**CARLOS EDUARDO TAPUDIMA DE OLIVEIRA**

# PITFALL! UTILIZANDO UNITY ENGINE

Trabalho de Conclusão de Curso apresentado à banca avaliadora do Curso de Sistemas de Informação, da Escola Superior de Tecnologia, da Universidade do Estado do Amazonas, como pré-requisito para obtenção do título de Bacharel em Sistemas de Informação.

Orientador(a): Prof. Dr. Jucimar Maia Junior

Manaus – fev – 2024

***Universidade do Estado do Amazonas - UEA***
***Escola Superior de Tecnologia - EST***

*Reitor:*
***André Luiz Nunes Zogahib***
*Vice-Reitora:*
***Kátia do Nascimento Couceiro***
*Diretor da Escola Superior de Tecnologia:*
***Jucimar Maia da Silva Junior***
*Coordenadora do Curso de Sistemas de Informação:*
***Marcela Sávia Picanço Pessoa***
*Coordenadora da Disciplina Projeto Final:*
***Polianny Almeida Lima***

*Banca Avaliadora composta por:* *Data da Defesa: 20/02/2024.*
***Prof. Dr. Jucimar Maia. da Silva Junior*** *(Orientador)*
***Prof. Dr. Clairon Lima Pinheiro***
***Prof. MSc. Eduardo Jorge Lira Antunes da Silva***


## Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).
**Sistema Integrado de Bibliotecas da Universidade do Estado do Amazonas.**

C284pp Oliveira, Carlos Eduardo Tapudima de
Pitfall! Utilizando Unity Engine / Carlos Eduardo Tapudima de Oliveira. Manaus : [s.n], 2024.
65 f.: color.; 30 cm.

TCC - Graduação em Sistemas de Informação - Bacharelado - Universidade do Estado do Amazonas, Manaus, 2024.
Inclui bibliografia
Orientador: Jucimar Maia da Silva Junior

1. Pitfall!. 2. Jogos Digitais. 3. Unity Engine. I. Jucimar Maia da Silva Junior (Orient.). II. Universidade do Estado do Amazonas. III. Pitfall! Utilizando Unity Engine

**Universidade do Estado do Amazonas - UEA**
**Escola Superior de Tecnologia - EST**
**Bacharelado em Sistemas de Informação - SI**

# FOLHA DE APROVAÇÃO

## Pitfall! Utilizando Unity Engine

## Carlos Eduardo Tapudima de Oliveira

Trabalho de Conclusão de Curso II defendido e aprovado pela banca avaliadora constituída pelos professores:

Prof. Dr. Jucimar Maia da Silva Júnior
Presidente

Prof. Dr. Clairon Lima Pinheiro
Avaliador

Prof. Msc. Eduardo Jorge Lira Antunes da Silva
Avaliador

Manaus, 20 de fevereiro de 2024.

# Agradecimentos

Gostaria de agradecer ao meu orientador Dr. Jucimar Junior que me orientou neste projeto e em várias outras etapas da minha vida acadêmica. Agradeço também a toda a todo o corpo docente da Universidade do Estado do Amazonas.

Agradeço a professora Marcela Pessoa que não só me incentivou a continuar quando eu tinha dúvidas como também me mostrou o poder da comunidade acadêmica.

Gostaria de agradecer a minha família e amigos que estiveram comigo durante minha jornada acadêmica em especial ao meu marido Gabriel, se não fosse você eu não teria chegado até aqui.

Por fim agradeço a instituição da Universidade do Estado do Amazonas que proporcionou um ambiente propicio e cheio de oportunidades para que eu pudesse me desenvolver pessoal e profissionalmente.

# Resumo

A área de jogos digitais cresceu muito no Brasil (FORTIM et al., 2022), aumentando assim também a quantidade de pessoas interessadas em adentrar a industria através do desenvolvimento. O fácil acesso a ferramentas poderosas de desenvolvimento como a Unity contribuem para esse fator porém aprender os básicos da produção de jogos as vezes torna-se um desafio haja-vista a preocupação portabilidade e performance impactam diretamente na popularidade de novos títulos. Este projeto mostra como o estudo de jogos clássicos, nesse caso o Pitfall! originalmente para Atari 2600 torna-se uma ótima estratégia para o aprendizado de boas práticas no desenvolvimento de jogos.

**Palavras-Chave**: Pitfall!; Jogos Digitais; Unity Engine.

# Abstract

The Gaming Industry grew larger in Brazil (FORTIM et al., 2022), with it more and more people became interested in entering this area thru developing their own titles. The easy access to powerful development tools like Unity also contribute to this fact but to learn the basics of game production becomes a challange due to complex problems like the portability and performance that impact direclty on new titles success. This project shows how the study of classic games, in this case Pitfall! originally for Atari 2600 becomes a great strategy for learning good game development practices.

**Keywords**: Pitfall!; Digital Games; Unity Engine.

# Sumário

# Lista de Tabelas

# Lista de Figuras

# Lista de Códigos

# Capítulo 1

# Introdução

"Três em cada quatro pessoas no Brasil utilizam celular, videogame ou computador para jogar" (TADEU; TORTELLA, 2022) é o que afirma a pesquisa realizada pela Pesquisa Brasil Games no ano de 2022. Com base nesse estudo pode-se assumir que: Jogos eletrônicos estão cada vez mais acessíveis ao público geral através de diversas mídias. Para garantir tal acessibilidade faz-se necessária a otimização no uso de dados afim de atingir o maior numero possível de jogadores. Métricas como o uso de memória RAM, armazenamento e quantidade de conteúdo jogáveis são essenciais no desenvolvimento de jogos.

*Pitfall!*, criado por David Crane em 1982, faz uso da otimização de espaço com maestria por conseguir contar com trilha sonora, 225 fases diferentes, 32 coletáveis e diversos obstáculos e inimigos tendo apenas 4KB (quilo-bytes) para armazenamento do software completo em um cartucho padrão de Atari 2600.

Utilizando-se da estratégia de otimização desenvolvida por Crane, que consiste em utilizar-se de um vetor pré-definido de números inteiros e, através de abstrações matemáticas, montar a instância do atual estágio baseado no posicionamento do índex, valor que representa a atual posição do vetor, utilizando essas informações para definir a existência e posicionamento de cada elemento que compõe o level design.

## 1.1 Justificativa

Uma das maiores dificuldades na produção de software dado o panorama de mercado atual é a otimização, é ter a certeza de que o software consegue alcançar diversos usuários com diferentes equipamentos e capacidade computacional múltipla. Esse também é o objetivo da área desenvolvimento de jogos de entretenimento eletrônico.

Tendo em vista que para atingir um grande número de jogadores atualmente o Game precisa ser capaz de performar de maneira satisfatória no maior número de dispositivos possível, a a otimização tem sido a forma mais comum de atingir estes objetivos através da garantia de que o software seja capaz de ser executado em dispositivos mais simples tecnologicamente.

Nesse contexto, esta monografia apresentará uma avaliação do real impacto em performance de software não-otimizado em comparação a um software otimizado utilizando uma abordagem criada por David Crane para o jogo *Pitfall!* originalmente para o Atari, comprovando assim a relevância crucial da otimização em jogos digitais.

## 1.2 Objetivos

O objetivo deste trabalho é realizar uma implementação baseada no método original implementado por David Crane no jogo *Pitfall!* utilizando a engine *Unity3D*. Dessa forma busca-se mostrar os ganhos em redescobrir métodos de otimização através da refatoração de técnicas oriundas dos jogos clássicos e suas limitações técnicas para solucionar problemas atuais de performance utilizando geração procedural.

### 1.2.1 Objetivos Específicos

O objetivo específico deste trabalho se concentra no desenvolvimento do jogo *Pitfall!* da empresa *Activision* seguindo a seguinte estrutura:

- Desenvolver as principais mecânicas do jogo *Pitfall!* em sua versão para Atari 2800
- Utilizar geração procedural para gerar as 255 fases;

- Utilizar apenas uma *"Scene"* da Unity para todas as fases do jogo;
- A análise a escalabilidade do algoritmo do jogo dependendo do numero de fases;

## 1.3 Metodologia de Pesquisa

A metodologia de pesquisa utilizada para o desenvolvimento e implementação do presente projeto foi a Pesquisa Exploratória que, de acordo com Toledo (TOLEDO; SHIAISHI, 2016) apresenta o objetivo de "coletar e documentar dados sobre um fenômeno específico". Afim de atingir este propósito foi feita uma pesquisa qualitativa extensiva por documentários, artigos e similares utilizando-se de palavras-chave como "atari", "pitfall", "arcade" e "game design". Foram encontrados o código-fonte original do jogo de Atari que utiliza a linguagem Assembly bem como entrevistas com o criador do jogo David Crane que explica e exemplifica a lógica por trás das linhas de comando produzidas.

# Capítulo 2

# Fundamentação Teórica

Este trabalho aborda principalmente o desenvolvimento uma versão otimizada do jogo *Pitfall!* desenvolvido inicialmente para o Atari 2600 utilizando a Unity Engine como ferramenta de programação.

Otimização segundo o dicionário Oxford significa "mudar dados, software, etc. com o intuito de fazer com o que o software funcione de maneira mais eficiente ou seguir um propósito específico". Para este projeto, tem-se como propósito otimizar o jogo eletrônico afim de fazer com o mesmo utilize menos recursos computacionais.

Recursos computacionais, recursos de sistema ou apenas recursos, podem ser definidos como qualquer componente físico ou virtual de quantidade finita dentro de um sistema computacional(CIPOLLA-FICARRA, 2021). Para fins deste projeto, serão considerados recursos: Capacidade de memória RAM e capacidade de memória de armazenamento.

Um jogo eletrônico é um programa de software voltado inicialmente para entretenimento, porém não está limitado apenas a este fim, outras funcionalidades como treinamentos, aprendizagem ou mesmo a preservação de costumes e culturas. Todos os jogos entretanto possuem regras a serem seguidas, condições para vitória ou derrota e objetivos. Essas características atuam em conjunto para formar a experiência de jogo (ERMI; MÄYRÄ, 2005).

Jogos de plataforma são um subgênero dos jogos muito popular e presente desde os primórdios dos jogos eletrônicos. Sua principal característica é ter seu mapa composto por plataformas e sua principal mecânica é o salto, onde o jogador deve atravessar as fases utilizando-se de tais

plataformas para tal (MINKKINEN, 2016). Outra característica muito comum do gênero é a existência de coletáveis que são itens dispostos ao longo das plataformas com diversos propósito, sejam pontos, vidas extras ou aumentos de poder (*Power-Ups*).

Geração procedural pode ser definida como a criação de dados através do uso de algoritmos, sendo esses dados sintéticos uma ótima alternativa para a arquitetura de fases que é o caso deste trabalho (SHORT; ADAMS, 2017). Através desta prática pode-se criar enorme variedade de níveis seguindo regras pré-determinadas pelo algoritmo. Todos os estágios são criados utilizando esta técnica com exceção do primeiro.

Mecânicas são definidas por Miguel Sicart (SICART, 2008) como ”métodos invocados por agentes, desenhados para interagir com o estado do jogo”em tradução livre. Expandindo um pouco essa definição e associando-a ao paradigma de programação orientada a objetos, pode-se chegar a conclusão que mecânicas são todos os métodos acionados a partir ou não de entradas do usuário que alteram a condição atual do jogo. T

## 2.1 Entendendo o algoritmo original de *Pitfall!*

O Algoritmo original produzido por David Crane utilizava de abstrações matemáticas para fazer a criação das fases de forma pseudo-procedural. Ao invés de colocar em forma de código todas os 255 níveis de e copiar todo os dados referentes a fase como qual fundo utilizar, quantos e quais inimigos existem em determinado estágio, se há ou não acesso a um túnel e a que nível ele está conectado, o algoritmo original consegue diluir todas estas informações utilizando-se de uma abstração matemática e operações com binários para representar os elementos do jogo.

| Vetor Ordenado | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | ... | 255 |
| Vetor de *Pitfall!* | | | | | |
| 12 | 45 | 23 | 142 | ... | 255 |

Figura 2.1: Exemplo de disposição do vetor de *Pitfall!*

A base da implementação é um contador que, ao invés de contar de maneira ordenada,

possui números aleatórios de 1 até 255 dispostos de forma aleatória. (figura a seguir) A partir daí, pode-se separar este número de 8 dígitos pode ser quebrado em 3 conjuntos de dois, três e três dígitos, cada um representando um elemento do estágio atual.

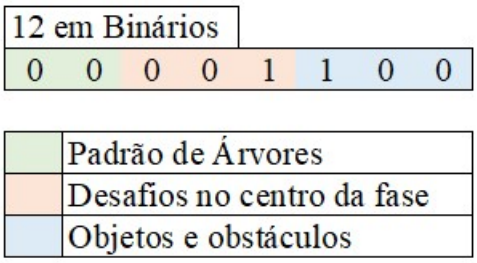


Figura 2.2: Exemplo da utilização dos números do vetor em *Pitfall!*.

O subconjunto de dois dígitos pode assumir até quatro estados. Ao se deparar com este subconjunto um dos fundos pré-estabelecido é carregado para a fase atual. Um dos subconjuntos de três dígitos representa qual o obstáculo que se encontrará no meio do nível podendo ser um lago, um lago com crocodilos, areia movediça, um buraco que abre e fecha ou uma escada para o nível subsolo. O terceiro e último subconjunto representa objetos ou obstáculos como por exemplo os tesouros, os inimigos: cobra e escorpião e os obstáculos como troncos. Apenas utilizando-se desta estratégia consegue-se criar 255 estágios distintos utilizando-se do mínimo possível de memória de armazenamento.

