

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
19 de novembro 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	Exemplo de um nível	2
1.2	Exemplo de entrada	3
1.3	Objetivos	3
1.4	Tempo restante no início do nível	3
1.5	Pontuação obtida	4
1.6	Entrada e Saída	5
1.7	Um espaço de ação	5
1.8	Ícones de arrastar	6
1.9	Ícone de trocas de conexões	6
1.10	Exemplos de conexões alteradas	6
1.11	Posição na ordem de ações (1)	6
1.12	Espaço para Argumentos	7
1.13	Espaço de Ação com Comando <i>Print</i>	7
1.14	Inventário com Comandos	7
1.15	Processo visual para If	8
1.16	Processo visual para Else	8
1.17	Valores das Variáveis	9
1.18	Processo visual para mudança nas variáveis	9
1.19	Setas de Entrada e Saída	10
1.20	Jogo não conectado	11
1.21	Jogo conectado	11
1.22	Exemplo de comandos disponíveis	12
1.23	Jogador preenchendo os argumentos	12
1.24	Sistema pronto para execução	13
1.25	Início da execução do sistema	13
1.26	Valor do <i>Input</i> antes da operação	14
1.27	Valor do <i>Input</i> após a operação	14
2.1	Exemplo de árvore	15

2.2	Logo da <i>Godot game engine</i>	16
2.3	Exemplo de nós	17
2.4	Símbolo que indica Cena Instanciada	18
2.5	Cena Tutorial	19
2.6	Cena InputOutput	19
2.7	Exemplo de instanciação	19
2.8	Sinais pré programados de um nó do tipo <i>Button</i>	22
2.9	Sinal <i>input_output_defined</i> criado por código	22
2.10	Sinal <i>input_output_defined</i> conectado via interface	22
2.11	Sinal <i>variable_changed</i> conectado via código (linha 12.)	22
3.1	Árvore da cena Inventory	26
3.2	Funções do script do nó Inventory	26
3.3	Variáveis do script do nó Inventory	26
3.4	Variáveis do Grid	27
3.5	inicialização do Grid	27
3.6	Método de inserção de um item	27
3.7	Marcação das posições do grid	28
3.8	Cena MovableActionSpace	28
3.9	Cena ActionSpace	28
3.10	Sistema gerador da Árvore de Execução	30
3.11	Esquema de Árvore de Execução	30
3.12	Cena Visual Process	32
3.13	Variáveis Visual Process	33
3.14	Variáveis Visual Process	34
3.15	Árvore da cena RunEnvironment	35
3.16	Funções de RunEnvironment.gd	35
3.17	Função do botão Rodar!	36
3.18	Sistema gerador da Árvore de Execução não Conectada	37
3.19	Esquema de Árvore de Execução não Conectada	37

Lista de Tabelas

Lista de Programas

Sumário

1	Introdução	1
1.1	Motivação e Objetivos	1
1.2	Organização do Projeto	2
1.3	Objetivo Dentro do Jogo	2
1.4	Elementos do Jogo	4
1.4.1	Entrada e Saída	4
1.4.2	Espaço de Ação	5
1.4.3	Inventário e Comandos	7
1.4.4	Setas de Entrada e Saída	9
1.4.5	Botões	10
1.5	Forma de Jogar	10
2	Conceitos Básicos	15
2.1	O Conceito de Árvore	15
2.2	O que é <i>Game Engine</i> ?	16
2.3	Entendendo sobre a <i>Godot Engine</i>	16
2.3.1	Nós	17
2.3.2	Cenas	18
2.3.3	Instâncias	18
2.3.4	<i>SceneTree</i>	20
2.3.5	Singleton	21
2.3.6	Sinais	21
2.3.7	<i>GDScripts</i>	23
3	Implementação do Projeto	25
3.1	Inventário	25
3.2	Espaço de Ação	28
3.3	Processo Visual	31
3.4	Ambiente de Execução	35

3.5	Nível Base	37
3.6	Cenas Genéricas	37
3.7	Comportamento dos Comandos	37
3.8	Ajuda ao Usuário	37
4	Bibliografia	39
Apêndices		
Anexos		
	Índice Remissivo	41

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, conseqüentemente, 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*¹ 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.

¹Uso de mecânicas e dinâmicas de jogos para engajar pessoas, resolver problemas e melhorar o aprendizado, motivando ações e comportamentos em ambientes fora do contexto de jogos.

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 Objetivo Dentro do Jogo

Para que seja mais fácil entender o projeto como um todo, esta seção explicará o objetivo que o jogador deve alcançar ao jogar *Phoenix Rising*. A imagem abaixo exemplifica um nível do jogo.



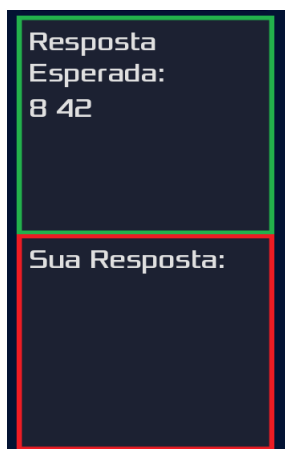
Figura 1.1: Exemplo de um nível

O jogador irá receber dados que serão mostrados dentro do retângulo azul, posicionado no lado esquerdo da tela. O exemplo abaixo mostra um nível que fornece ao jogador os números 7 e 41 como dados iniciais.

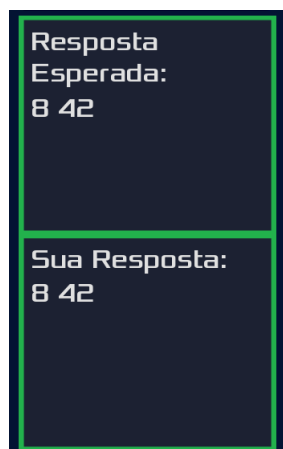


Figura 1.2: *Exemplo de entrada*

O objetivo do jogador é conseguir reproduzir o que está no retângulo verde, posicionado no lado direito da tela, chamado "Resposta Esperada". No exemplo abaixo o nível exige que o jogador reproduza os números 8 e 42. Após o programa do jogador ser executado a saída que ele obteve aparecerá abaixo de "Sua Resposta" que está no retângulo vermelho (fig(a)) e caso o resultado em "Sua Resposta" for igual ao que está em "Resposta Esperada" o retângulo também ficará verde (fig (b)).



(a) *Objetivo não alcançado*



(b) *Objetivo alcançado*

Figura 1.3: *Objetivos*

O jogador deve cumprir o objetivo dentro do tempo limite para acumular pontos. Este tempo é mostrado constantemente na tela e sempre iniciará com 120 segundos restantes.



Figura 1.4: *Tempo restante no início do nível*

Ao concluir o nível o jogador ganhará pontos iguais ao valor de tempo que restava ao apertar o botão "rodar!", obviamente os pontos só serão obtidos caso o objetivo seja alcançado. Este fato está relacionado com a duração do processo visual que o jogo possui e será explicado mais adiante, por hora deve-se entender que existe um sistema de pontuação.



Figura 1.5: Pontuação obtida

Note que, se o jogador tinha zero pontos quando concluiu este nível, o programa criado que alcançou o objetivo foi executado quando ainda havia 101 segundos restantes, porém a animação continuou executando, consequentemente o tempo continuou correndo, para que o usuário pudesse visualizar o que está acontecendo durante a execução do programa criado e aprender como funciona cada comando.

Portanto o objetivo final de *Phoenix Rising* é completar o maior número de níveis no menor tempo possível para maximizar o somatório de pontos. Para isso o jogador deve aprender a mecânica de jogo, o que e como cada comando executa sua instrução e como montar o quebra cabeça dos diferentes níveis.

1.4 Elementos do Jogo

Agora que o objetivo do jogo foi explicitado, deve-se entender quais elementos estão envolvidos para que o jogador possa concluir o desafio.

1.4.1 Entrada e Saída

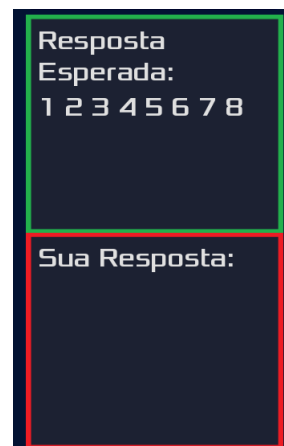
Estes termos são recorrentes na computação e geralmente são chamados de *Input*² e *Output*³ respectivamente. Neste jogo a entrada e saída estão delimitadas pelos retângulos coloridos e servem para mostrar para o jogador o que ele receberá para processar e o que ele deve produzir com o código gerado.

²Dados fornecidos para o sistema processar

³Dados que o sistema gera após o processamento



(a) Exemplo de Entrada



(b) Exemplo de Saída

Figura 1.6: Entrada e Saída

1.4.2 Espaço de Ação

Os espaços de ação são as áreas móveis do jogo que permitem montar o quebra-cabeças e que recebem os comandos que executarão as ações de processamento dos dados de entrada. Portanto este é um dos principais itens do jogo.

**Figura 1.7:** Um espaço de ação

O jogador pode:

- Movimentar o espaço de ação ao clicar no ícone de arrastar.



(a) Ícone de arrastar não pressionado (b) Ícone de arrastar pressionado

Figura 1.8: Ícones de arrastar

- Modificar as conexões de entrada e saída ao clicar nos ícones de troca de conexões.



Figura 1.9: Ícone de trocas de conexões



Figura 1.10: Exemplos de conexões alteradas

- Verificar qual a posição do espaço de ação na sequência de operações.



