ele está sendo pressionado, o botão pode emitir um sinal quando é pressionado e assim quem receber o sinal poderá executar o que é necessário.

Servem, portanto, para dissociar os objetos do jogo, o que leva a um código melhor organizado, mais legível e limpo. Também faz com que os objetos do jogo não precisem estar sempre em conexão com outros, pois um nó pode emitir um sinal e apenas os nós interessados em tratar tal evento, aqueles que o emissor se conectou, recebam este sinal.

Alguns nós já vem com uma serie de sinais prontos para serem conectados, como visto na figura abaixo:



**Figura 2.2:** Sinais pré programados de um nó do tipo Button

Note que o sinal *pressed()* relativo ao nó chamado *FullScreen* já está conectado (note o ícone em verde, que simboliza a conexão).

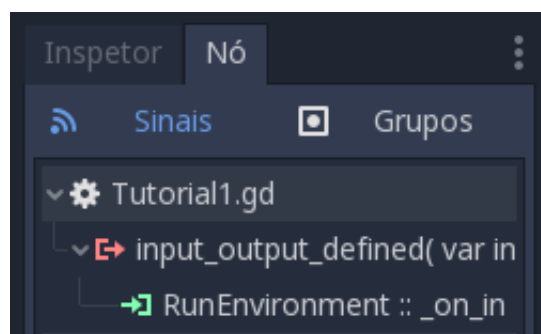
Entretanto nem sempre estes sinais cobrem a necessidade do projeto. Sendo assim é preciso criar o próprio sinal, utilizando código. Veja o exemplo abaixo:

```
signal input_output_defined(input, output)
```

**Figura 2.3:** Sinal *input\_output\_defined* criado por código

Veja que foi criado o sinal chamado *input\_output\_defined* que carrega dois parâmetros: *input* e *output*.

Depois pode-se conectar o sinal utilizando a interface de programação que o *Godot* oferece ou conectá-lo via código como visto nas imagens a seguir:



**Figura 2.4:** Sinal *input\_output\_defined* conectado via interface

```
11 func execute(input, arguments, action_number):
12     connect("variable_changed", get_parent().get_node("VariablesMap"), "_on_variable_changed")
13     if (not arguments.empty()):
```

**Figura 2.5:** Sinal *variable\_changed* conectado via código (linha 12.)



### 2.3.7 *GDScripts*

*GDScript* é uma linguagem de programação de alto nível e tipagem dinâmica usada para criar e modelar o comportamento dos nós. Ela usa uma sintaxe semelhante ao *Python* (os blocos são baseados em indentação e muitas palavras-chave são semelhantes). Seu objetivo é ser otimizada e fortemente integrada ao Godot Engine, permitindo grande flexibilidade para criação e integração de conteúdo.

Quando adicionado ao nó o script adiciona comportamento a ele, controlando seu funcionamento e as interações com outros nós: filhos, pais, irmãos e assim por diante. O escopo local do script é o próprio nó. Em outras palavras, o script herda as funções fornecidas por esse nó.



## Capítulo 3

# Bibliografia

### REFERÊNCIAS BIBLIOGRÁFICAS

<https://www.cse.unr.edu/~sushil/class/gas/papers/GameAIp27-lewis.pdf>

[https://docs.godotengine.org/en/3.1/getting\\_started/step\\_by\\_step/scenes\\_and\\_nodes.html](https://docs.godotengine.org/en/3.1/getting_started/step_by_step/scenes_and_nodes.html)

[https://docs.godotengine.org/en/3.1/getting\\_started/step\\_by\\_step/instancing.html](https://docs.godotengine.org/en/3.1/getting_started/step_by_step/instancing.html)

[https://docs.godotengine.org/en/3.1/getting\\_started/scripting/gdscript/gdscript\\_basics.html#doc-gdscript](https://docs.godotengine.org/en/3.1/getting_started/scripting/gdscript/gdscript_basics.html#doc-gdscript)

[https://docs.godotengine.org/en/latest/getting\\_started/step\\_by\\_step/scene\\_tree.html](https://docs.godotengine.org/en/latest/getting_started/step_by_step/scene_tree.html)

[https://docs.godotengine.org/en/3.1/getting\\_started/step\\_by\\_step/signals.html](https://docs.godotengine.org/en/3.1/getting_started/step_by_step/signals.html)



# Índice Remissivo

## C

Código-fonte, *veja* Floats

Captions, *veja* Legendas

## E

Equações, *veja* Modo Matemático

## F

Fórmulas, *veja* Modo Matemático

Figuras, *veja* Floats

Floats

Algoritmo, *veja* Floats, Ordem

## I

Inglês, *veja* Língua estrangeira

## P

Palavras estrangeiras, *veja* Língua es-

trangeira

## R

Rodapé, notas, *veja* Notas de rodapé

## S

Subcaptions, *veja* Subfiguras

Sublegendas, *veja* Subfiguras

## T

Tabelas, *veja* Floats

## V

Versão corrigida, *veja* Tese/Dissertação,  
versões

Versão original, *veja* Tese/Dissertação,  
versões