### 2.1.1 Mecânicas

As mecânicas presentes no jogo original são:

- Pulo;
- Movimentação lateral para a direita ou esquerda;
- O cipó (ou corda);
- Obstáculos (areia movediça, troncos, lagos, buracos, etc.);
- Túneis subterrâneos;

- Inimigos;
- Coletáveis (tesouros);
- Temporizador (decrescente a partir de 20 min).

Todos os itens acima precisam ser levados em consideração pelo gerador de fases procedural. Cada um tornando-se variáveis de cada instância de fase do jogo. A existência de cada um destes elementos será decidido através de uma variável pseudo-randômica utilizando a classe *Random* presente na própria *Unity Engine*.

Além dos itens da lista, em sua implementação original, outro detalhe que também era alterado no decorrer dos níveis era o fundo, composto por árvores que se dispunham de formas diferentes conforme a abstração matemática definida por Crane ditava. Algo semelhante também deve ser implementado na versão produzida para este trabalho.

### 2.1.2 Gerador de Fases

Elementos estruturais da fase como obstáculos, cipós e escadas para os túneis sempre estão presentes no na parte central do nível (em laranja) e é essa área que sofrerá o maior número de alterações pelo gerador. Dito isso, faz-se necessário o uso de âncoras para definir exatamente para o algoritmo a área a ser alterada durante a criação do nível novo.

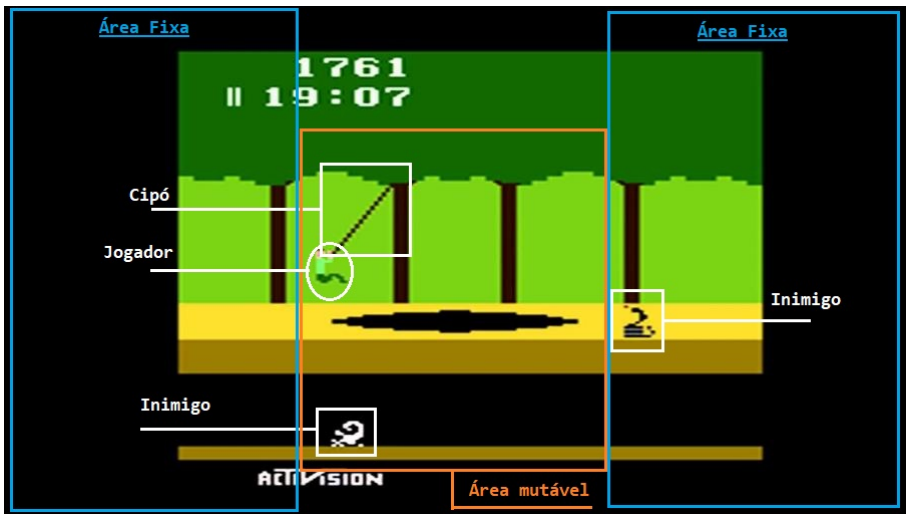


Figura 2.3: Esquema visual exemplificando as áreas e elementos na tela.

Alguns elementos não seguem esse padrão como a cobra por exemplo, então também faz-se necessário o uso de âncoras para tais elementos. Os coletáveis também possuem local fixo na tela, logo também precisarão de âncoras. Elementos de UI como a pontuação e o temporizador no canto superior da direito da tela serão perenes juntamente com os elementos presentes na área azul que fazem parte da esturura básica da disposição da fase na tela.

### 2.1.3 O Jogador: *Pitfall Harry*

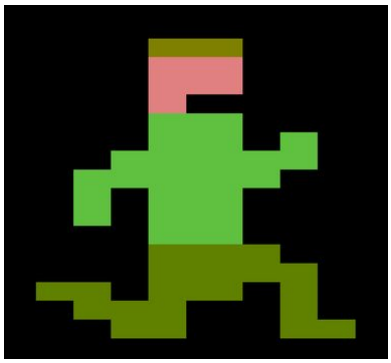

Figura 2.4: Imagem do personagem Pitfall Harry original Fonte: Pitfall! Activision 1926

O personagem que representa o jogador, Pitfall Harry precisa ter configurado as habilidades de andar, tanto para a direita quanto para a esquerda, a velocidade dele vai ser sempre fixa, ou seja, não haverá o modo de ”corrida”porém a animação precisa transparecer a ideia de velocidade. A interação com elementos da tela como o cipó não podem alterar a velocidade de Harry.

O pulo e o alcance do pulo também possuem altura e velocidade constantes. Caso o jogador sofra dano e perca uma vida ele deve ser recolocada no início da fase corrente, seja no solo ou no subsolo. Ao perder todas as vidas o jogador perde o jogo e deve ser levado para a tela de ”Fim de Jogo”tendo a opção de recomeçar do zero.

### 2.1.4 Coletáveis

O objetivo principal do jogo e também o seu critério de vitória é a obtenção de todos os tesouros espalhados pelos diversos níveis de *Pitfall!*. Os tesouros podem estar no subterrâneo ou no solo.

Ao total serão 20 tesouros assim como no original e os que se encontrarem no subsolo não podem ser inacessíveis, ou seja, sempre precisa haver uma entrada para aquele tesouro.

### 2.1.5 O Subsolo

O Subsolo é uma parte integral das mecânicas de *Pitfall!* ele consiste em uma serie de túneis interligados e tem seu acesso através de uma escada presenta na o meio do nível corrente. Cada túnel tem apenas dois acessos, uma entrada e uma saída, caso o jogador esteja no subsolo nenhuma parede pode impedir a passagem dele, entretanto uma escada está sempre acompanhada de uma parede a sua direita ou a sua esquerda, indicando assim a direção em que o jogador deve seguir.


Figura 2.5: Imagem de fase com entrada para o subsolo

### 2.1.6 Os inimigos

Os inimigos aparecem de diversas formas em *Pitfall!* mas para este trabalho serão apenas dois, a cobra, o escorpião. O escorpião sempre se move apenas para um um lado, da direita para a

esquerda e sempre aparece no subsolo, já a cobra aparece apenas no solo, não se move. Ambos devem ser evitados pelo jogador, caso ele encoste em algum deles ele sofre dano e perde uma vida.

### 2.1.7 Obstáculos

Obstáculos podem ser vários mas para este projeto usaremos a areia movediça, o lago de crocodilos, três buracos intercalados e os troncos rolantes. Cair ou entrar em contato com qualquer um destes ocasiona na perda de uma vida para o jogador. A mecânica específica dos crocodilos foi descrita na seção anterior a esta.

Para fazer a travessia por estes obstáculos, Harry pode pular por cima deles ou pode fazer o uso do cipó. O cipó sempre estará disponível quando os obstáculos forem a areia movediça ou o lago de crocodilos e pode também estar disponível no caso dos troncos. Ao encostar no cipó, Harry é levado junto com ele na direção que ele está indo, o peso ou o contato do jogador com o cipó não altera a física do mesmo, e para que Harry quebre o a mecânica de contato com o cipó basta ele pressionar o botão de pulo.

### 2.1.8 Tempo

O tempo é umas das condições de perda do jogo, o temporizador começa assim que o primeiro nível começa, e ele continua contando independente das ações do jogador. Não há como aumentar de forma alguma e a diminuição dele é de acordo com o tempo normal. Perder uma vida, mudar de fase, entrar em no subsolo ou interagir com qualquer mecânica presente no nível não altera o temporizador.

### 2.1.9 Pontuação

A pontuação se dá ao coletar cada um dos tesouros, ela é incremental apenas, não há forma de perder pontos, perder uma vida, atravessar um obstáculo ou transpassar um nível não afetam a pontuação, tendo em vista esse comportamento para fins deste projeto a pontuação tradicional

foi trocada por um contador de tesouros, sendo mais fácil e claro para o jogador identificar a sua progressão.

### 2.1.10 O Mapa Original

A seguir tem-se a imagem completa com todas as fases criadas por Crane com suas respectivas características utilizando o algoritmo explicado anteriormente. Aqui pode-se ter a noção do escopo do jogo original e quanta informação pode ser abstraída utilizando-se.

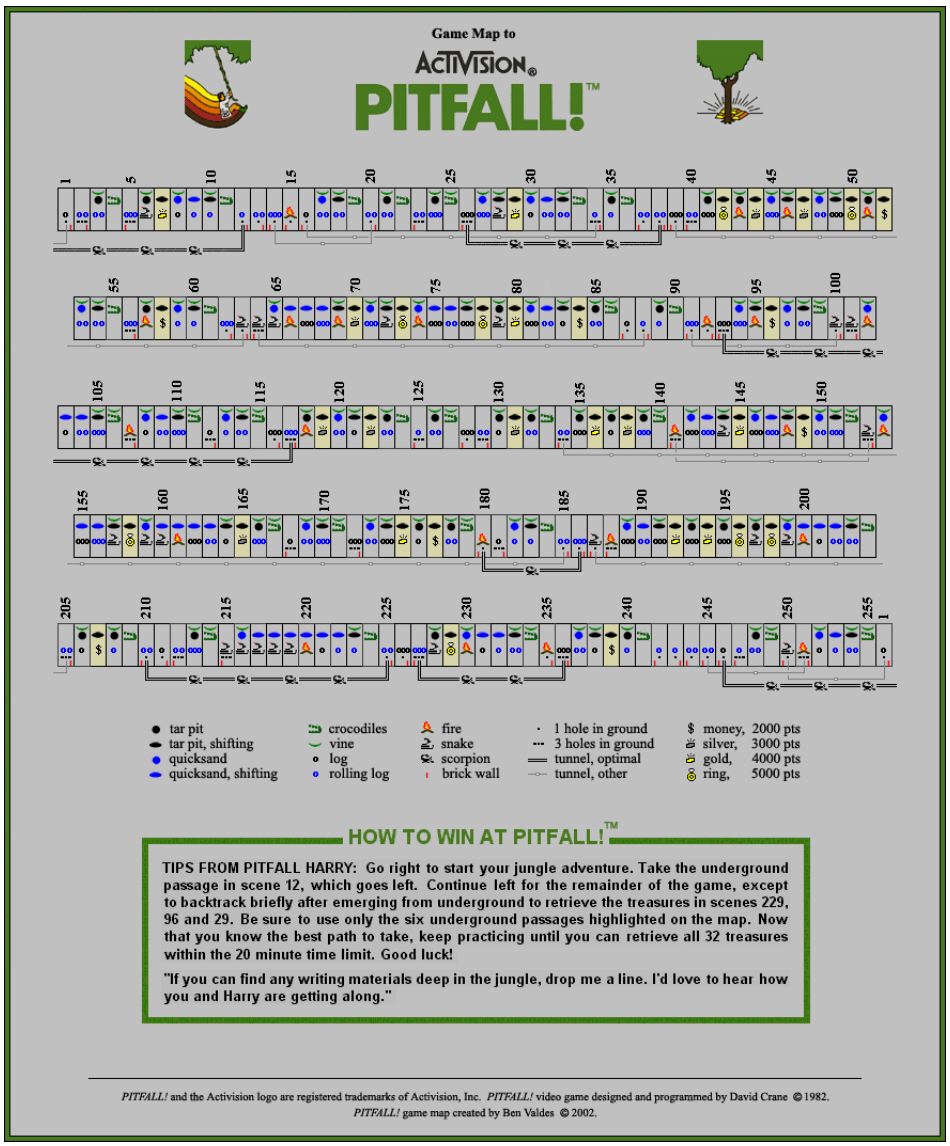


Figura 2.6: Mapa Completo de *Pitfall!* produzido pela *Actvision*

# Capítulo 3

# Metodologia de Desenvolvimento

Neste capítulo será detalhado todo o processo de desenvolvimento utilizado na implementação do presente trabalho sendo o primeiro paço entender a lógica utilizada por David Crane na implementação original.

## 3.1 Desenvolvimento em Cascata

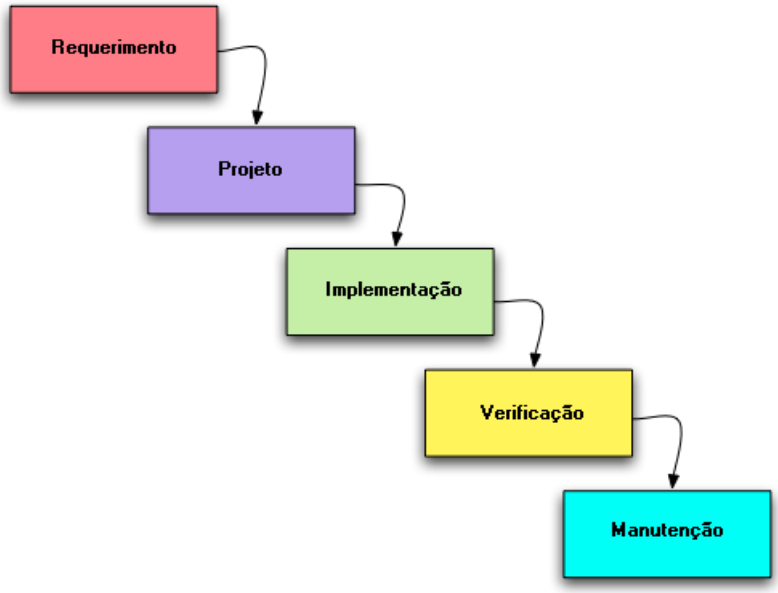


Figura 3.1: Modelo de desenvolvimento em Cascata

O modelo de desenvolvimento de software escolhido foi o modelo em cascata. Segundo o

autor (SENARATH, 2021) este modelo é caracterizado por sua estrutura linear e sequencial, apresentando estágios bem definidos de desenvolvimento que não se sobrepõe em nenhum momento. Uma característica essencial deste modelo de desenvolvimento é o seu planejamento. A fase de planejamento é capaz de elucidar muitos pontos que podem estar ainda fora de compreensão. A fase de testes garante a qualidade do software resultante desta modelagem. São extensos testes de performance, de funcionalidade e bem documentados através dos requisitos levantados na fase de "Requerimento".