Figura 1.11: Posição na ordem de ações (1)

- Preencher os *argumentos*⁴ necessários para os diferentes comandos

⁴É um valor, proveniente de uma variável ou de uma expressão mais complexa, que pode ser passado para um comando (sub-rotina). Um comando utiliza os valores atribuídos aos parâmetros para alterar o seu comportamento em tempo de execução.

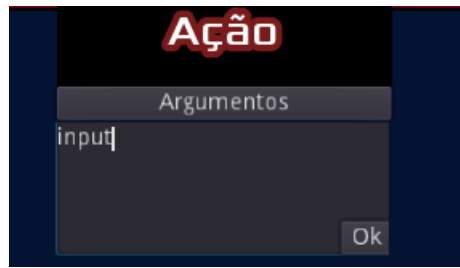


Figura 1.12: Espaço para Argumentos

- Posicionar o comando a ser utilizado no respectivo espaço de ação.



Figura 1.13: Espaço de Ação com Comando Print

Note que, dentro do jogo, um espaço de ação não pode ser movimentado caso um comando esteja posicionado, isso faz com que o jogador tenha que completar o quebra cabeças antes de pensar quais comandos serão utilizados.

1.4.3 Inventário e Comandos

Um comando é uma ação que processa o dado de uma forma específica de acordo com os argumentos que recebe, portanto todo comando possui nome e uma função. Os comandos do jogo estão em vermelho na imagem abaixo.



Figura 1.14: Inventário com Comandos

Os três primeiros comandos na imagem são de soma, subtração e multiplicação respectivamente. Estes comandos executam as operações básicas como conhecemos e apenas a operação de soma funciona como operador de concatenação caso o dado a ser processado seja uma string.

O quarto comando da sequência, chamado *Print*, escreve na saída "Sua Resposta" o valor do *Input* ou de uma variável do programa, dependendo de qual argumento passado.

O quinto comando da sequência, chamado *Pass*, serve apenas para conectar o sistema sem executar nenhum processamento dos dados. O sexto comando da sequência, chamado *If/Else*, serve como controle de fluxo do programa, ou seja, o comando recebe como argumento uma expressão, nomeada condição, e durante a execução o comando *If/Else* avalia se tal condição é verdadeira ou falsa, executando o ramo de ação referente ao resultado da avaliação.

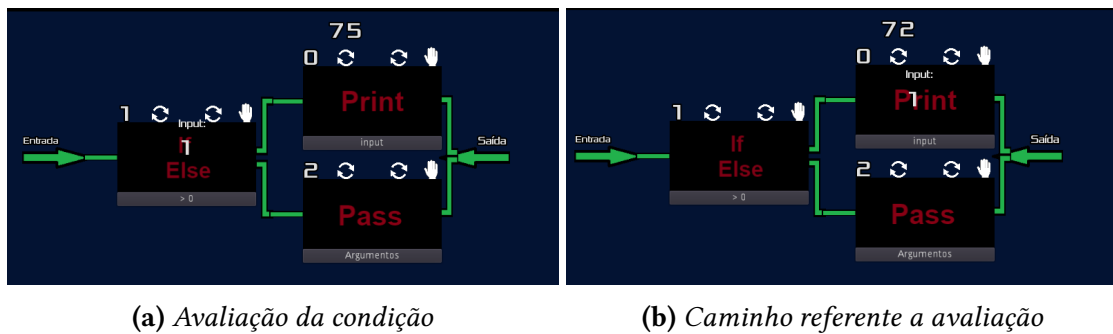


Figura 1.15: Processo visual para *If*

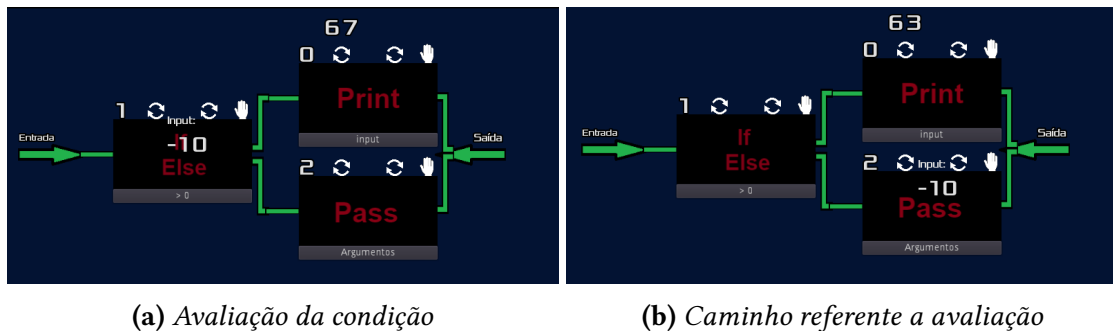


Figura 1.16: Processo visual para *Else*

Nos exemplos acima o input segue o caminho de cima caso ele seja maior que zero, caso contrário seguirá o caminho de baixo. Para o sistema ter esse comportamento basta passar como argumento para o comando *If/Else* " > 0 ".

O sétimo e oitavo comandos, chamados *A* e *B* respectivamente, são variáveis e podem armazenar informações do programa para serem utilizadas posteriormente. Além disso o jogador pode acompanhar os valores de *A* e *B* durante a execução do programa olhando para a região de *Valores das Variáveis* localizada no canto superior direito da tela.

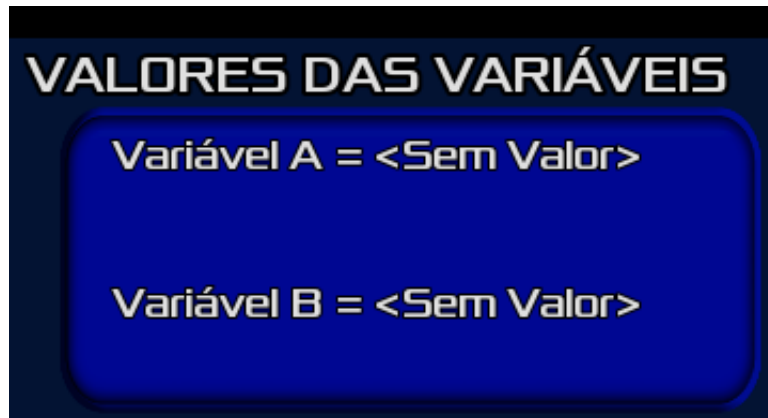
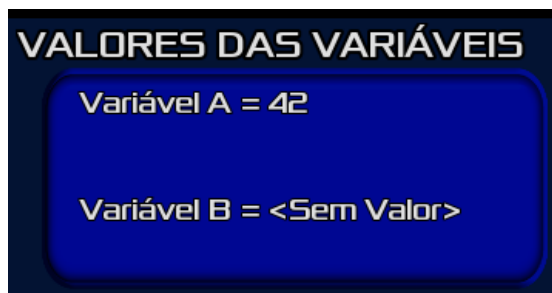


Figura 1.17: Valores das Variáveis

Conforme a execução do programa estes valores serão modificados, veja na imagem abaixo.



(a) Atribuição do valor 42 para variável A (b) Mudança na região de Valores das Variáveis

Figura 1.18: Processo visual para mudança nas variáveis

Note também que o *Input* era vazio, portanto valores de variáveis podem ser valores fixos atribuídos pelo jogador, como o exemplo acima mostra, ou podem ser dinâmicos, ou seja, o jogador pode armazenar em uma variável o valor corrente do *Input*.

O nono comando, chamado *Error*, não deve ser utilizado, pois só aparece caso a importação de algum dos comandos anteriores dê errado. Por conta disso o comando de erro não possui comportamento algum e, se tudo der certo durante o jogo, este comando não aparecerá em nenhum momento.

1.4.4 Setas de Entrada e Saída

Estas setas são as conexões iniciais que o jogador deve se preocupar. O programa do jogador irá receber os valores disponibilizados a serem processados a partir da *Seta de Entrada*, portanto a primeira conexão que deve ser feita é entre um espaço de ação e esta seta. Já a *Seta de Saída* será a última conexão que o jogador terá que fazer, pois o programa só estará apto a ser executado quando existir uma conexão entre as duas setas.



Figura 1.19: Setas de Entrada e Saída

1.4.5 Botões

Na tela de jogo constam os seguintes botões cujo comportamento está especificado ao lado:

- **Tela Cheia** - Coloca o jogo em tela cheia.
- **Menu Principal** - Retorna ao menu principal
- **Velocidade da Animação** - Modifica a velocidade da animação de execução do programa.
- **Rodar!** - Inicia a execução do programa criado caso o sistema esteja conectado e não haja erro na utilização dos comandos.
- **Reiniciar Nível** - Recomeça o nível atual.
- **Próximo Nível** - Inicia o nível seguinte do nível atual.

1.5 Forma de Jogar

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 Seta de Entrada esteja conectada com a Seta de Saída.