## 3.2 A análise dos resultados

Após o desenvolvimento do projeto, uma análise sobre o consumo de recursos será feita, um relatório de performance analisando a consistência das fases geradas levando em consideração a quantidade de níveis similares, o tempo (em segundos) necessário para a geração de todas as 255 fases ao longo de 500 ciclos de geração, o consumo de memória RAM e memória de armazenamento.

Já a qualidade das fases será analisada através dos seguintes critérios: todos os critérios das mecânicas mencionadas neste artigo devem ser seguidos, os estágios não se repetirem de forma alguma, todos os obstáculos e/ou inimigos devem possuir uma solução para serem ultrapassados.

## 3.3 Modelagem do Sistema

Com a produção do diagrama de caso de uso com base no que foi aprendido sobre o desenvolvimento original do jogo e suas mecânicas mais elementares foram encontrados dois atores internos do jogo, um deles o "Gerador de Fases" que como o nome diz, está encarregado de fazer toda a geração procedural dos níveis do jogo e armazená-los em um vetor.

O Vetor que possui todas as telas geradas proceduralmente é gerenciado pelo ator "Gerenciador de Fases" e é ele o encarregado de fazer mudanças na cena atual para simular a passagem de nível. No jogo original *Pitfall!* a movimentação do jogador é feita tando de forma bi-lateral ou seja, pode ser tanto da esquerda para a direita quanto da direita para a esquerda. O geren-

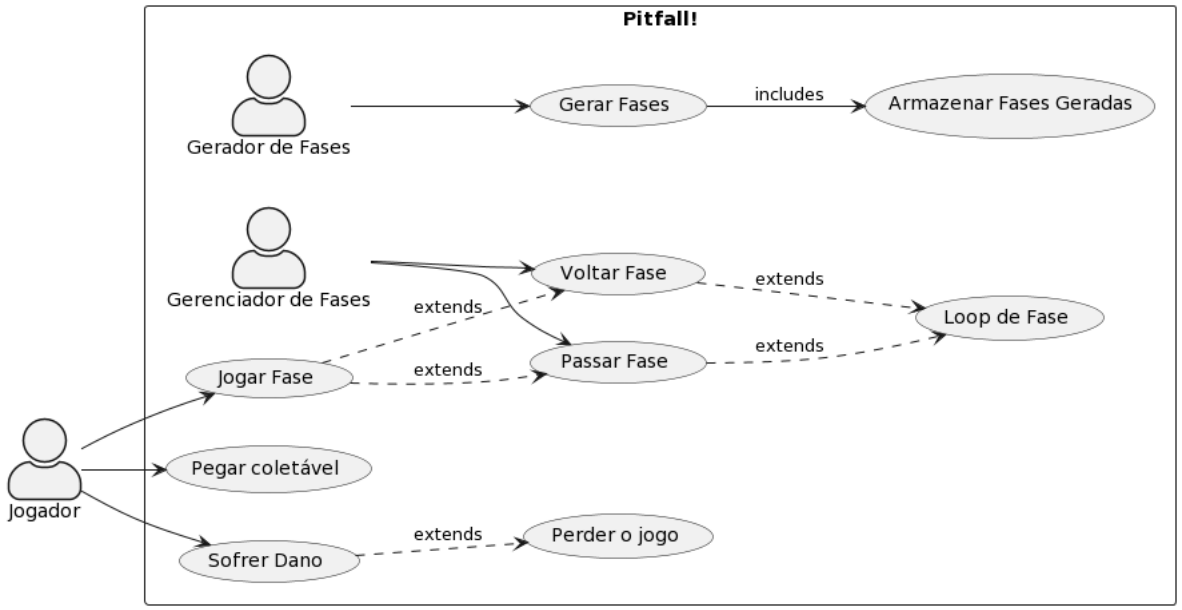


Figura 3.2: Diagrama de Caso de Uso do jogo *Pitfall!*

ciador de fases possui uma função de *looping* que faz com que caso o jogador chegue ao final do vetor de fases ele voltará para a primeira fase e vice-versa.

### 3.3.1 Diagrama de Classes

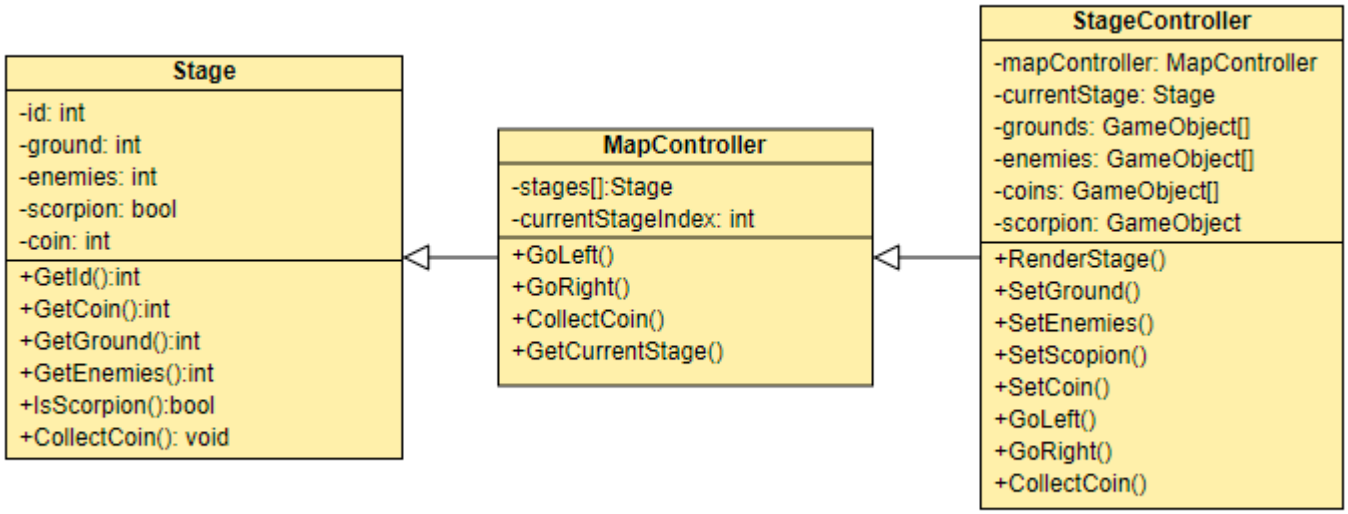


Figura 3.3: Diagrama de Classes do gerador de fases do jogo Pitfall

O Diagrama de classes acima demonstra estrutura das classes que compõe a função de criação de fases. Uma estrutura básica chamada Stage é utilizada pela classe MapController que fará a criação de todas as fases da classe Stage e os armazenará em um vetor. Já a classe StageController é a responsável pela instância dos elementos de cada uma das fases. As informações de todas as fases estão armazenadas em objeto MapController.

### 3.3.2 Modelagem C4

A modelagem C4 possui este nome pois agrega quatro características: contexto, contêineres, componentes e código. Os autores (VÁZQUEZ-INGELMO; GARCÍA-HOLGADO; GARCÍA-PEÑALVO, 2020) classificam a modelagem como um conjunto de diagramas hierárquicos que você pode usar para descrever sua arquitetura de software em diferentes níveis de zoom, cada um útil para públicos diferentes.

**Primeiro Nível: Contexto**

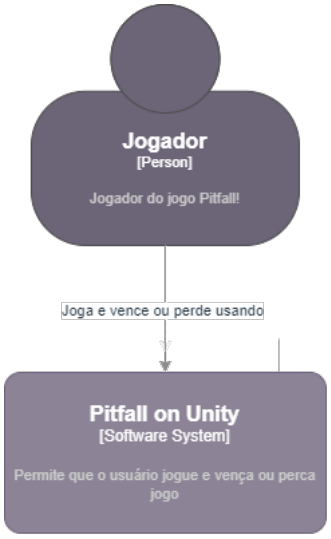


Figura 3.4: Diagrama C4 de nível 1: Contexto

O primeiro nível é contexto, detalhes não são muito importantes, apenas o entendimento de onde o software está inserido, com quem se relaciona e quais são suas restrições. Aqui temos o diagrama de C4 de nível 1 do jogo *Pitfall!*.

### Segundo Nível: Contêiner

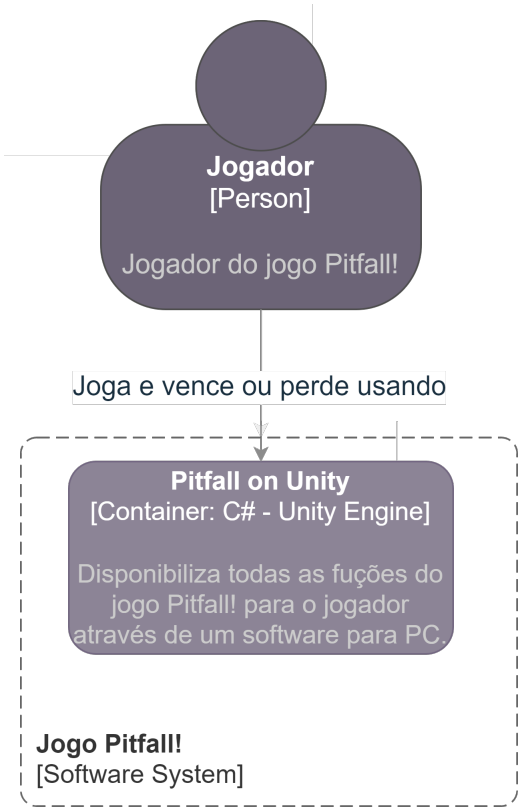


Figura 3.5: Diagrama C4 de nível 2: Contêiner

O segundo nível é o nível de contêiner, é o correspondente a delimitação. Aqui tem-se uma visão mais focada a código, expõe-se as tecnologias e sistemas a serem utilizados (aplicativos, microsserviços, banco de dados, etc.) no software a ser implementado. A seguir temos o diagrama C4 Nível 2 do jogo *Pitfall!*.

### Terceiro Nível: Componentes

O terceiro nível é reservado para os componentes. Aqui há uma quebra ainda maior nos componentes do contêiner. Cada um representa um conglomerado de abstrações ou códigos com determinada função. No caso do presente projeto, vê-se o aparecimento de alguns controladores específicos para os inimigos ou para o controle do avatar pertencente ao jogador.

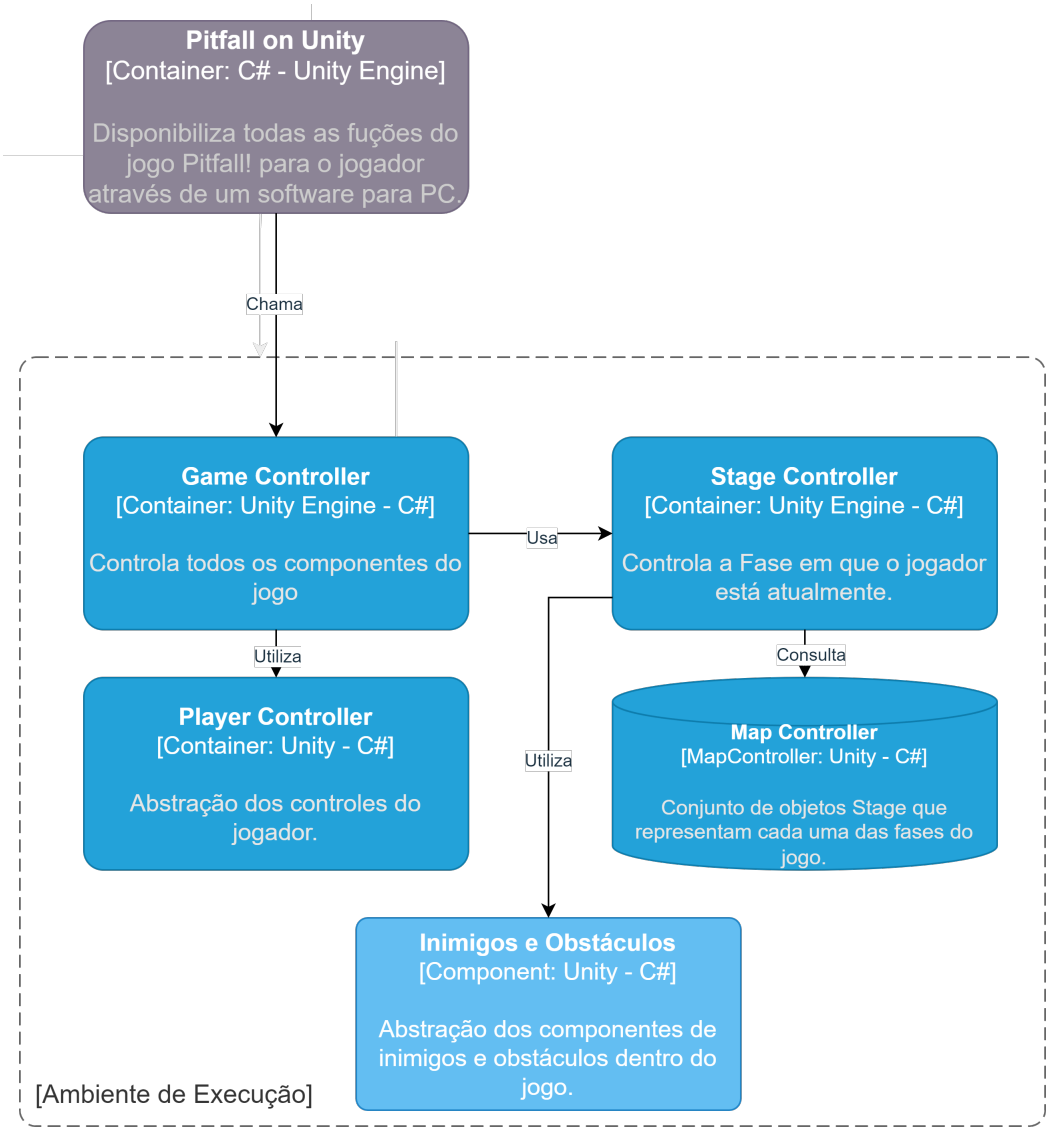


Figura 3.6: Diagrama C4 nível 3: Componentes

# Capítulo 4

# Desenvolvimento e Implementação

Neste capítulo será abordada a implementação do software propriamente dita, utilizando-se de código e discutindo em detalhes cada parte da execução do projeto. O desenvolvimento pode ser separado entre a criação do ambiente que será interagido pelo jogador e o avatar do jogador que será sua representação neste mundo virtual.