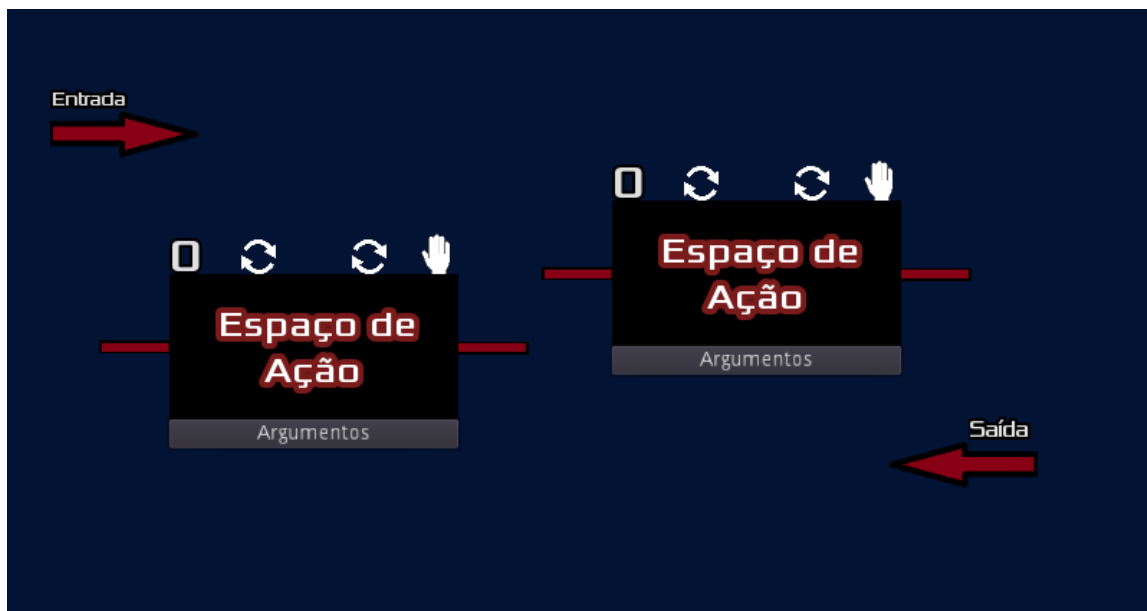


Figura 1.20: Jogo não conectado

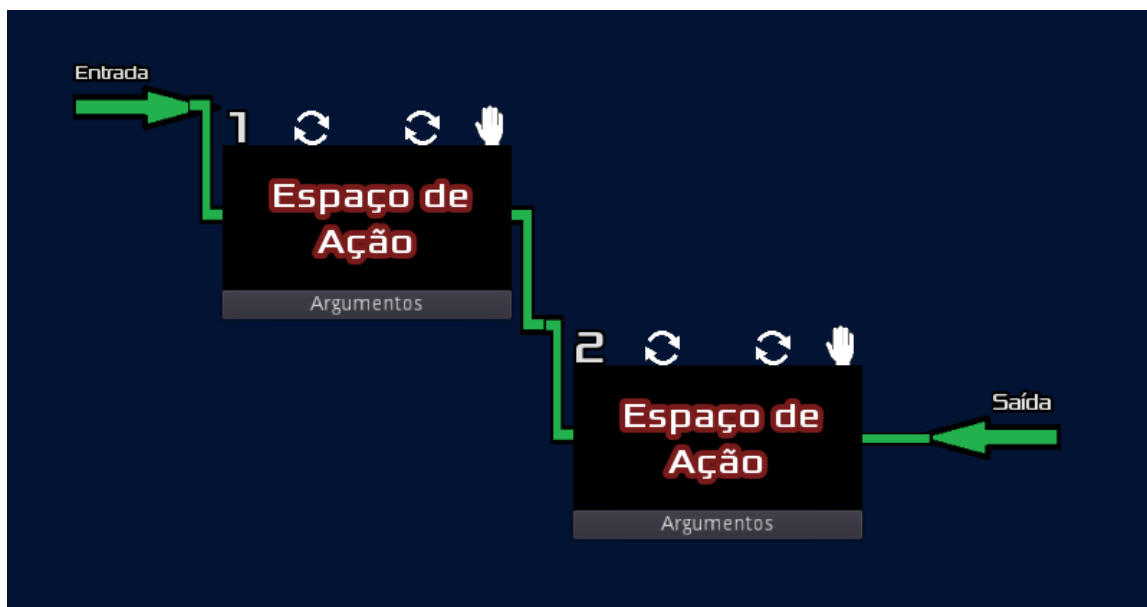


Figura 1.21: Jogo conectado

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 1.22: *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.

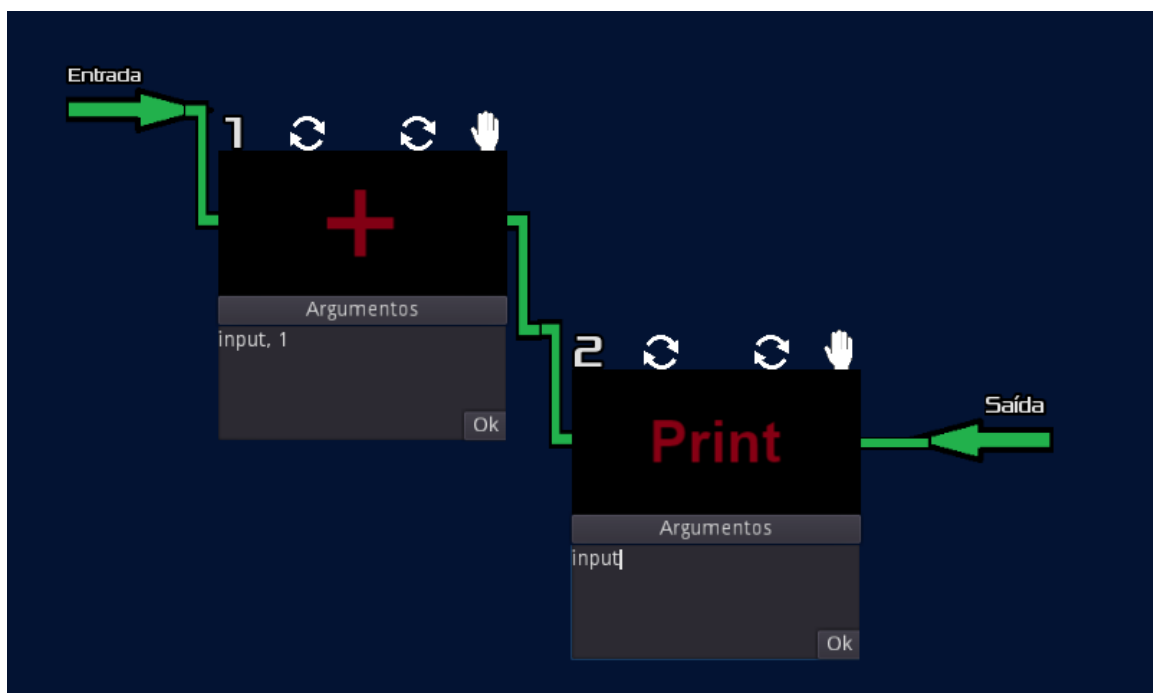


Figura 1.23: *Jogador preenchendo os argumentos*

Agora o sistema está pronto para ser executado.

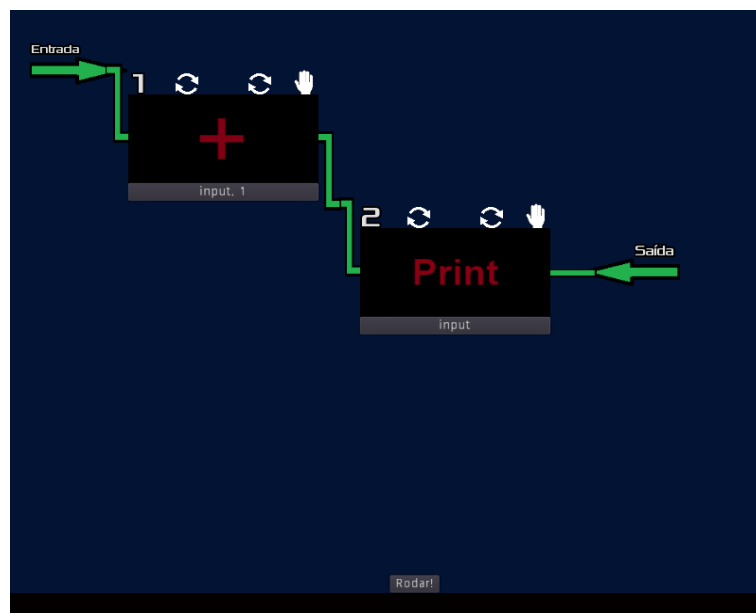


Figura 1.24: Sistema pronto para execução

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*:

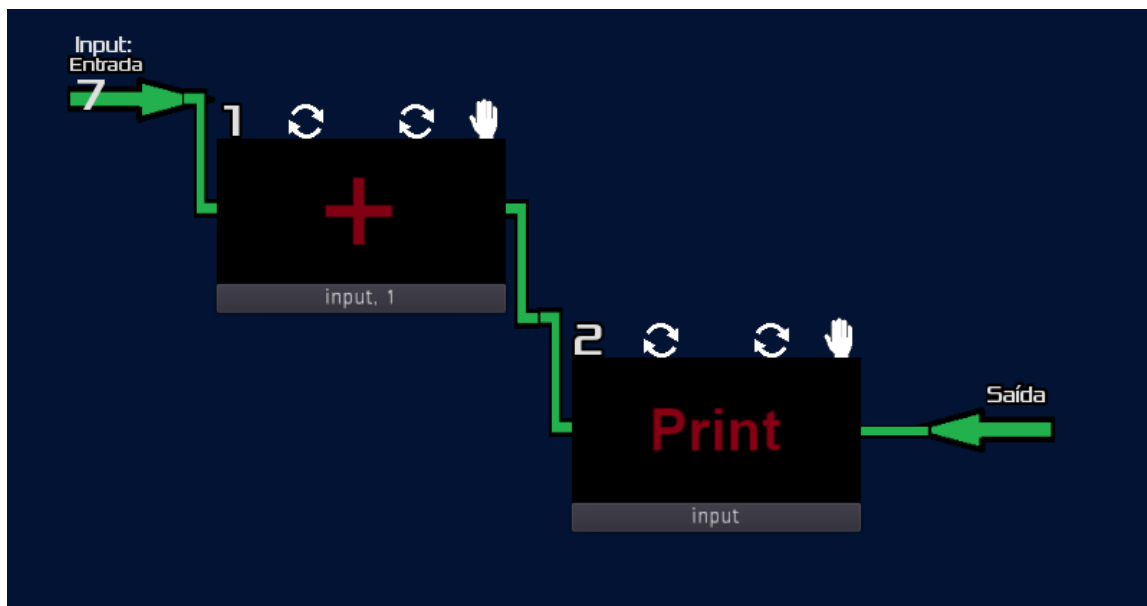


Figura 1.25: Início da execução do sistema

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.