## 4.1 Criando uma selva virtual

Esta seção tem por objetivo aprofundar ainda mais a implementação do ambiente virtual das fases. Como cada uma foi abstraída a nível de código e como se faz possível a navegabilidade no mundo do jogo.

### 4.1.1 A Fase

Como discutido no capítulo anterior antes mesmo de podermos gerenciar as fazes precisamos primeiro tê-las instanciadas, foi criada uma classe chamada *Stage* com o único objetivo de armazenar as informações de cada uma das fases do jogo que forem geradas. É uma implementação de classe relativamente simples, cada um dos elementos possui o encapsulamento de *readonly* o que indica que estes campos não podem ser alterados após a instância de um objeto desta classe. Os campos *ground*, *enemies* e *coin* representam a posição do elemento no respectivo vetor no *script StageController.cs* que faz a renderização dos elementos na tela do jogador.

Listing 4.1: Classe Stage

```
1
2
3  public class Stage
4  {
5      private readonly int id;
6      private readonly int ground;
7      private readonly int enemies;
8      private readonly bool scorpion;
9      private readonly int coin;
10
11
12     public Stage(int id, int coin, int ground, int enemies, bool
           scorpion)
13     {
14         this.id = id;
15         this.coin = coin;
16         this.ground = ground;
17         this.enemies = enemies;
18         this.scorpion = scorpion;
19     }
20     public int GetId() { return id; }
21     public int GetCoin() { return coin; }
22     public int GetGround() { return ground;}
23     public int GetEnemies() {  return enemies;}
24     public bool IsScorpion() {  return scorpion;}
25
26 }
```

### 4.1.2 O Construtor do Mapa

O vetor de objetos da classe *Stage* supracitada é classificado dentro deste projeto como Mapa, e para fazer o controle do mesmo faz-se necessária a criação de um *script* para controlar a criação e navegação de cada uma das fases.

Listing 4.2: Enum referência da classe Stage

```
1      private enum groundType {safe, ladder, ladderThreeHoles, oneHole,
           threeHoles, pitfall};
2      private enum enemyType {safe, snake, fire, log, rollingLog,
           threeRolingLogs};
3      private enum hascoin {no, above, below};
```

*Enum* é um tipo nativo da linguagem C# conhecido como enumerador (BILLWAGNER, ), cada uma das posições deste enumerador pode ser acessada através da utilização *<enum>.<posição>* Ex: *groundType.safe* é a mesma coisa que *0* já que o mesmo é o primeiro na declaração supracitada. Este enumerador serve de guia na hora da instância tornado a classe mais legível pois ao invés de usarmos valores inteiros podemos utilizar uma referência a um valor do enumerador. Ao invés de *Stage.ground = 3* pode-se escrever *Stage.ground = groundType.oneHole*. Esta implementação tem o objetivo de tornar o código mais legível.

Listing 4.3: Construtor da Classe MapController

```
1   static MapController()
2   {
3       stages = new Stage[255];
4       stages[0] = new Stage(0, (int)hascoin.no, (int)groundType.safe
            , (int)enemyType.safe, true);
5
6       for (int i = 1; i < stages.Length; i++)
7       {
8           bool scorpion = Random.Range(0,2) == 0;
9           int ground = Random.Range(0,6);
10          int enemy = Random.Range(0, 6);
11          int coin = Random.Range(0, 3);
12          Stage newStage = new Stage(i, coin, ground, enemy,
                scorpion);
13          stages[i] = newStage;
14      }
15      currentStageIndex = 0;
16  }
```

Nota-se a necessidade de uma classe do tipo *static* haja-visto que a mesma não muda após a primeira execução e é utilizada por outro *script* chamado de *StageController.cs* responsável pela tela propriamente vista que será mostrada para o jogador. A partir do construtor são escolhidos de forma pseudo-randômica através das funções *Random. Range()* os valores para cada um dos campos da classe *Stage*. Nota-se também que a quantidade de fases gerada por esse construtor é de 255 que é o mesmo número de fases do jogo original.

Esta classe além de ser responsável por armazenar o vetor que compõe todas as fases do game também possui duas outras funções, *GoLeft()* e *GoRight()* que tem como retorno uma

instância da classe *Stage.cs*. É através desta instância que o *script StageController.cs* sabe quais elementos para o jogador.

Listing 4.4: Função GoRight() do Controlador do Mapa

```
internal Stage GoRight()
    {
        currentStageIndex = currentStageIndex + 1;
        if (currentStageIndex >= stages.Length)
        {
            currentStageIndex = 0;
        }
        stages[currentStageIndex]);
        return stages[currentStageIndex];
    }
```

Tanto a função *GoLeft()* e *GoRight()* possuem uma lógica interna que gera um *loop* no mapa, no caso o jogador ultrapasse os limites do vetor. No casso deste exemplo, índice da próxima fase seja maior ou igual ao tamanho máximo do vetor de fases, o índice recebe 0, ou seja, a primeira posição. Uma lógica parecida é aplicada na função *GoLeft()* onde o índice recebe o último possível do vetor.

### 4.1.3 O Controlador das Fases

O *script StageController.cs* já foi citado anteriormente em relação a outros *scripts* e agora é o responsável por renderizar todos os elementos do jogo na tela para que o jogador possa interagir. Para que isso aconteça é necessário pontuar que todos os elementos que compõe a fase como o tipo de terreno, os inimigos, os obstáculos e os coletáveis estão sempre presentes e instanciados na fase, apenas desativados. Isso é capaz graças a uma funcionalidade da *Unity* chamado *[SerializeField]*. A seguir temos um exemplo de como esta funcionalidade foi utilizada no código do controlador de fases.

Listing 4.5: Declaração do campo SerializeField

```
1     [SerializeField] private GameObject[] grounds = new GameObject[6];
2     [SerializeField] private GameObject[] enemies = new GameObject[5];
3     [SerializeField] private GameObject[] coins = new GameObject[2];
4     [SerializeField] private GameObject scorpion;
```

Destaca-se o uso do tipo *GameObject* nas declarações, a linguagem C# sendo uma linguagem orientada a objeto (BILLWAGNER, ) e utilizada dentro do contexto de desenvolvimento através da *Unity* possui várias classes e funções que são nativas deste ambientes. Uma de suas principais é o tipo *GameObject* que é o objeto *pai* de todo e qualquer elemento que é adicionado a uma cena da *Unity*. Todo e qualquer objeto presente em uma cena é derivado deste.

Após declarar um elemento com o rótulo *[SerializeField]* é possível relacionar um elemento da cena dentro do código através da ferramenta *Object Inspector* da *Unity Engine*, arrastando o objeto presente na hierarquia da cena para criar o relacionamento dentro do *script*. Sendo visualmente explícito conforme demonstra a imagem a seguir.

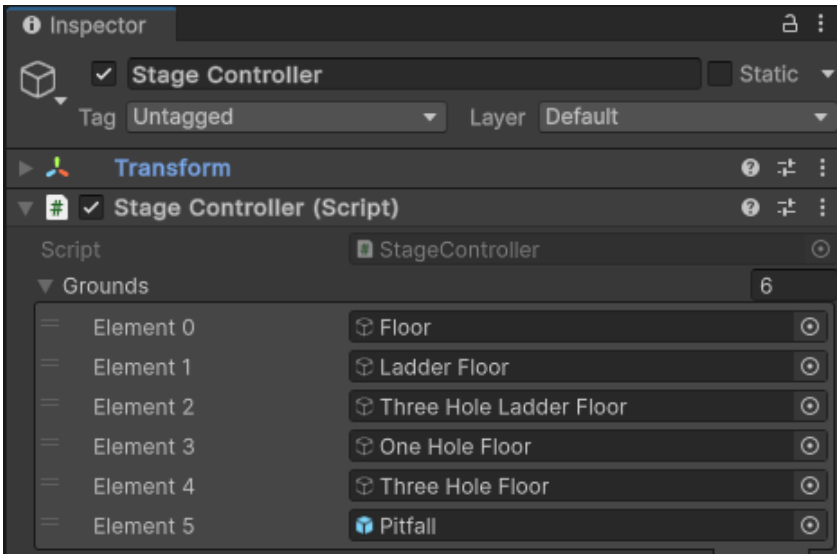


Figura 4.1: Imagem de fase com entrada para o subsolo

Feita essa relação através da *Unity*, torna-se possível a manipulação de cada um dos objetos relacionados pelo código. Neste exemplo podemos ver cada um dos tipos de chão disponíveis para cada fase. Todos estes elementos possuem sendo eles em ordem na imagem: um chão simples e sem qualquer obstáculo, um chão com apenas um buraco e uma escada, um chão com

três buracos e uma escada, um chão com apenas um buraco, um chão com três buracos e por fim o chão com areia movediça.

A seguir temos a função executada quando o jogo começa, qualquer *script* que possui uma função do tipo *void Start()* tem seu conteúdo executado antes de todas as outras linhas de código de todos os *scripts*. Aqui tem-se referências a outros *scripts* previamente descritos como o *MapController.cs* vê-se também a variável da classe *Stage* também anteriormente citada.

Listing 4.6: Método Start() da classe StageController

```
1  void Start()
2      {
3          mapController = new MapController();
4          currentStage = mapController.getCurrentStage();
5          movingPartsRespawn = new Vector3[enemies.Length];
6          for (int i = 0; i < enemies.Length; i++)
7          {
8              movingPartsRespawn[i] = enemies[i].transform.position;
9          }
10
11         renderStage();
12     }
```

A linha 5 do código acima mostra a instância do vetor *movingPartsRespawn* do tipo *Vector3*. *Vector3* é um objeto nativo da *Unity Engine* que guarda um vetor de 3 posições correspondentes a coordenadas X,Y e Z utilizadas pelo atributo *transform.position* de qualquer objeto da classe *GameObject*. A *Unity* aceita a ação: *GameObject.transform.position = Vector3()*, os dois tipos são considerados equivalentes.

Listing 4.7: Código de renderização da fase

```
public void renderStage()
{
    if (currentStage != null)
    {
        SetGround(currentStage.GetGround());
        SetEnemies(currentStage.GetEnemies());
        SetScorpion(currentStage.IsScorpion());
        SetCoin(currentStage.GetCoin());
    }
}
```

Já este código foi criado explicitamente para esse *script*. Ele é responsável por fazer a chamada de outras funções internas que, respectivamente de cima para baixo gerenciam em tela: o tipo de chão da fase, o tipo de inimigos/obstáculos da fase, se há ou não um escorpião na caverna da fase e se há ou não um coletável na fase.

Listing 4.8: Função SetGround()

```
    private void SetGround(int ground)
    {
        for (int i = 0; i < grounds.Length; i++)
        {
            if (i == ground)
            {
                grounds[i].SetActive(true);
            }
            else
            {
                grounds[i].SetActive(false);
            }
        }
    }
```

Esta função faz a instância dos terrenos na fase, vale ressaltar que estes terrenos referem-se apenas ao chão da fase e não ao chão da caverna, este mantêm-se sempre imutável. Os tipos de chão da fase são: normal, com um buraco e uma escada, com 3 buracos e uma escada, com apenas um buraco, com três buracos e por fim a areia movediça.

A função percorre todo o vetor que contém o objeto *GameObject* associado a cada um dos

tipos de terreno. Se o índice desejado *ground* for o mesmo valor de *i*, o elemento do vetor *grounds* naquele índice é ativado através da função nativa de objetos *GameObject* que é a *SetActive()*.

Quando um elemento é ativado, todos os *scripts*, animações e objetos associados a ele no *Object Inspector* da *Unity Engine* também são ativados, essencialmente trazendo-o a vida. Ao ser desativado todos estes também são. Exemplo: em uma fase (A) o chão possui três buracos e na próxima fase (B) o chão não possui nenhum buraco, se o jogador estiver na fase (B) em posição correspondente a um buraco da fase (A) o mesmo não é afetado pelo buraco pois para este contexto, o buraco não existe. Esta função torna apenas um tipo de piso para a fase ativo por vez, fazendo com que todos os outros sejam desativados através do loop.

Listing 4.9: Função SetEnemies()

```
1  private void SetEnemies(int enemy)
2  {
3      if (enemy > 0)
4      {
5          for (int i = 0; i < enemies.Length; i++)
6          {
7              if (i == enemy)
8              {
9               enemies[i].SetActive(true);
10             }
11             else
12             {
13              enemies[i].SetActive(false);
14              enemies[i].transform.position = movingPartsRespawn[i];
15             }
16         }
17     }
18     else if (enemy <= 0)
19     {
20         for (int i = 0; i < enemies.Length; i++)
21         {
22            enemies[i].SetActive(false);
23            enemies[i].transform.position = movingPartsRespawn[i];
24         }
25     }
26
27 }
```

Já a função *SetEnemies()* possui uma lógica diferente. A verificação feita com o valor de *enemy* ser maior que zero pois o valor zero significa que a fase não possui inimigos. Esta lógica faz referência ao Listing 4.2 deste documento. Se o valor de *enemy* for 0, significa que a fase não possui inimigos, neste caso todos os inimigos presentes no vetor *enemies* são desativados e a posição daquele inimigo na tela é também e redefinida para seu estado original armazenado no vetor *movingPartsRespawn*. Quando a fase possui inimigos, o vetor inteiro de inimigos também é percorrido pois faz-se necessário desativar todos os outros inimigos que não sejam o de índice *enemy* que é recebido via parâmetro pela função.

Listing 4.10: Função SetScorpion()

```
1     private void SetScorpion(bool scor)
2     {
3         if (scor)
4         {
5             scorpion.SetActive(true);
6         }
7         else
8         {
9             scorpion.SetActive(false);
10        }
11
12    }
```

A função responsável pela renderização do escorpião, o único inimigo presente na caverna da fase é esta. Ela é bem simples, como a variável *scorpion* é um *[SerializeField]* ela referencia diretamente o objeto *GameObject* referente ao escorpião na tela, a condição aqui é mais simples, ou o escorpião está ou não está na fase. Dada essa simplicidade apenas a variável booleana *scor* é o suficiente para representar esta presença na fase.

Listing 4.11: Função SetCoin()

```
1     private void SetCoin(int coin)
2     {
3         if (coin > 0)
4         {
5             for (int i = 0; i < coins.Length; i++)
6             {
7                 if (coin == i)
8                 {
9                     coins[i].SetActive(true) ;
10                }
11                else
12                {
13                    coins[i].SetActive(false);
14                }
15            }
16        }else if (coin <= 0)
17        {
18            for (int i = 0; i < 2; i++)
19            {
20                coins[i].SetActive(false);
21            }
22        }
23    }}
```

A função *SetCoin()* é bem parecida com a função *SetEnemies()*, o valor 0 simboliza que não há moedas na faze, 1 significa que a moeda está na fase no nível do chão e 2 simboliza que a moeda está na fase no nível da caverna. A função desativa ou reativa o respectivo *GameObject* da moeda com baso no valor de *coin* adquirido na chamada da função.

Listing 4.12: Funçóes GoLeft() e GoRight()

```
1     public void GoLeft()
2     {
3         currentStage = mapController.GoLeft();
4         renderStage();
5     }
6
7     public void GoRight()
8     {
9         currentStage = mapController.GoRight();
10        renderStage();
11    }
```

Esta função recebe a próxima instância do vetor de fases presente na variável *mapController* da classe descrita no item *Listing 4.3*. A instância retornada pela função é amazenada na variável *currentStage* que também é da classe *Stage.cs* e após isso chamamos a função *renderStage()* descrito no *Listing 4.7*.

### 4.1.4 Controlador de Limites da Fase

Listing 4.13: Controlador de Limites ConerController.cs

```
1  public class CornerController : MonoBehaviour
2  {
3      [SerializeField] private Transform otherSide;
4      [SerializeField] private GameObject stageController;
5      private float padding = 1.5f;
6
7      private void OnTriggerEnter2D(Collider2D collision)
8      {
9          if (collision.gameObject.CompareTag("Player"))
10         {
11             if (otherSide.gameObject.CompareTag("RightCorner"))
12             {
13                 collision.gameObject.transform.position = new Vector3(
                       otherSide.transform.position.x - padding, collision
                       .gameObject.transform.position.y, collision.
                       gameObject.transform.position.z);
14                 stageController.GetComponent<StageController>().GoLeft
                       ();
15             }
16             else if (otherSide.gameObject.CompareTag("LeftCorner"))
17             {
18                 collision.gameObject.transform.position = new Vector3(
                       otherSide.transform.position.x + padding, collision
                       .gameObject.transform.position.y, collision.
                       gameObject.transform.position.z);
19                 stageController.GetComponent<StageController>().
                       GoRight();
20             }
21         }
22     }
23
24 }
```

O controlador de limites da fase é um *script* chamado de *CornerController.cs*. Este *script*

faz o uso de um uma variável *[SerializeField]* chamada de *otherSide* que recebe um objeto *GameObject*. Tais objetos podem possuir campos chamados de *Tags* que são inseridos através da ferramenta *Object Inspector* presente na *Unity Engine*. Essas *Tags* podem ser customizadas e, no caso deste *script*, utiliza-se a função de retorno booleano *CompareTag* para definir se o objeto de tag *Player* será enviado para o lado esquerdo ou direito da fase.

Essa implementação torna possível que uma fase finita, de câmera fixa consiga simular um mapa muito maior. O *script* traz uma sobrescrição da função nativa do componente *BoxCollider2D* da classe *GameObject* , a função *OnTriggerEnter2D()*. Esta função é ativada com base em um gatilho que ocorre quando dois objetos *GameObject* que possuem componentes *Collider2D* se chocam no ambiente virtual da *Unity*. Utilizando-se do objeto da *Unity Collider2D collision* que neste caso representa uma colisão de dois objetos.

Se o objeto colisor possui a tag *Player* ele representa o jogador, o *script* sobrescreve a posição do jogador utilizando um objeto *Vector3* comentado anteriormente neste documento, preservando a altura na qual o mesmo se encontra, colocando-o apenas no canto oposto ao qual o jogador colidiu. Se ele colidiu com o canto direito da fase, o jogador reaparece no canto esquerdo e vice-versa.

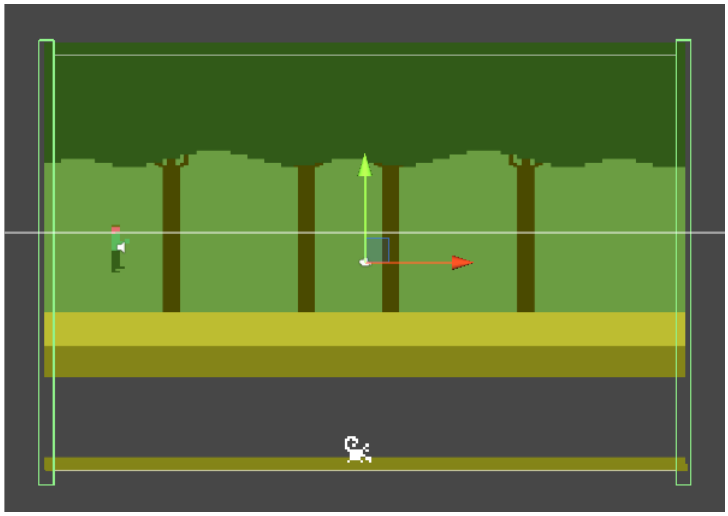

Figura 4.2: Objetos *GameObject* em verde correspondentes aos limites da fase.

O valor da variável *padding* auxilia na impressão de que o jogador realmente passou dos limites da tela já pois ele ultrapassa um pouco dos cantos da tela o que auxilia na imersão.

## 4.2 O Jogador

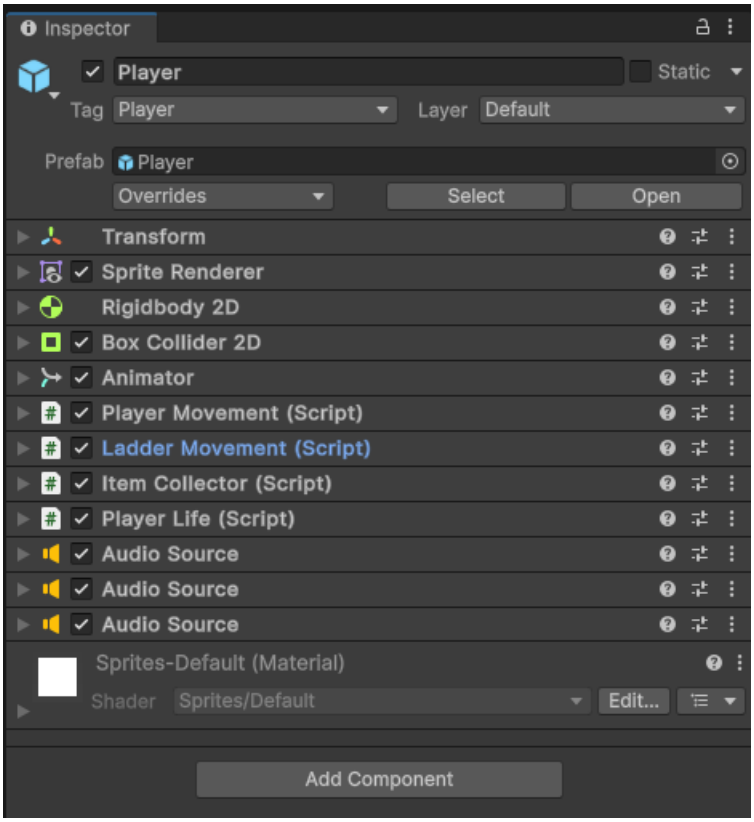


Figura 4.3: Lista de componentes do objeto *Player*

O objeto que simboliza o jogador possui a maior quantidade de componentes. Para que um *script* seja executado pela *Unity Engine* ele precisa estar associado como um componente em algum objeto *GameObject* em cena. Com base na figura acima podemos ver que o objeto representante do jogador é o que mais possui componentes. Ele é o responsável por como o jogador interage com o cenário, com os obstáculos, com os inimigos e com os coletáveis.

Além disso ele também é responsável por três efeitos sonoros distintos. Nesta seção serão explicados cada um destes *scripts* e componentes e como eles moldam a interação do jogador com o jogo. É importante destacar que todos os sons inerentes ao próprio jogador sendo eles, o som de pulo, morte e coleta de moedas são associados como componentes do objeto *Player*.

### 4.2.1 O movimento de Pitfall Harry

Listing 4.14: Componente principal do script PlayerMovement.cs

```
1  public class PlayerMovement : MonoBehaviour
2  {
3      [...]
4      private void Update()
5      {
6          dirX = Input.GetAxisRaw("Horizontal");
7
8          rbPlayer.velocity = new Vector2(dirX * moveSpeed, rbPlayer.
               velocity.y);
9
10         if (Input.GetButtonDown("Jump") && isGrounded())
11         {
12             jumpSFX.Play();
13             rbPlayer.velocity = new Vector2(rbPlayer.velocity.x,
                   jumpForce);
14         }
15         UpdateAnimationState();
16     }
17     [...]
18 }
```

A *Unity Engine* permite a configuração de controles customizados para todos os tipos de jogos, a entrada realizada pelo jogador, ou seja, o botão apertado por ele consegue ser capturado através do objeto *Input*. A função *GetAxisRaw()* retorna um valor decimal do tipo ponto flutuante *float*. O parâmetro *Horizontal* delimita que apenas o botão (*Input*) correspondente à movimentação horizontal seja capturado no código acima. No caso desta implementação os botões de movimentação horizontal são: [A,D,←, →]

Este valor é utilizado no componente *RigidBody2D* do objeto *Player* fazendo com que a velocidade do personagem seja aplicada conforme o clique do jogador (linha 8 do código acima). A função *Update()* de qualquer *script* adicionado a um objeto *GameObject* é executada uma vez por *frame* de execução do jogo.

Listing 4.15: Função IsGrounded()

```
1    private bool isGrounded()
2    {
3        return Physics2D.BoxCast(collPlayer.bounds.center, collPlayer.
             bounds.size, 0f, Vector2.down, .1f, jumpableGround);
4    }
```

A função *IsGrounded()* utiliza-se de uma projeção em formato de caixa do componente *Physics2D* que retorna um valor booleano correspondente a posição do jogador. Se ele estiver em contato com um objeto considerado "solo pulável" ou seja, uma superfície que representa um chão em que o boneco do jogador está "de pé" sobre, o retorno é positivo (*true*).

Para que o jogador possa performar o efeito sonoro de pulo e receber velocidade no eixo Y, ou seja, para cima. A parte de baixo do boneco precisa estar conectada com o solo e o jogador precisa ter pressionado o botão de pular, para esta implementação este botão é a barra de espaço.

Por último faz-se necessário ajustar o *sprite* do boneco do jogador e animá-lo, se ele etiver indo para esquerda, o boneco precisa estar olhando para esquerda. O mesmo vale para a direita. Se o boneco está pulando, o *sprite* do boneco precisa refletir este estado. Para isso utiliza-se o componente *Animator*.