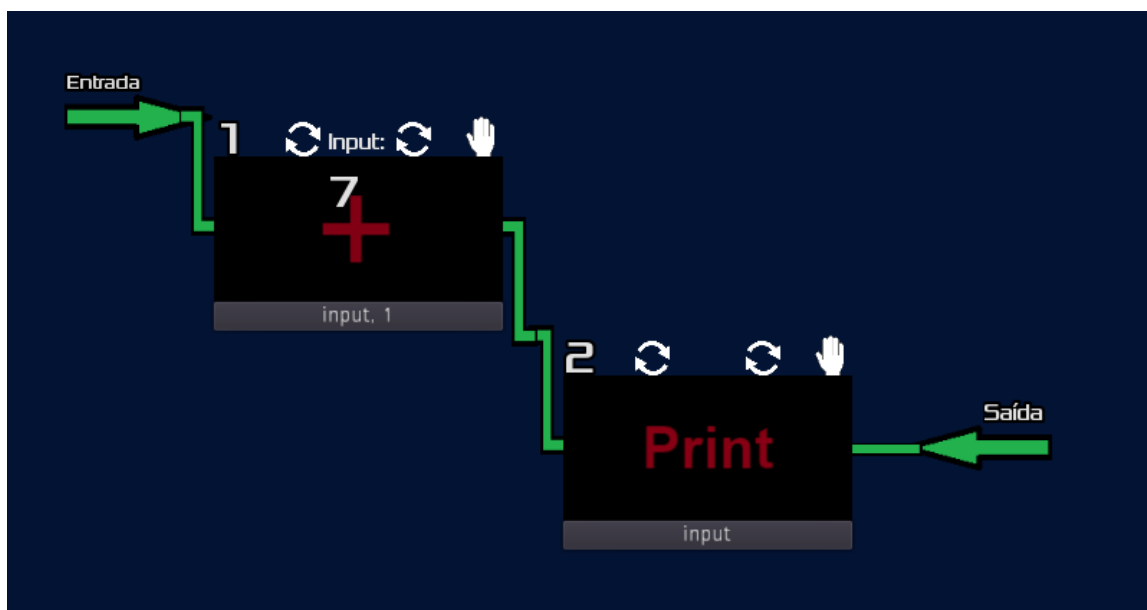


Figura 1.26: Valor do Input antes da operação

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*.

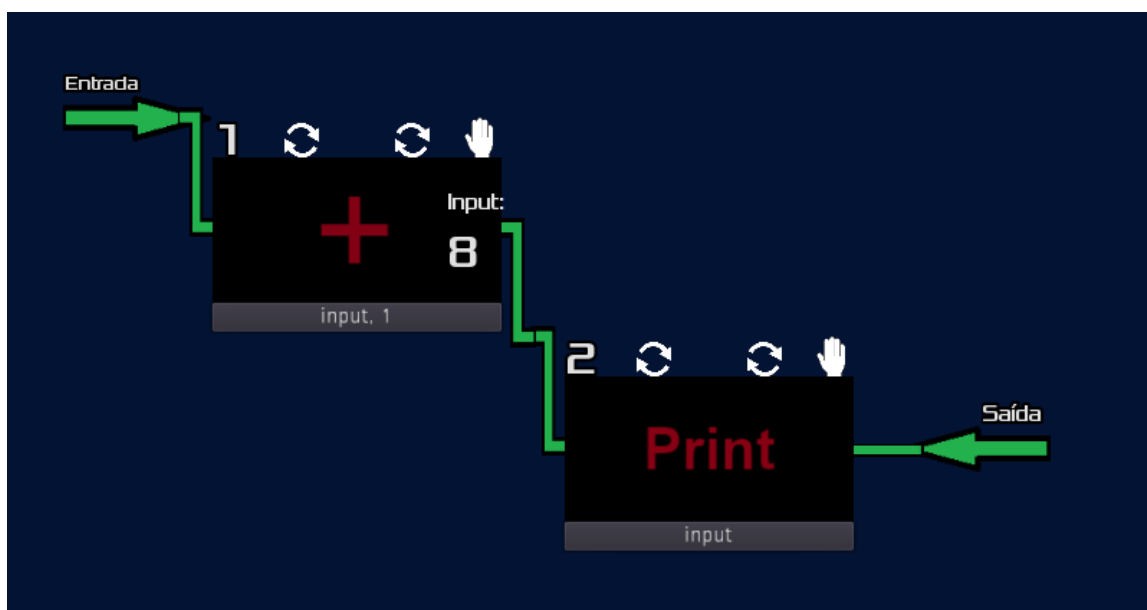


Figura 1.27: Valor do Input após a operação

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.

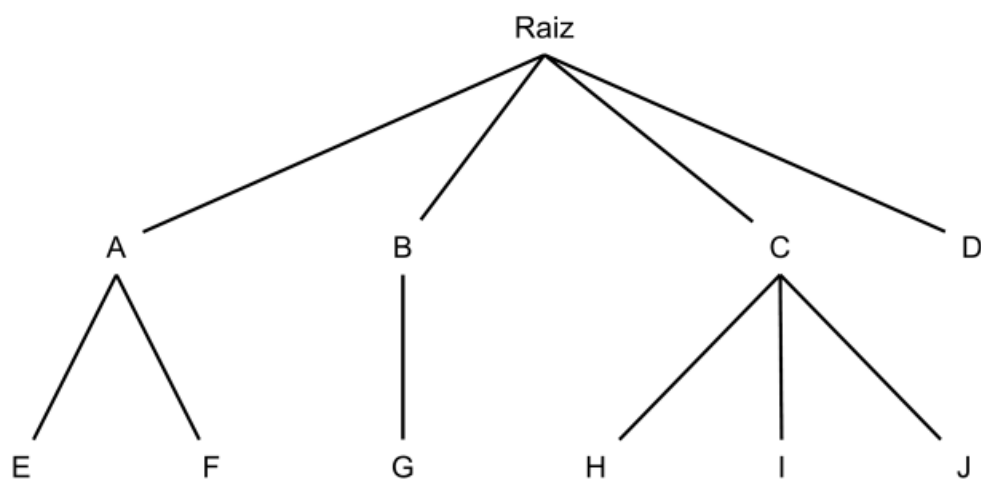


Figura 2.1: *Exemplo de árvore*

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.

Pelas facilidades com a estrutura e organização do código, a linguagem própria e por ser um *software open source* sob a licença MIT, desenvolvido de forma independente pela comunidade, foi escolhido a *Godot game engine* para este projeto, tendo como motivação extra o grupo de extensão de jogos da Universidade de São Paulo, chamado USPGameDev.



Figura 2.2: Logo da Godot game engine

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*

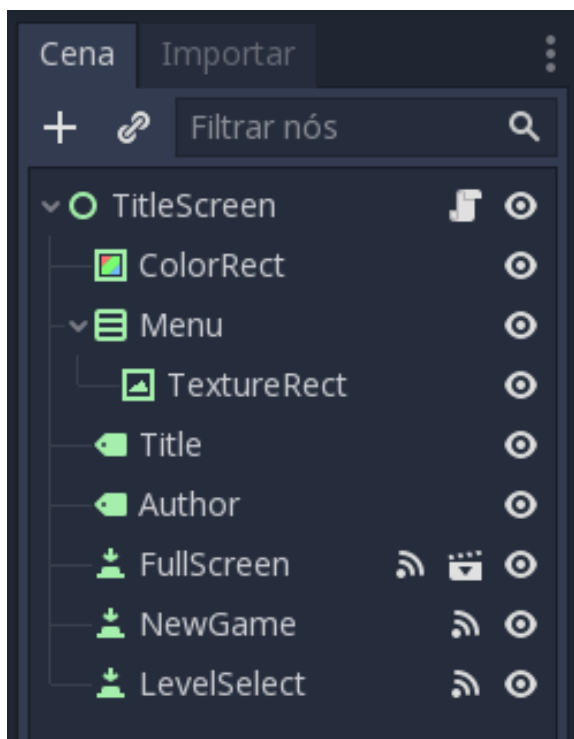
A seguir estão as explicações dos conceitos básicos.

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.



Note que na imagem ao lado existem vários nós diferentes, porém todos são verdes, isso se deve ao fato de todos herdarem da classe *Control*, nós que compõem a interface de usuário.

Os ícones do lado esquerdo de cada nome dos nós indicam a qual classe filha pertence o nó escolhido. De uma maneira informal pode-se entender isso como a especialização deste nó dentro da classe *Control*.

O nó chamado *Menu* é filho da raiz da cena, chamada *TitleScreen*, e também tem um filho chamado *TextureRect*, exemplificando o último atributo citado acima.

Figura 2.3: Exemplo de nós

A partir do exemplo acima é possível entender como a organização dos nós contrói a árvore de cena, conceito explicado na próxima seção.

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 salva no disco e carregada de volta.
- Pode ser instanciada.

Neste caso o nó ficará marcado com o símbolo:



Figura 2.4: *Símbolo que indica Cena Instanciada*

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 incontrolável. 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".

Veja no exemplo abaixo que a cena *Tutorial* é composta por vários outros nós, dentre eles o nó *InputOutput* também é uma cena e foi instanciada.