Listing 4.16: Lógica de animação do boneco *Player*

```
    [...]
    private enum MovementState {idle, running, jumping, falling}
    [...]
    private void UpdateAnimationState()
    {
        MovementState state;
        if (dirX > 0f)
        {
            state = MovementState.running;
            sprRenPlayer.flipX = false;
        }
        else if (dirX < 0f)
        {
            state = MovementState.running;
            sprRenPlayer.flipX = true;
        }
        else
        {
            state = MovementState.idle;
        }

        if (rbPlayer.velocity.y > .1f)
        {
            state = MovementState.jumping;
        }
        else if (rbPlayer.velocity.y < -.1f)
        {
            state = MovementState.falling;
        }


        aniPlayer.SetInteger("state",(int) state);
    }
```

Caso a variável *dirX* seja positiva, quer dizer que o jogador comanda o boneco a ir para a direita, se o valor for negativo, o jogador está a comandar o boneco a ir para a esquerda, e neste caso o *sprite* precisa ser virado no eixo X. Caso o valor de *dirX* seja zero, o jogador não está movimentando o boneco nem para direita e nem para a esquerda. Isso não quer dizer que o mesmo esteja se movimentando pois o boneco pode estar caindo ou pulando.

É neste movendo que entra a segunda validação lógica, se a velocidade no eixo Y for maior de 0.1 o boneco está se movendo para cima. Se a velocidade no eixo Y for menor que 0.1 quer

dizer que o boneco está caindo, indo para baixo.

Percebe-se que todos estes estados lógicos da animação são repassados utilizando o *enum* MovementState, este valor é um número inteiro repassado para o componente *Animator* que utiliza-o logicamente para alternar qual a animação está sendo mostrada para o jogador na tela.

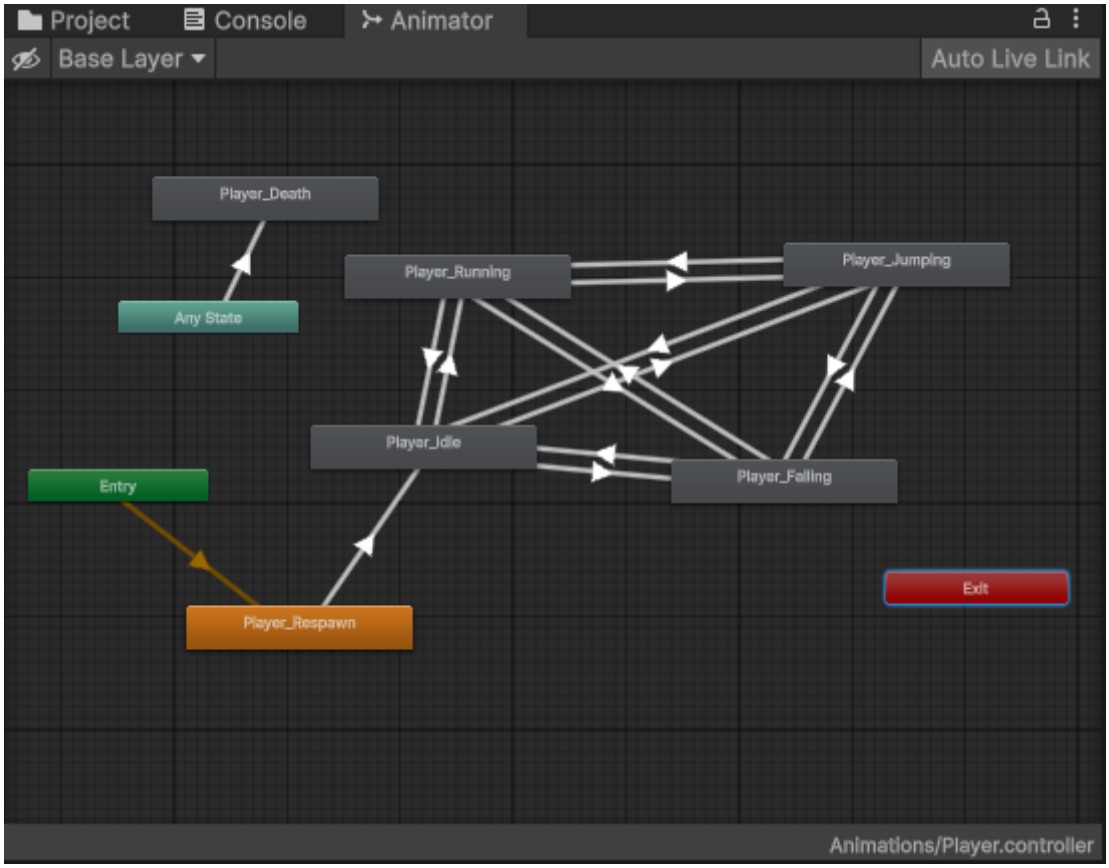


Figura 4.4: Ferramenta *Animator* mostrando todos os status do objeto *Player*

Cada uma das células representadas na imagem acima possui um identificador que corresponde ao *enum* do código apresentado anteriormente. Durante a execução do jogo o estado fica flutuando entre cada um dos valores do *enum* e consequentemente entre cada uma destas células. As retas com setas significam as transições de status que podem acontecer. Por exemplo, a animação *Player_Respawn* só acontece uma vez e quando termina vai para a animação *Player_Idle*. Já a *Player_Idle* consegue ir e voltar para as células: *Player_Running*, *Player_Jumping* e *Player_Falling*.

Listing 4.17: Script LadderMovement.cs

```
public class LadderMovement : MonoBehaviour
{
    [...]
    void Update()
    {
        vertical = Input.GetAxisRaw("Vertical");
        if (isLadder && Mathf.Abs(vertical) > 0f)
        {
            isClibing = true;

        }else if (isLadder && Mathf.Abs(vertical) < 0f)
        {
            isClibing = true;
        }
    }
    private void FixedUpdate()
    {
        if (isClibing)
        {
            rbPlayer.gravityScale = 0;
            rbPlayer.velocity = new Vector2(rbPlayer.velocity.x,
               vertical * speed);
        }
        else
        {
            rbPlayer.gravityScale = 3;
        }
    }
    private void OnTriggerEnter2D(Collider2D collision)
    {
        isLadder = collision.CompareTag("Ladder");

    }
    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Ladder"))
        {
            isLadder = false;
            isClibing = false;
        }
    }
}
```

Como o *GameObject Player* é o único que possui a possibilidade de subir e descer as escadas em fases. Ele também é o único que possui o *script LadderMovement.cs* como componente.

Este possui uma lógica similar a do *script PlayerMovement.cs* onde novamente é analisada uma colisão *Collider2D* entre dois objetos que possuem componentes do tipo *Collider2D*.

A função *Uptade()* verifica a cada frame do jogo se a entrada do jogador é referente ao movimento vertical e diferente de 0, se for e o jogador está em contato com uma escada (variável *isLadder*). No caso desta verificação ser verdadeira, a função *FixedUpdate()* manipula o componente *RigidBody2d* do objeto *Player* para que o mesmo consiga se movimentar verticalmente em contato com a escada.

Neste *script* tem-se a sobrescrição de duas funções, *OnTriggerEnter2D()* e *OnTriggerExit2D()*, estas funções são executadas apenas uma vez quando o gatilho do *Collider2D* é disparado. Em *OnTriggerEnter2D()* faz-se com que a variável booleana *isClibing* reflita se a colisão foi ou não com uma escada, representada pela etiqueta *Ladder* presente no *Object Inspector*.

Já *OnTriggerExit2D()*, que é executada quando os dois objetos deixam de colidir, quando o *Player* deixa de colidir com a escada, as variáveis *isLadder* e *isClibing* que representam que o jogador está em contato com uma escada e que ele está subindo/descendo a escada respectivamente, recebem falso.

### 4.2.2 Coletando Moedas

Listing 4.18: Script ItemCollector.cs

```
[...]
public class ItemCollector : MonoBehaviour
{
    private int coins = 0;
    [SerializeField] private Text coinCounter;
    [SerializeField] private AudioSource collectSFX;
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("Coin"))
        {
            collectSFX.Play();
            collision.gameObject.SetActive(false);
            coins++;
            coinCounter.text = "Treasures:␣" + coins;
            stageController.GetComponent<StageController>().
               CollectCoin();
        }
    }
}
```

Assim como as escadas, o objeto *Player* é o único que possui a habilidade de coletar as moedas que podem estar espalhadas pela fase. Assim como o *Script LadderMovemen.cs* há a sobrescrição do método *OnTriggerEnter2D*.

Verifica-se se a colisão foi com um objeto que possui a etiqueta *Coin*. Se a premissa for verdadeira, desativa-se o objeto com o qual o *Player* colidiu, soma-se mais um a variável inteira *coins* e atualiza-se o contador mostrado na tela do jogador para que ele possa acompanhar o seu progresso.

Por último é chamada a função da instância única e principal do *script StageController.cs* que está associada ao objeto *Stage Controller* na fase. Esta é a mesma instância acessada pelo *CornerController.cs* por exemplo.

### 4.2.3 Controlando a vida de Pifall Harry

Listing 4.19: Script PlayerLife.cs

```
1  [...]
2  public class PlayerLife : MonoBehaviour
3  {
4      private Animator anim;
5      private Rigidbody2D rbPlayer;
6      [SerializeField] private AudioSource deathSFX;
7
8      void Start()
9      {
10         anim = GetComponent<Animator>();
11         rbPlayer = GetComponent<Rigidbody2D>();
12     }
13
14     private void OnCollisionEnter2D(Collision2D collision)
15     {
16         if (collision.gameObject.CompareTag("Trap"))
17         {
18             Die();
19         }
20     }
21  1619
22     private void Die()
23     {
24         deathSFX.Play();
25         rbPlayer.bodyType = RigidbodyType2D.Static;
26         rbPlayer.velocity = Vector2.zero;
27         anim.SetTrigger("death");
28     }
29
30     private void RestartLevel()
31     {
32         SceneManager.LoadScene(SceneManager.GetActiveScene().name);
33     }
34 }
```

Para controlar as vidas do jogador, o objeto *Player* também possui este *script*. Ao começar o jogo, o script captura e armazena a referência aos componentes *Animator* e *RigidBody2D* nas linhas 10 e 11 respectivamente. Ao detectar uma colisão com um objeto que possui a etiqueta *Trap*, executa-se a função *Die()* que é exclusiva deste *script*.

A função *Die()* faz com que o efeito sonoro de morte do jogador seja disparado e através

da manipulação do componente *RigidBody2D* objeto *Player* faz com que o jogador perca momentaneamente a capacidade de controlar Pitfall Harry, porém apenas enquanto a animação de morte é mostrada.

Por último, quando o jogador "morre", função *RestartLevel()* é chamada. *SceneManager* é um componente nativo da *Unity Engine* que faz o gerenciamento das *Scenes*, com ele pode-se sair de uma cena a outra, encerrar o jogo ou até mesmo voltar ao menu inicial. No caso desta implementação, a *Scene* é reiniciada, porém, o mapa que foi previamente instanciado se mantém, bem como a pontuação do jogador.

## 4.3 Os Obstáculos

Uma série de obstáculos foi criada a fim de impedir o progresso e propor maiores desafios para o jogador. Nesta sessão falaremos de obstáculos físicos que representam inimigos para Pitfall Harry. São eles, o tronco, a cobra, a fogueira e o escorpião.


Figura 4.5: Inimigos marcados com a etiqueta *Trap*

Nota-se também o contorno verde em cada um dos inimigos, este contorno representa o

componente *Collider2D* amplamente discutido em outras seções deste documento.

### 4.3.1 Inimigos Estáticos

Alguns inimigos são estáticos, ou seja, não se movem. Estes são o a fogueira, a cobra e o tronco. O tronco porém tem suas exceções já que o mesmo pode ou não mover-se de acordo com o que for decidido pelo *script* de criação de fases.

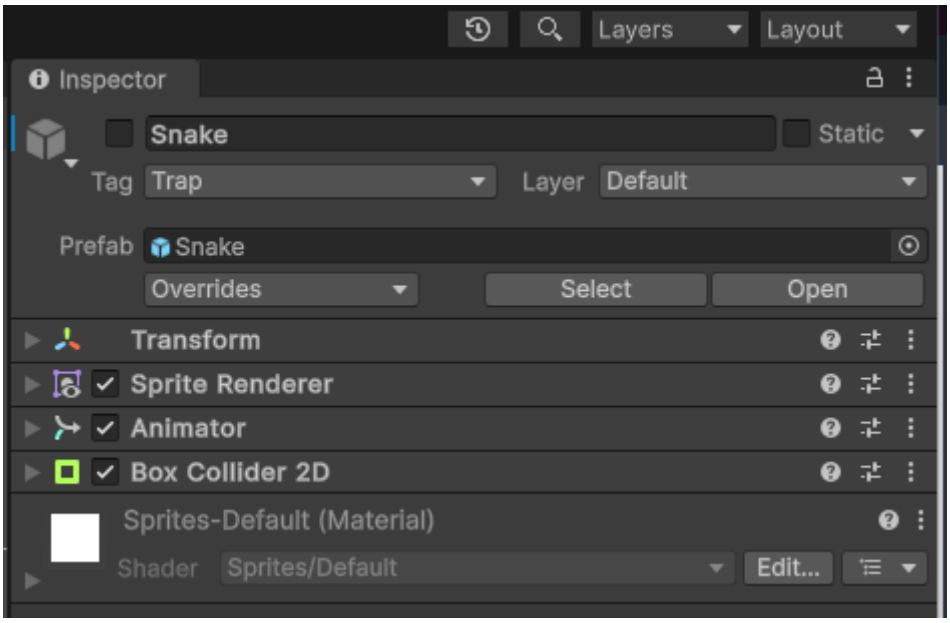


Figura 4.6: Inspetor do objeto *Snake*

Nota-se na imagem anterior a presença da etiqueta (*Tag*) "Trap". Essa é a etiqueta utilizada pelo script *PlayerLife.cs*. Por serem objetos estáticos, estes inimigos não precisam de auxilio de nenhum *script* para realizarem suas funções, apenas os componentes corretos e a utilização de etiquetas, diminuindo assim o retrabalho na hora do processamento do jogo.

### 4.3.2 Inimigos Dinâmicos

Inimigos que possuem movimento são chamados de "Dinâmicos". Estes sim possuem *scripts* para controlar o seu movimento. Inimigos considerados dinâmicos são: Um tronco móvel, Três troncos móveis e por fim o escorpião presente na caverna.

### Troncos Móveis

Listing 4.20: RollingLogController.cs

```
[...]
public class RollingLogController : MonoBehaviour
{
    [SerializeField] private float speed = 8f;
    [SerializeField] private GameObject target;
    [SerializeField] private GameObject respawn;
    void Update()
    {
        if (Vector2.Distance(target.transform.position, transform.
           position) < .1f)
        {
            transform.position = respawn.transform.position;
        }
        transform.position = Vector2.MoveTowards(transform.position,
           target.transform.position, Time.deltaTime * speed);

    }
}
```

O código responsável pelo movimento dos troncos é o mesmo, ele utiliza-se de objetos "vazios" da classe *GameObject* que estão presentes na fase. Os troncos sempre rolam da direita para a esquerda e sempre na mesma velocidade, *8f*. O objeto *target* representa a coordenada na fase para onde o tronco deve seguir. Em termos cartesianos, o movimento ocorre apenas no eixo X (VOLKWYN et al., 2020) o que significa que não há movimento vertical, apenas horizontal.

Na função *Update()* verifica-se se o tronco está perto do seu destino no espaço virtual da *Scene*. Caso essa distância seja menor que *0.1f* a posição do tronco é sobrescrita com a posição do objeto *respawn*. Caso o contrário, utiliza-se a função *MoveTowards* da classe nativa *Unity Vector2*. Esta função faz com que o objeto, ao invés de assumir bruscamente uma nova posição, desloque-se levemente conforme ditado pela velocidade.

### O Escorpião

O Escorpião é um caso mais específico, ele é o único inimigo encontrado na caverna e possui uma movimentação em loop. Da esquerda para a direita e depois da direita para a esquerda.

Listing 4.21: Script WaypointFollower.cs

```
[...]
public class WaypointFollower : MonoBehaviour
{
    private int curWaypointIndex = 0;
    private SpriteRenderer sprRenderer;
    [SerializeField] private float speed = 2f;
    [SerializeField] private GameObject[] waypoints = new GameObject
       [2];

    private void Start()
    {
        sprRenderer = GetComponent<SpriteRenderer>();
    }

    private void Update()
    {
        if (Vector2.Distance(waypoints[curWaypointIndex].transform.
           position, transform.position) < .1f)
        {
            sprRenderer.flipX = true;
            curWaypointIndex++;
            if (curWaypointIndex >= waypoints.Length)
            {
                curWaypointIndex = 0;
                sprRenderer.flipX = false;
            }
        }
        transform.position = Vector2.MoveTowards(transform.position,
           waypoints[curWaypointIndex].transform.position, Time.
           deltaTime * speed);
    }
}
```

No momento que o jogo a inicializado é executada a função *Start()*. O código armazena a referência ao componente *SpriteRenderer* para que ele possa ser manipulado. Este script pode ser aplicado em qualquer objeto que precise seguir um padrão fixo de movimentação em loop entre dois pontos. Utilizando-se de um vetor de dois objetos sem componentes *GameObject* presentes na cena *Unity*. A função *Update()* confirma se o objeto atual está fisicamente perto do seu alvo, caso esteja o *sprite* do escorpião é virado no eixo X, ou seja, se ele estava apontando para a direita agora ele estará apontado para a esquerda.

### 4.3.3 A areia movediça

A Areia movediça é um caso a parte, ela não se enquadra como um inimigo como o escorpião ou a cobra mas sim como um obstáculo de substitui o solo da fase. Entrar em contato com a areia movediça faz com que o jogador perca instantaneamente devido a sua etiqueta "Trap".

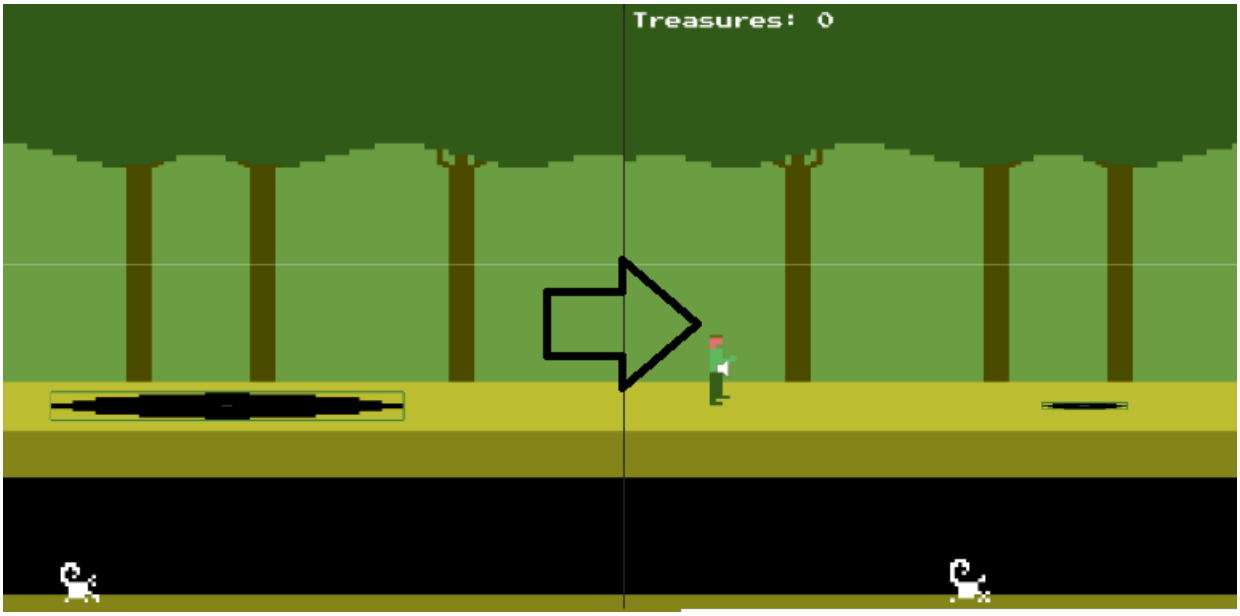


Figura 4.7: Animação da Areia Movediça

A caixa verde destacada no objeto é o componente *BoxCollider2D*. Este componente tem seu tamanho escalonado junto com os valores do campo *Scale* do componente *Transform* deste objeto. O escalonamento propriamente dito deste objeto dá-se através da ferramenta *Animator*.

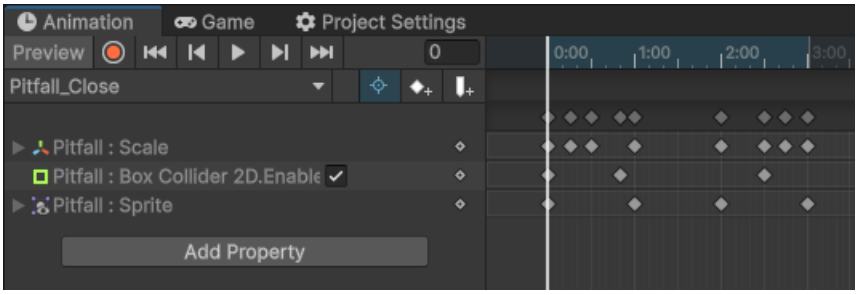


Figura 4.8: Animação da areia movediça na ferramenta *Animator*

Em dado momento, o componente *BoxCollider2D* é desativado, sem ele não há colisão para ser detectada pelo *script PlayerLife.cs* sendo assim o jogador não sofre qualquer dano.

## 4.4 Montando o Cenário

Além do nível de código, muita coisa também é feita a nível visual devido a natureza do desenvolvimento utilizando *Unity*. Para a criação do fundo de tela da fase principal foi utilizado um arquivo de imagem do tipo PNG que contém as árvores e o piso da caverna.

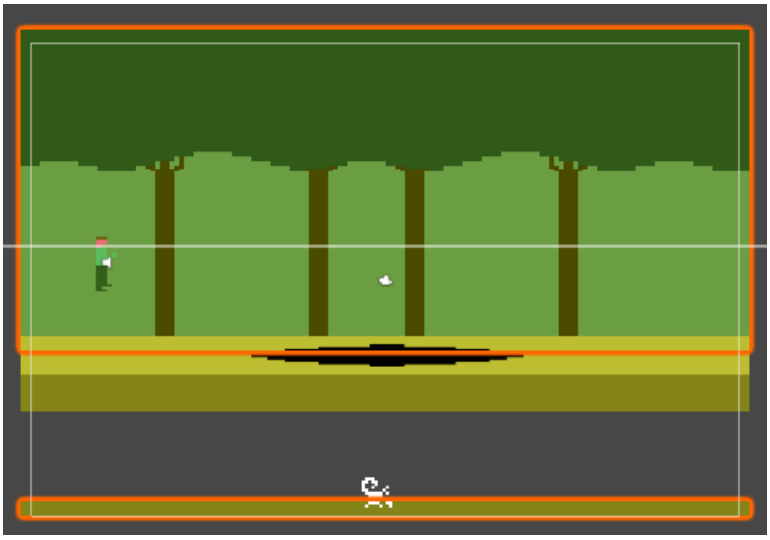

Figura 4.9: Objeto *Background* no editor de cenas da *Unity*

Em laranja vê-se destacado apenas os elementos visuais do fundo de tela. O piso, o jogador e o escorpião estão em camadas separadas de renderização. Sendo a camada mais ao fundo o *Background*.

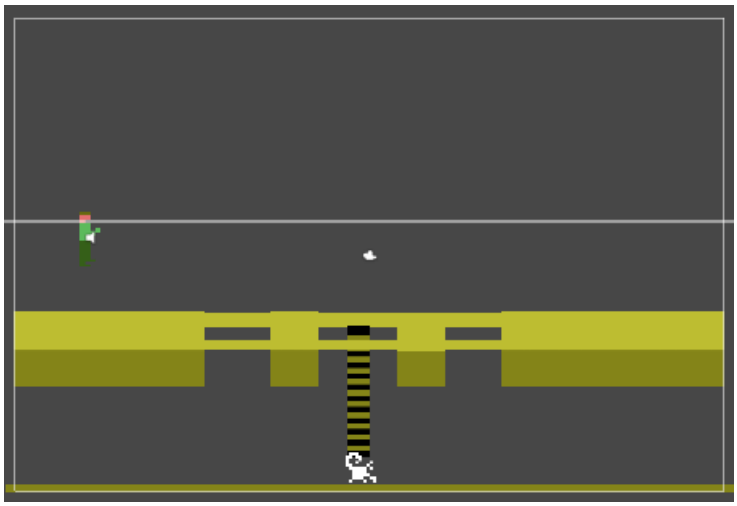

Figura 4.10: Fase sem *Background* no editor de cenas da *Unity*

### 4.4.1 O piso e a etiqueta *Ground*

Viu-se também neste documento a implementação da checagem *isGround* no arquivo *PlayerMovement.cs*. Esta etiqueta está presente nos objetos que compõe o piso da fase e o piso da caverna. Esta verificação impede o jogador de pular caso não esteja no solo.

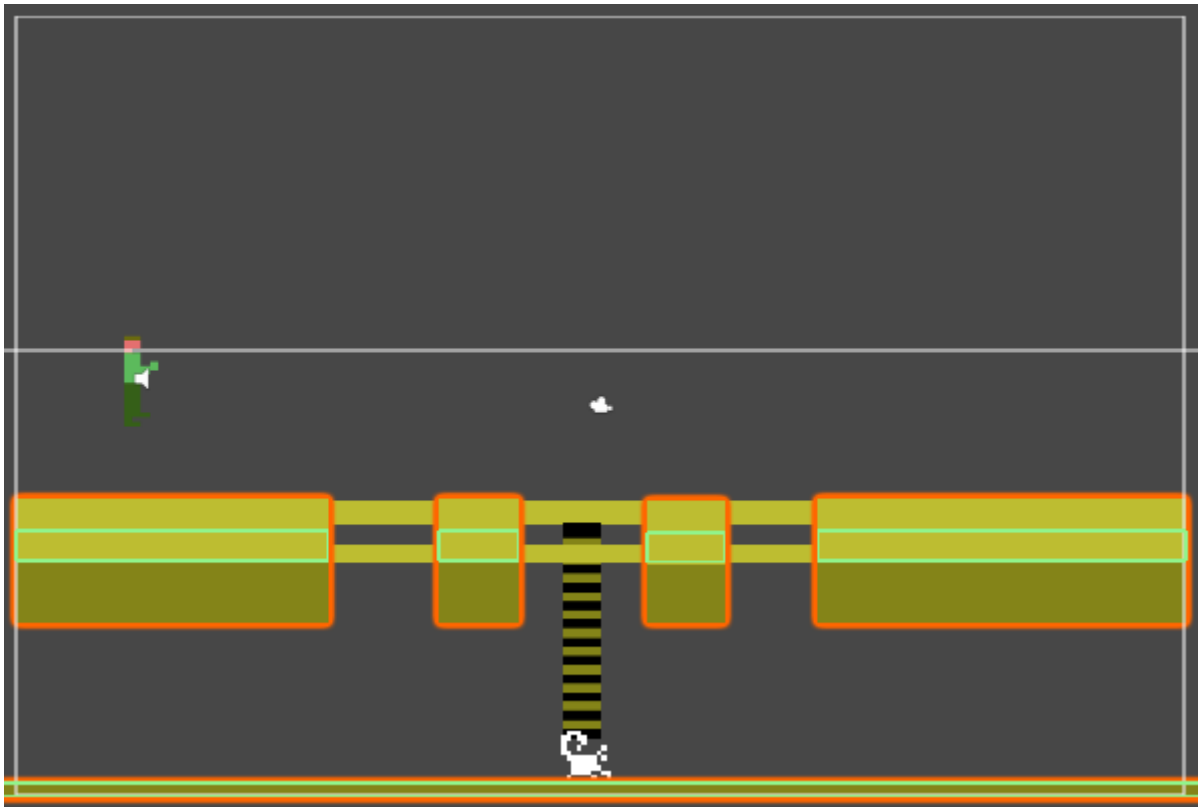

Figura 4.11: Destaque para os elementos de solo da fase

Em laranja temos o contorno do objeto que representa o solo como um todo. Em verde, temos os componentes colisores *Collider2D*. Nota-se que os colisores são menores que o tamanho total do objeto, isso se dá pelo fato dos colisores serem utilizados pelos componentes *RigidBody2D* a fim de delimitar uma área "sólida" pela qual nenhum outro corpo sólido consiga passar.