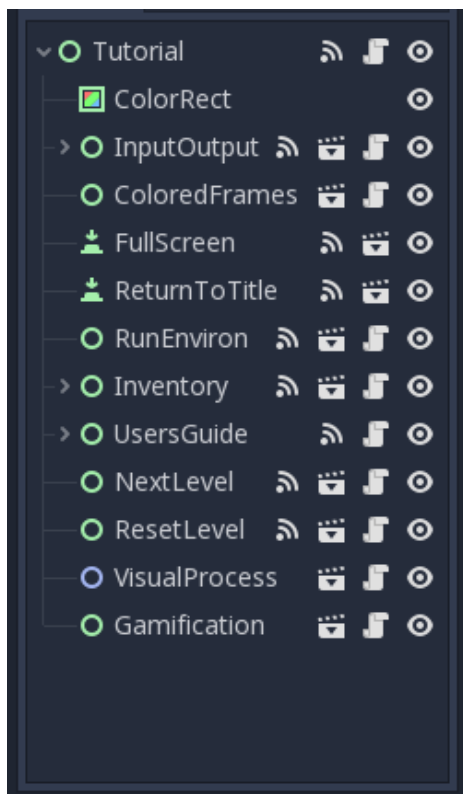


Figura 2.5: Cena Tutorial

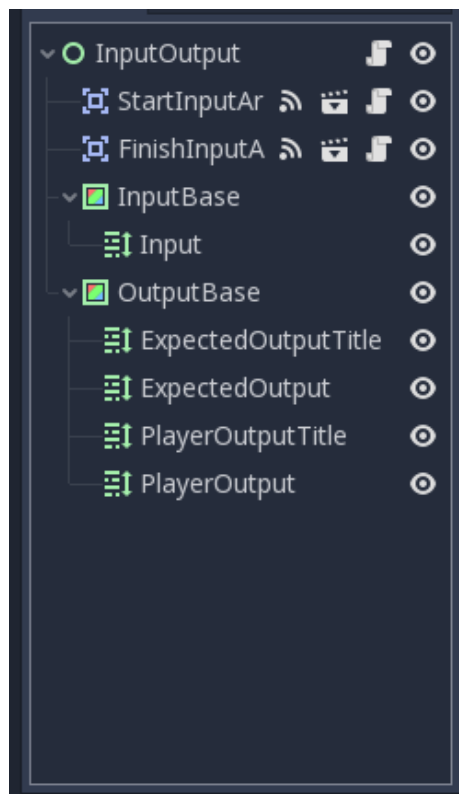


Figura 2.6: Cena InputOutput

Instanciar uma cena utilizando *Godot* é bem fácil, basta clicar no botão de instanciar cena e selecionar qual cena salva que se deseja instanciar.

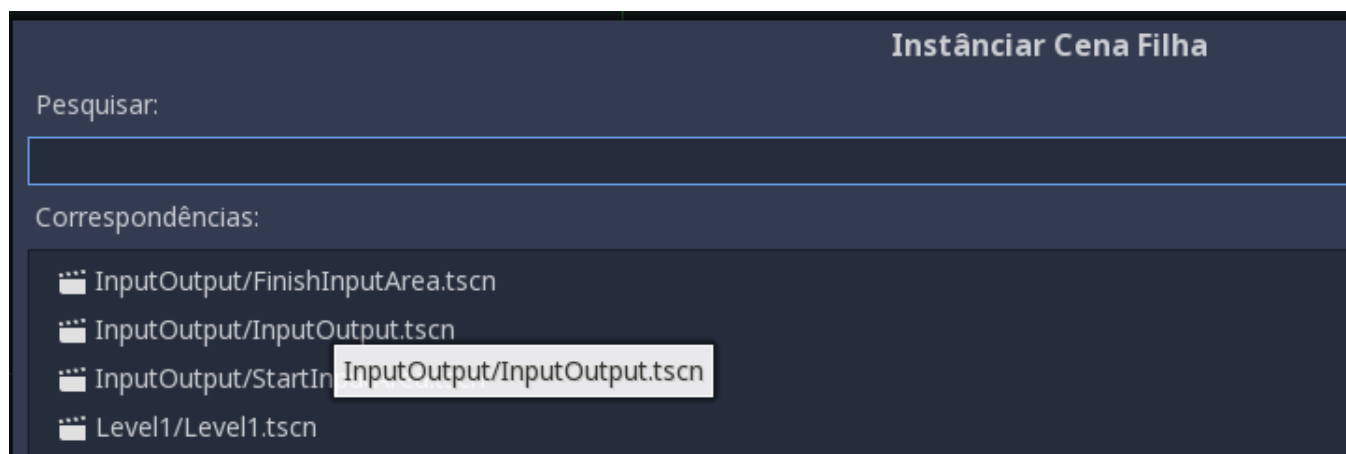


Figura 2.7: Exemplo de instanciação

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* da 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 *Singltons*, 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:

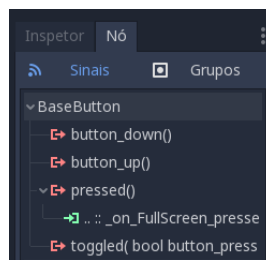


Figura 2.8: Sinais pré programados de um nó do tipo Button

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 2.9: 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:

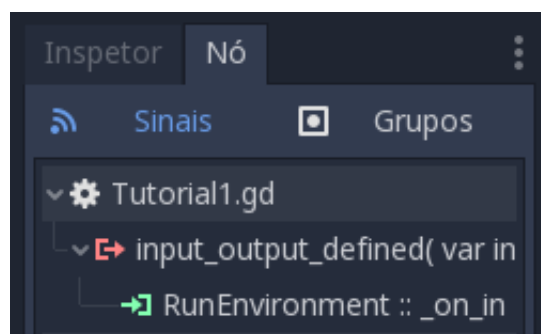


Figura 2.10: Sinal *input_output_defined* conectado via interface

```
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 2.11: 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

Implementação do Projeto

Agora que foram definidas as mecânicas de jogo do ponto de vista de um jogador e explicado que uma das intenções do projeto é facilitar o aprendizado de um iniciante em computação é preciso explicar melhor os detalhes da implementação para concluir a segunda intenção do projeto que é permitir a extensão do jogo por alguém que já conheça um pouco de programação ou esteja interessado em modificar o código fonte, inserindo novas funcionalidades.

Para isso, essa seção divide a explicação dos arquivos do jogo agrupando por funcionalidades, assim entender o código de implementação torna-se mais simples.

3.1 Inventário

Dentro dos arquivos do jogo (diretório *Phoenix Rising*) é encontrado na pasta *Generic-GameScenes* e administra os comandos disponíveis e a movimentação deles na tela.

A seguir estão imagens com os nós que compõem a cena, os métodos do script atrelado ao nó raiz, chamado *Inventory* e as variáveis mais importantes para entender o código.

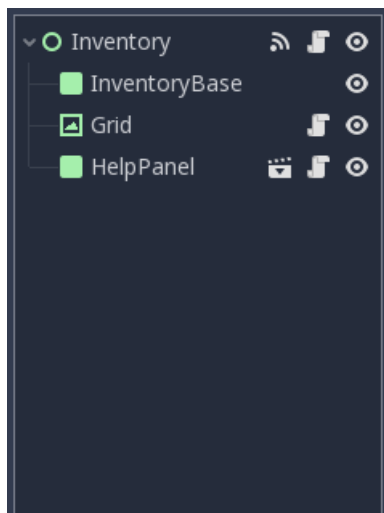


Figura 3.1: *Árvore da cena Inventory*

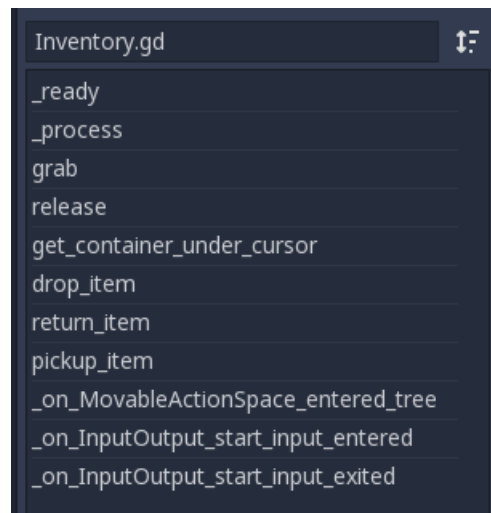


Figura 3.2: *Funções do script do nó Inventory*

```

9  #Variables that handles the Inventory Items and Containers
10 onready var grid = $Grid
11 var containers = [null]
12 var item_held = null
13 var item_offset = Vector2()
14 var last_container = null
15 var last_pos = Vector2()

```

Figura 3.3: *Variáveis do script do nó Inventory*

O nó *InventoryBase* apenas dá cor ao espaço destinado aos comandos no início do jogo e o nó *HelpPanel* é uma cena instanciada que gerencia o menu de ajuda que aparece ao solicitar mais explicações sobre um comando.

Dentro do jogo *Phoenix Rising* os comandos disponíveis devem ocupar o espaço de áreas específicas. Estas áreas são chamadas de recipientes (do inglês *container*). Um recipiente pode ser o próprio *Grid* do inventário ou espaços definidos dentro dos Espaços de Ação, que serão explicados posteriormente.

O nó *Grid* tem grande importância dentro da cena, pois é ele quem separa cada recipiente da tela de jogo em pequenos quadradinhos, facilitando a mecânica de mover os comandos disponíveis. Estes recipientes são adicionados na lista de *containers*, definida na linha 11 da figura 3.3.

Para facilitar o entendimento, veja as variáveis mais importantes e o método de inicialização da grade (do inglês *grid*):

```

3  #The list of every item in the grid
4  var items = []
5  #Used to create a metric for item positioning
6  var grid = {}
7  #Size of each grid cell
8  var cell_size = 32
9  #Size of the grid
10 var grid_width = 0
11 var grid_height = 0

```

Figura 3.4: Variáveis do Grid

```

16 #_ready initializes the grid
17 func _ready():
18     var s = get_grid_size(self)
19     grid_width = s.x
20     grid_height = s.y
21
22     for x in range (grid_width):
23         grid[x] = {}
24         for y in range (grid_height):
25             grid[x][y] = false

```

Figura 3.5: inicialização do Grid

O tamanho de cada quadrado é armazenado e definido na linha 8 da figura 3.4 e as medidas da grade são armazenadas pelas variáveis definidas nas linhas 10 e 11 da mesma figura.

Na inicialização da grade do inventário o método *get_grid_size* recebe o retângulo que foi definido para ser o recipiente e devolve seu tamanho (largura e altura) usando o sistema quadriculado, ou seja, devolve quantos quadrados de largura e altura ele ocupa. Depois, cada posição desta grade recebe o valor "falso", sinalizando que aquele quadrado está vazio, ou seja, que nenhum item do jogo ocupa aquelas posições.

A partir dessa divisão todo comando que é disponibilizado no inventário ocupará um número de quadrados do *grid* e a movimentação destes itens é feita pelos métodos de pegar (do inglês *grab*) e soltar (do inglês *release*) definidos no *scripts* do nó *Inventory*. Ao posicionar um comando em alguma região do recipiente os quadrados são marcados com "verdadeiro" para sinalizar que aquela região está preenchida.

```

27 #This function inserts the item on Action Space if it is possible
28 #returns true when it was possible to insert and false when it's not
29 func insert_item(item):
30     var item_pos = item.rect_global_position + Vector2(cell_size/2, cell_size/2)
31     var g_pos = pos_to_grid_coord(item_pos)
32     var item_size = get_grid_size(item)
33     if is_grid_space_available(g_pos.x, g_pos.y, item_size.x, item_size.y):
34         set_grid_space(g_pos.x, g_pos.y, item_size.x, item_size.y, true)
35         item.rect_global_position = rect_global_position + Vector2(g_pos.x, g_pos.y) * cell_size
36         items.append(item)
37         return true
38     else:
39         return false

```

Figura 3.6: Método de inserção de um item

```

61 #Receives a (x, y) grid position and sets to
62 #state every position on the rectangle (x, x+w, y, y+h)
63 ▾ func set_grid_space (x, y, w, h, state):
64 ▾     for i in range (x, x + w):
65 ▾         for j in range (y, y + h):
66             grid[i][j] = state
67

```

Figura 3.7: Marcação das posições do grid

3.2 Espaço de Ação

Dentro dos arquivos do jogo (diretório *Phoenix Rising*) é encontrado na pasta *ActionSpace*. O espaço de ação serve para conectar o sistema fazendo com que seja possível executá-lo e posicionar os comandos que serão utilizados no programa criado pelo jogador.

Basicamente o espaço de ação é dividido em duas partes, uma móvel e outra fixa. A cena *MovableActionSpace* é a parte que permite a mobilidade e a cena *ActionSpace* cuida de receber o item posicionado e os argumentos.

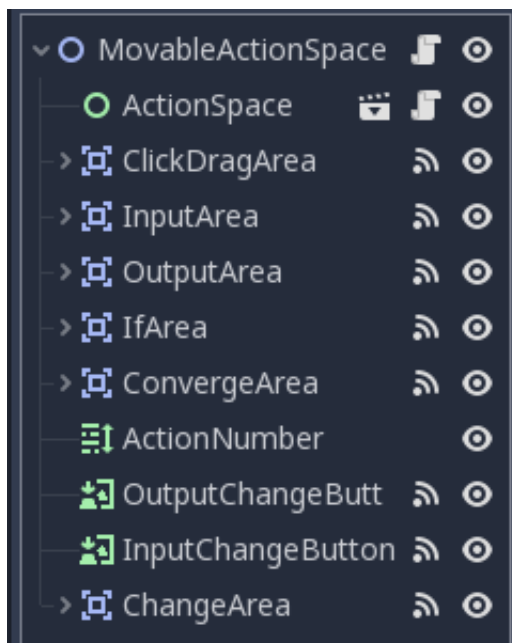


Figura 3.8: Cena MovableActionSpace

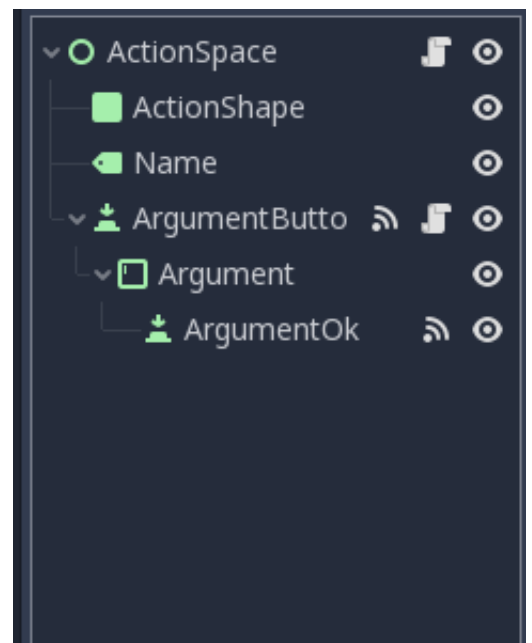


Figura 3.9: Cena ActionSpace

Note que, como *MovableActionSpace* é móvel, foi utilizado um nó do tipo *Node2D* e como *ActionSpace* é fixo, foi utilizado um nó do tipo *Control*.

A cena *MovableActionSpace* é formada pelos seguintes nós:

- **ActionSpace** - Cena que trata dos comandos e argumentos que são posicionados para a execução.
- **ClickDragArea** - Área com o ícone da mãozinha, destinada a mover o espaço de ação pela tela.
- **InputArea** - Conexões simples de *input*.
- **OutputArea** - Conexões simples de *output*.
- **IfArea** - Conexão complexa para administrar o comando *if/else*.
- **ConvergeArea** - Conexão complexa para unir os dois caminhos gerados a partir de um *if/else*.
- **ActionNumber** - Indica em qual posição na ordem de processamento que está aquela ação.
- **OutputChangeButton** - Área que permite a troca da conexão de saída (*output*)
- **InputChangeButton** - Área que permite a troca da conexão de entrada (*input*)
- **ChangeArea** - Área que marca em qual momento será feita a troca do valor *input* definido no processo visual.

O *script* atrelado ao nó *MovableActionSpace* torna possível os comportamentos descritos acima, os detalhes de como isso é feito não são relevantes neste momento, por isso não haverá explicação detalhada sobre o código, entretanto os curiosos que quiserem adicionar uma nova conexão simples de *input* devem seguir os passos:

- Seguindo o padrão de nomenclatura, criar um nó *sprite* chamado *NewConnection* e um nó de forma de colisão (*CollisionShape2D*) chamado *NewCollisionShape*, filhos de *InputArea*.
- Definir a forma de colisão do nó *NewCollisionShape*.
- Abrir o *script MovableActionSpace.gd* e adicionar na lista *input_connections* o nome do nó *sprite* que foi criado, seguindo o exemplo:

\$InputArea/NewConnection

- Ainda no *script MovableActionSpace.gd*, adicionar na lista *input_collisions* uma lista contendo o nome do nó *CollisionShape2D* que foi criado, seguindo o exemplo:

[*\$InputArea/NewCollisionShape*]

- Adicionar na lista *input_connected_textures* o caminho que está a imagem referente a nova conexão quando conectada. O *DEFAULT_PATH* está definido como "res://Accessories/art/", se a imagem estiver neste diretório basta adicionar:

DEFAULT_PATH + "minha_nova_conexao.png"

- Adicionar na lista *input_not_connected_textures* o caminho que está a imagem referente a nova conexão quando não conectada. O *DEFAULT_PATH* está definido como "res://Accessories/art/", se a imagem estiver neste diretório basta adicionar:

DEFAULT_PATH + "minha_nova_conexao_nao_conectada.png"

- Por último basta incrementar o número total de conexões da variável *num_input_connections* e pronto.

Esta sequência de ações pode ser utilizada para criar conexões de saída, basta preencher as respectivas listas que controlam as conexões de *output*.

O motivo das conexões serem tão importantes para este elemento do jogo vai além de só conectar o sistema e permitir sua execução, pois cada espaço de ação possui também duas variáveis chamadas *right_child* e *left_child* que fazem referência a qual outro espaço de ação está conectado a sua conexão de saída, criando a árvore de execução, utilizada pelo *RunEnvironment* e *VisualProcess*.

A árvore de execução permite obter a sequência de ações que o jogador projetou no nível, além disso esta estrutura facilita o comando condicional (*if/else*), o comando de loop e outros comandos que podem ser implementados como subrotinas.

Depois que o sistema está totalmente conectado, podemos criar um esquema da árvore de execução a partir de um sistema gerador da seguinte forma:

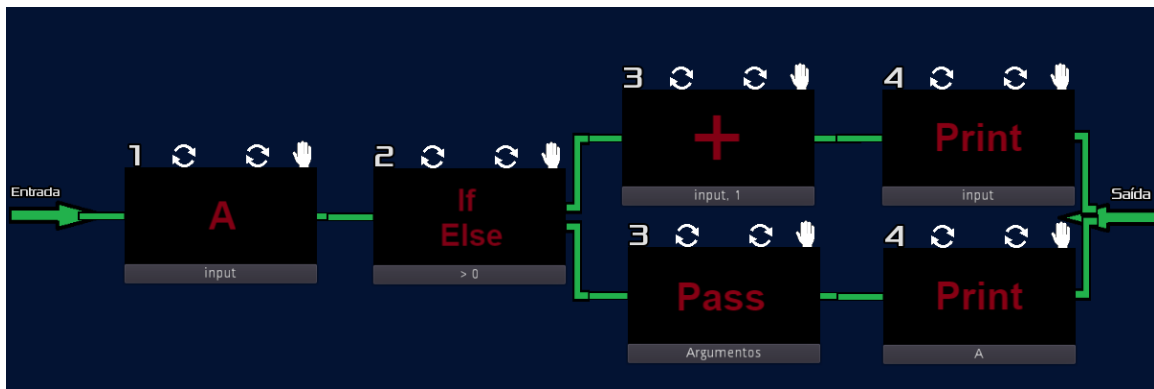


Figura 3.10: Sistema gerador da Árvore de Execução

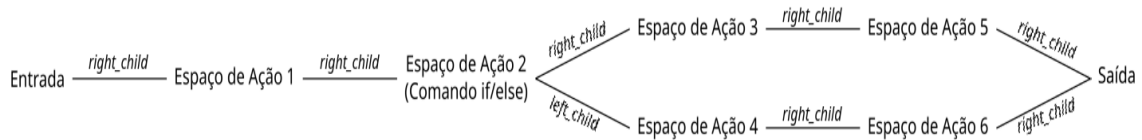


Figura 3.11: Esquema de Árvore de Execução

Vale ressaltar dois pontos importantes:

- Os filhos da esquerda, conhecidos como *left_child*, que não foram representados no esquema, para melhorar a apresentação e deixá-lo mais fácil de entender, existem e possuem o valor *null*.
- A numeração do sistema refere-se a ordem que os comandos serão executados, já a numeração do esquema enumera apenas a quantidade de espaços de ação que foram utilizados, portanto as numerações podem diferir.

Agora será dada a explicação sobre a cena *ActionSpace*, parte fixa do espaço de ação mostrada na figura 3.9, que é composta pelos seguintes nós:

- **ActionShape** - Delimita, com um retângulo preto, o espaço ocupado pelo espaço de ação.
- **Name** - Coloca o nome "Espaço de Ação" dentro do retângulo preto, pois tutorial faz referências a este item.
- **ArgumentButton** - Botão que permite ao jogador abrir a área de preenchimento dos argumentos que serão passados para um comando.
- **Argument** - Área de preenchimento dos argumentos, o que for escrito nesta área será passado como argumento para o comando posicionado.
- **ArgumentOk** - Quando pressionado fecha a área de preenchimento dos argumentos.

Novamente, vale lembrar que os detalhes de implementação não são relevantes, basta saber que esta cena é considerada um recipiente (*container*) para o inventário e caso um comando esteja posicionado nesta região não será possível movimentar o *MovableActionSpace* pela tela.

Uma consideração a ser feita a respeito dos argumentos serem preenchidos utilizando o teclado é que um menu de seleção com o mouse limitaria as opções do jogador, sendo assim ele poderia testar todas as opções disponíveis até que uma delas funcionasse, indo contra o objetivo do jogo que é ensinar e não apenas terminá-lo. Forçar o usuário a escrever o argumento com o formato pedido faz com que ele se acostume a ler o menu de ajuda ou as mensagens de erro e seguir os padrões que a computação exige, forçando-o a estar ciente do que está fazendo e concluindo o objetivo principal que é facilitar o aprendizado.

3.3 Processo Visual

Dentro dos arquivos do jogo (diretório *Phoenix Rising*) é encontrado na pasta *Visual-Process*. Este processo visual trata da animação que auxilia o jogador a entender o que está acontecendo com os valores do programa durante a execução.

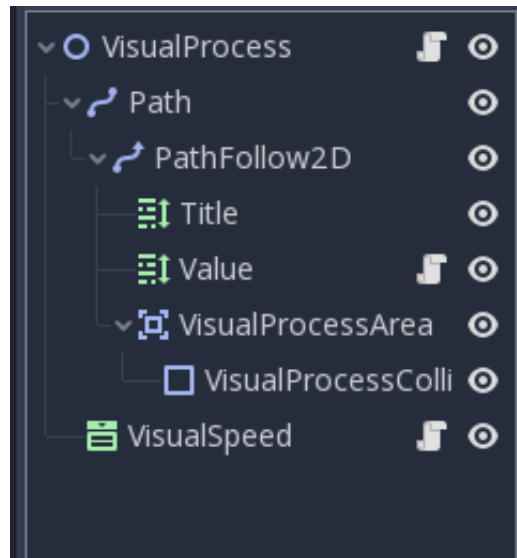


Figura 3.12: Cena Visual Process

- **Path** - Nó que contém a curva de pontos a ser seguida pela parte visual que aparece na tela.
- **PathFollow2D** - Trata dos pontos da curva a ser seguida.
- **Title** - Texto de identificação de qual variável a parte visual está mostrando na tela. (neste projeto apenas é mostrado o valor do *Input*)
- **Value** - Valor corrente da variável mostrada pelo processo visual que é identificada pelo *Title*.
- **VisualProcessArea** - Área definida para se efetuar a mudança dos valores em *Value*, administra alguns parâmetros da colisão.
- **VisualProcessCollision** - Área de colisão (utilizada pela *Godot*) definida para identificar o momento de se efetuar a mudança dos valores em *Value*.
- **VisualSpeed** - Menu que controla a velocidade que a animação do processo visual será mostrada.

Para entender melhor o processo visual deve-se compreender o básico sobre o funcionamento dos nós *Path*¹ e *PathFollow2D*² na *Godot*. O nó *Path* espera receber uma curva de pontos, já *PathFollow2D* pega seu *Path* pai e devolve as coordenadas de um ponto dentro dele, dada a distância do primeiro vértice, sendo útil para fazer outros nós seguirem um caminho, sem codificar o padrão de movimento. Para isso, os nós devem ser descendentes desse nó. Em seguida, ao definir um deslocamento neste nó, os nós descendentes se moverão de acordo.

Seguindo o funcionamento de *Path* e *PathFollow2D* ao instanciar *Title*, *Value* e *VisualProcessArea* como filhos de *PathFollow2D* todos eles irão seguir o caminho da curva

¹https://docs.godotengine.org/en/3.1/classes/class_path.html

²https://docs.godotengine.org/en/3.1/classes/class_pathfollow.html#class-pathfollow

definida, o que permite movimentar o valor do *Input* pela tela de jogo. Veja abaixo algumas variáveis que são utilizadas no controle deste processo:

```

6  var starting_pos = null
7  var finish_pos = null
8  var start_input = ''
9  var curve = Curve2D.new( )
10 var total_inputs = 0
11 var current_start_input = 0
12 var processed_input = null
13 var CurrentActionNode = null
14
15 var is_exit_sucess = false

```

Figura 3.13: Variáveis Visual Process

- ***starting_pos*** e ***finish_pos*** - demarcam onde o processo visual inicia e onde ele termina, respectivamente.
- ***start_input*** - recebe o valor inicial da entrada.
- ***curve*** - guarda a curva de pontos, que será preenchida conforme a execução do programa.
- ***total_inputs*** - guarda a quantidade de valores que foi fornecido na entrada do programa
- ***current_start_input*** - guarda qual valor da entrada que será processado.
- ***processed_input*** - armazena os valores que se alteram durante a execução e que são exibidos na animação.
- ***CurrentActionNode*** - guarda qual espaço de ação que será executado
- ***is_exit_sucess*** - sinaliza se houve algum erro durante a execução.

A função que possibilita o processo visual executar as modificações no *input* e seguir um caminho na tela é a *_on_MovableActionSpace_change_area_entered* definida no script do nó *VisualProcess*. Esta função é chamada sempre que há colisão entre *VisualProcessCollision* e *ChangeArea*, que está definido no espaço de ação.

Veja abaixo o código:

```

63 ▾ func _on_MovableActionSpace_change_area_entered():
64 ▾     processed_input = (CurrentActionNode.function).call_func(processed_input[0],
65                               CurrentActionNode.arguments, CurrentActionNode.action_number)
66 ▾     if (processed_input[0] == null):
67         is_exit_sucess = false
68         _clear_all_process()
69         emit_signal("end_path")
70 ▾     else:
71 ▾         if (processed_input[1] == true):
72             CurrentActionNode = CurrentActionNode.right_child
73 ▾         else:
74             CurrentActionNode = CurrentActionNode.left_child
75 ▾         if (CurrentActionNode == null):
76             _add_next_point(finish_pos)
77 ▾         else:
78             _add_next_point((CurrentActionNode.node).global_position)
79     ValueNode.text = str(processed_input[0])
80

```

Figura 3.14: Variáveis Visual Process

Note que a função `_on_MovableActionSpace_change_area_entered` é chamada sempre que é detectada uma colisão entre um espaço de ação e o *input*, portanto a variável *CurrentActionNode* está armazenando uma referência para o espaço de ação que colidiu.

Iniciando a função, na linha 64, é executado o código do comando que está posicionado dentro do espaço de ação e o novo *input* que foi processado bem como qual o caminho a seguir, serão armazenados na lista *processed_input*.

A lista *processed_input* guarda, na primeira posição (0), o valor resultante da operação do comando posicionado e terá *null* caso algo tenha dado errado, por exemplo um erro na passagem do argumento. O *if*, na linha 66, verifica se algo deu errado na execução do comando. Caso nada de errado tenha acontecido é executado o *else*, na linha 70, que identificará qual será o caminho que o processo visual deverá tomar.

A lista *processed_input* guarda, na segunda posição (1), qual o caminho que deverá ser seguido, caso a posição esteja marcada com *true*, então será seguido o caminho do filho da direita, se o valor for *false*, então será seguido o caminho do filho da esquerda. Para entender esta parte talvez seja necessário relembrar o que significam estes "caminhos" observando novamente a figura 3.11: Esquema de Árvore de Execução, definida na seção "Espaço de Ação".

Depois há a verificação se o processo visual chegou ao fim, na linha 75, ou se há mais algum ponto a seguir, na linha 77-78. Por fim, na linha 79, atualiza-se o texto que aparece na tela para o jogador visualizar.

Note que a execução dos comandos acontece enquanto o processo visual é mostrado na tela, assim o jogador pode acompanhar o andamento do seu programa até que algo

de arrado aconteça, facilitando o entendimento de cada comando individualmente e a correção dos erros que aparecerem.

3.4 Ambiente de Execução

Dentro dos arquivos do jogo (diretório *Phoenix Rising*) é encontrado na pasta *RunEnvironment*. Este ambiente trata de construir o arcabouço para executar o sistema.

As imagens abaixo ilustram os nós que compõem a cena principal, chamada *RunEnvironment.tscn* e as funções que estão definidas no script *RunEnvironment.gd* atrelado ao nó principal.

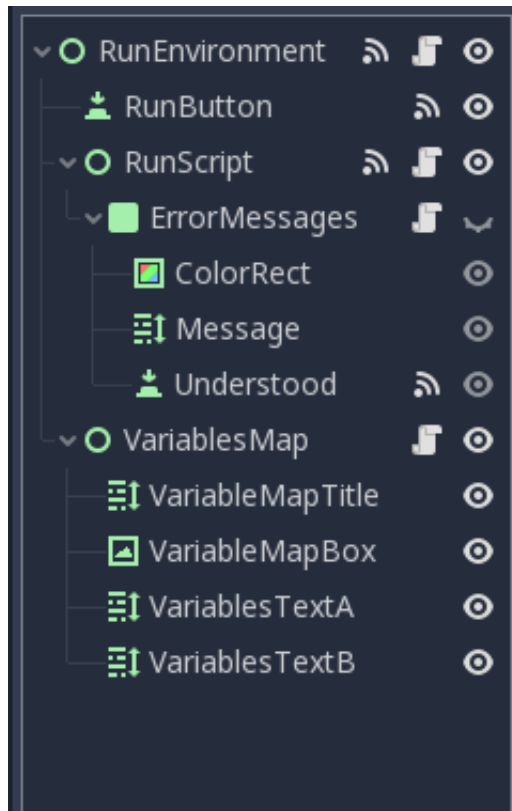


Figura 3.15: Árvore da cena *RunEnvironment*

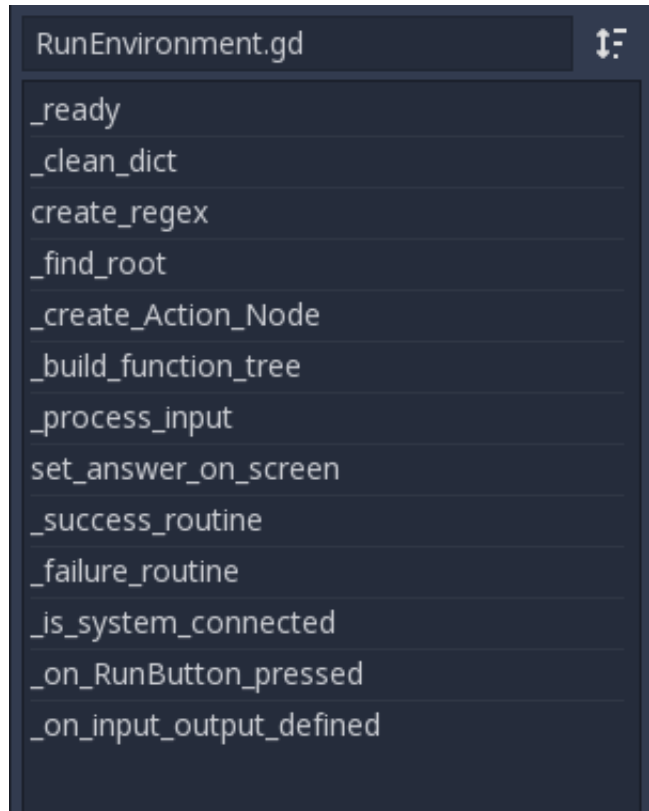


Figura 3.16: Funções de *RunEnvironment.gd*

- **RunButton** - Botão que, ao ser pressionado, faz com que inicie uma tentativa de executar o sistema.
- **RunScript** - Carrega o script *RunScript.gd*, neste *script* estão as funções de comportamento dos comandos.
- **ErrorMessage** - Espaço definido para a mensagem de erro.
- **ColorRect** - Dá cor ao espaço destinado as mensagens de erro.
- **Message** - Texto da mensagem de erro.

- ***Understood*** - Botão que permite fechar a mensagem de erro e dar sequência ao jogo.
- ***VariablesMap*** - Carrega o script *VariablesMap.gd*, neste *script* estão as funções que dão comportamento ao mapa de variáveis.
- ***VariablesMapTitle*** - Título do mapa de variáveis.
- ***VariablesMapBox*** - Textura destinada ao mapa de variáveis.
- ***VariablesMapTextA*** - Texto da valoração da variável A.
- ***VariablesMapTextB*** - Texto da valoração da variável B.

Depois de definir algumas variáveis como o texto do mapa de variáveis e qual a saída esperada esta cena estará pronta para exercer seu papel mais importante: **Iniciar a tentativa de execução do sistema.**

Inicialmente será apenas uma tentativa, pois o sistema pode não estar devidamente conectado, impedindo o início da execução ou algum comando pode resultar em erro, interrompendo a execução. A ocorrência de ambas as opções será explicada a seguir.

Note a função que é chamada assim que o botão **Rodar!**, definido pelo nó *RunButton*, é pressionado.

```

108 ▾ func _on_RunButton_pressed():
109     emit_signal("attempt_to_run")
110     var PlayerOutput = get_parent().get_node("InputOutput/OutputBase/PlayerOutput")
111     PlayerOutput.clear()
112     _clean_dict()
113 ▾ if (_is_system_connected(_find_root())):
114     var was_sucessfull = yield(_process_input(input_list), "completed")
115 ▾     if (was_sucessfull):
116         var answer_string = PlayerOutput.text
117 ▾         if (answer_string == output):
118             _success_routine()
119 ▾         else:
120             _failure_routine()
121 ▾     else:
122         _failure_routine()
123

```

Figura 3.17: Função do botão **Rodar!**

Daqui pra baixo apenas posicionei as figuras que serão utilizadas.

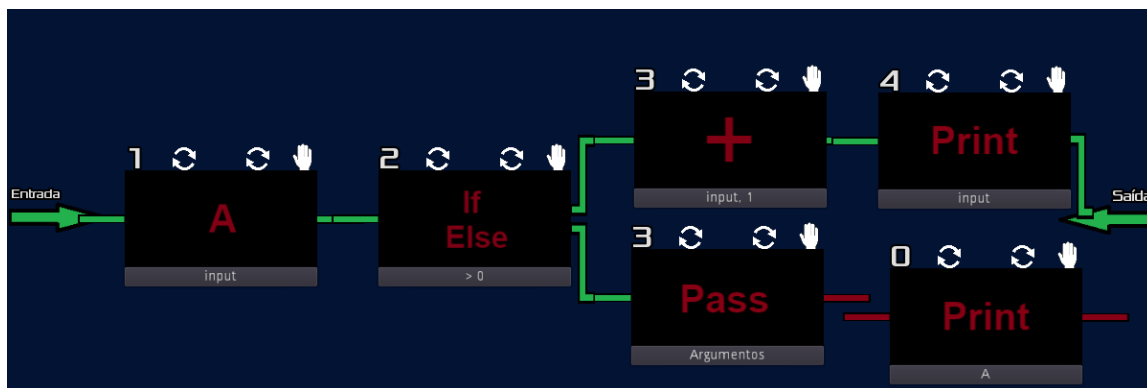


Figura 3.18: Sistema gerador da Árvore de Execução não Conectada

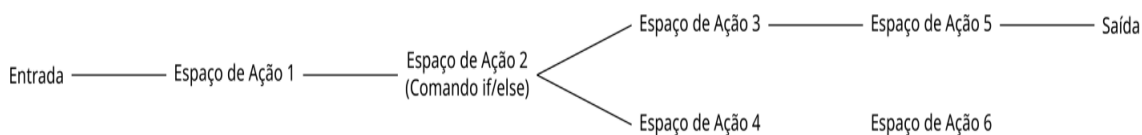


Figura 3.19: Esquema de Árvore de Execução não Conectada

3.5 Nível Base

Dentro dos arquivos do jogo (diretório *Phoenix Rising*) é encontrado na pasta *BaseLevel*

3.6 Cenas Genéricas

Dentro dos arquivos do jogo (diretório *Phoenix Rising*) é encontrado na pasta *GenericGameScenes*

3.7 Comportamento dos Comandos

Dentro dos arquivos do jogo (diretório *Phoenix Rising*) é encontrado na pasta *RunEnvironment*

3.8 Ajuda ao Usuário

Dentro dos arquivos do jogo (diretório *Phoenix Rising*) é encontrado na pastas *Users-Guide* e em *GenericGameScenes*

Capítulo 4

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/instancing.html

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/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