## 4.5 Animando os Personagens

A animação dos personagens foi feita através de *Sprites* no formato de imagem PNG. Alinhado com a ferramenta *Animator* presente na *Unity*. Cada fragmento da animação é composto de

pelo menos duas imagens.

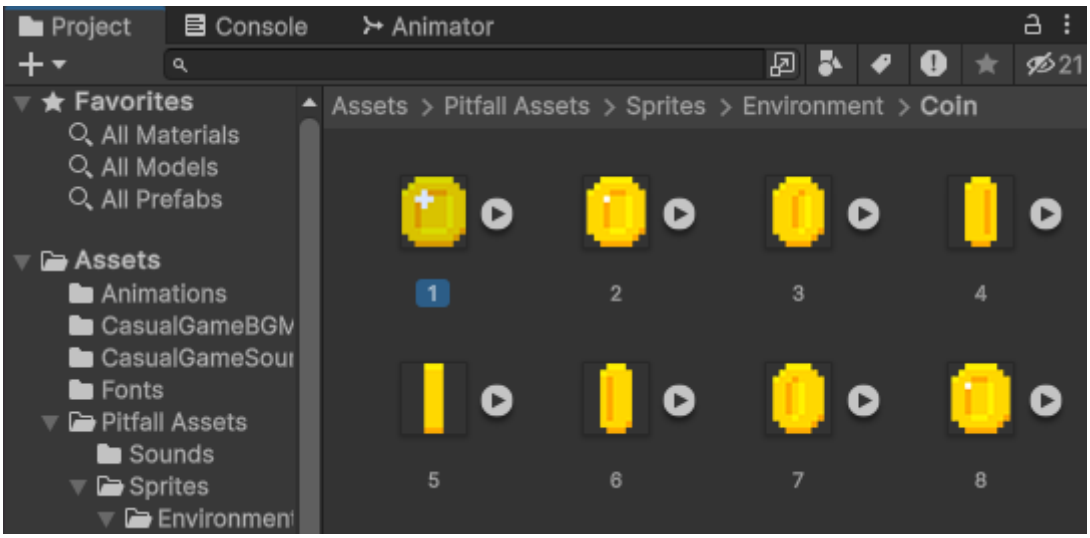


Figura 4.12: Imagens que compõe a animação da moeda

Primeiro coloca-se um objeto *GameObject* na cena. Depois, adicionam-se os componente *Sprite Renderer* e *Animator* para que associe-se uma imagem estática e a animação propriamente dita ao objeto. É importante que todos os aquivos que representam o *frame* da animação sejam do mesmo tamanho pois facilita a animação e evita a quebra de *sprites*.

Com o auxílio da ferramente *Animator* consegue-se criar todas as animações necessárias. No caso da moeda, as oito imagens são levadas para a ferramenta e dispostas ao longo de 4 segundos.

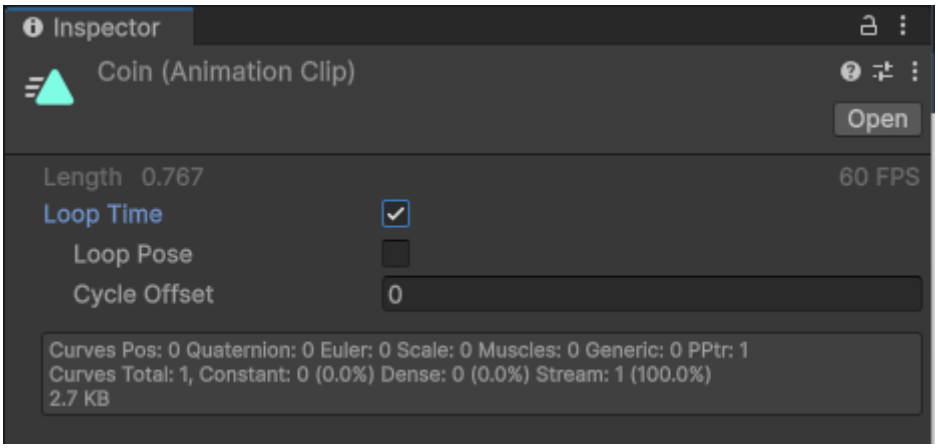


Figura 4.13: Opção Loop Time marcada no inspetor de objetos

Salvo o objeto do jogador, todos os outros objetos possuem uma animação. Esta animação

precisa ficar em loop e por causa dessa necessidade faz-se necessária a ativação da opção *Loop Time* através do inspetor de objetos da *Unity*.

## 4.6 A sonoplastia da floresta

A fim de aumentar ainda mais a imersão do jogador, faz-se necessária também a implementação de áudio (GORMANLEY, 2013). Tanto a música de fundo quanto os efeitos sonoros do jogo. Para isso, utiliza-se o componente *Audio Source* em associação com o componente *Audio Listener* pois, nenhum som pode ser ouvido se não há nenhum receptor para ele. No caso desta implementação, o objeto "ouvinte" é a própria câmera do jogo, ela captura os efeitos sonoros do jogador ou pular, nascer e morrer bem como a música de fundo que é reproduzida em loop.

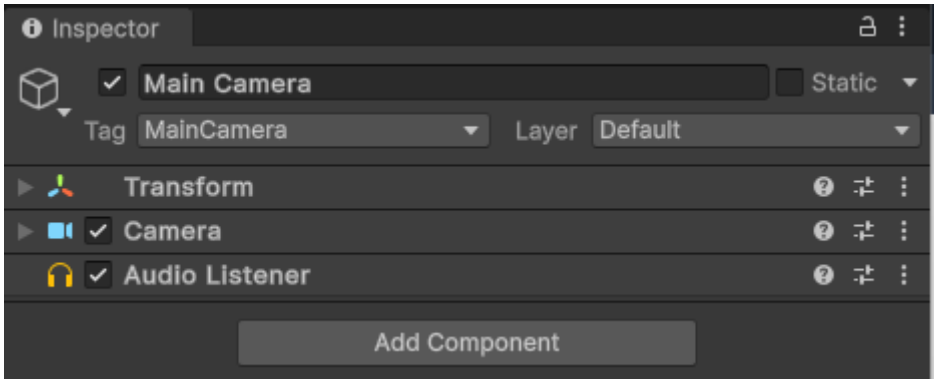


Figura 4.14: Objeto da câmera visto no inspetor da *Unity Engine*

Já o som de fundo do jogo fica em um objeto bem parecido com vistos anteriormente neste documento. É um objeto sem componente *Sprite Renderer*, ou seja, invisível porém presente na fase. A Unity apresenta suporte para áudio Stereo e dinâmico mas tal funcionalidade não é utilizada neste projeto. Todo e qualquer efeito sonoro ou música fica "no mesmo nível" para o jogador que estiver ouvindo.

Vale ressaltar que a música de fundo precisa ficar em *loop*, por isso ela possui a marcação *Loop*. Outra marcação importante neste objeto é a *Play On Awake* que faz com que a música seja tocada assim que a cena for iniciada, e não para até que o jogador saia da cena, seja vencendo o jogo ou perdendo.

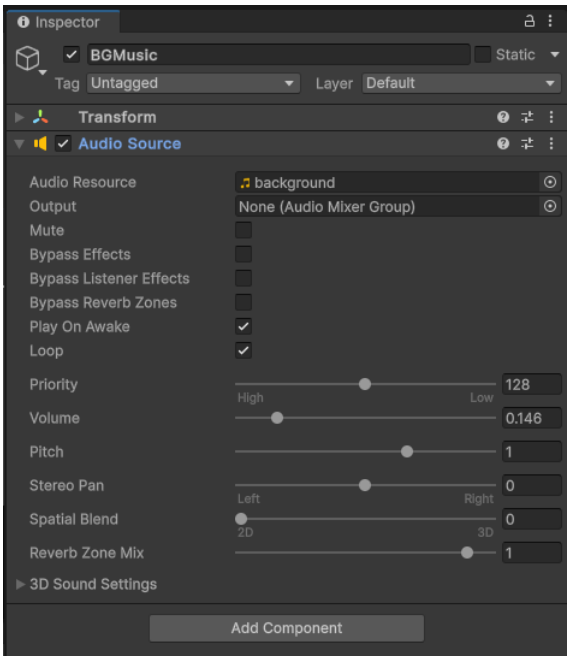


Figura 4.15: Objeto BGMusic visto no inspetor da *Unity*

## 4.7 Testes

A fim de avaliar a qualidade do *software* desenvolvido, durante todo o processo de desenvolvimento, foram realizados diversos testes de diferentes níveis. Desde testes por função e por recurso (TALBY et al., 2006) e com base nestes resultados descobriu-se por exemplo que com a implementação inicial, os coletáveis eram reabilitados ao sair da cena e em seguida voltar. Este problema influencia diretamente na experiência do jogo já que o mesmo poderia ser terminado apenas entrando e saindo de uma fase para pegar a moeda. A partir daí mudou-se a implementação dos *scripts: ItemCollector.cs, MapController.cs, StageController.cs e Stage.cs.*

A mudança principal ocorreu no código do arquivo *MapController.cs.* Foi implementada uma função que sobrescreve a instância da fase armazenada no vetor *stages* do objeto *MapController.cs.*

Listing 4.22: Correções feitas para a coleta de itens

```
1     // Script StageController.cs
2     public void CollectCoin()
3     {
4         mapController.CollectCoin();
5     }
6     // Script MapController.cs
7     internal void CollectCoin()
8     {
9         Stage current = getCurrentStage();
10
11        if (current.GetCoin() != 0)
12        {
13            current.CollectCoin();
14            stages[currentStageIndex] = current;
15        }
16    }
17    //Script Stage.cs
18    public void CollectCoin()
19    {
20        this.coin = 0;
21    }
22    //Script ItemCollector.cs
23    private void OnTriggerEnter2D(Collider2D collision)
24    {
25        if (collision.gameObject.CompareTag("Coin"))
26        {
27            [...]
28            stageController.GetComponent<StageController>().
                  CollectCoin();
29        }
30    }
```

O script *ItemCollector.cs* chama a função *CollectCoin()* do *StageController.cs* que por sua vez invoca a função de mesmo nome do script *MapController.cs* que por fim faz a sobrescrição da informação de moeda da fase no vetor.

# Capítulo 5

# Resultados Finais

Como resultado da implementação de tudo documentado nesta peça, chega-se a conclusão de que o software final resultante atende as expectativas. Tendo como principal objetivo uma reprodução que consiga retomar a experiência de se jogar *Pitfall!* porém utilizando tecnologias mais recententes.

O mais desafiador desta implementação foi com certeza a criação das fases, fazer a classificação dos diferentes componentes possíveis para uma fase e como cada um deles se comporta foi uma tarefá árdua e trabalhosa. O processo de depuração trouxe várias melhorias cada vez que era executado. Outra dificuldade foi implementar a criação de fases de maneira escalável, utilizando-se da lógica original de David Crane para abstrair quais elementos precisam estar na tela.

A implementação deste trabalho só foi possível graças a robusta e completa ferramenta *Unity Engine*. Todas as funcionalidades necessárias estavam presentes. A utilização de etiquetas facilitou muito a implementação de inimigos e obstáculos, basta um *Collider2D* e a etiqueta "Trap" para que o código fosse executado, tornando a implementação mais limpa pois não havia a necessidade de implementar ou adicionar um script por inimigo.

Por fim, como atividades futuras podem ser implementados novos inimigos e mecânicas presentes em outros jogos da série *Pitfall!* tendo em vista que este trabalho teve como base a versão original para o console *Atari 2600*. Mecânicas como cavernas de múltiplos níveis de profundidade ou novos inimigos como morcegos e fantasmas presentes em outros jogos da

franquia.

Há também a possibilidade da implementação de sprites mais fieis ao jogo original haja visto que os utilizados neste projeto tiveram sua origem em diversos sites online e nenhum recurso sonoro utilizado foi o mesmo do jogo original.

# Referências Bibliográficas


BILLWAGNER. *A tour of C - overview - C*. Disponível em: <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>.

CIPOLLA-FICARRA, F. V. Handbook of research on software quality innovation in interactive systems. In: _____. [S.l.]: Engineering Science Reference, an imprint of IGI Global, 2021. p. 267–285.

ERMI, L.; MÄYRÄ, F. Fundamental components of the gameplay experience: Analysing immersion. 2005.

FORTIM, I. et al. Pesquisa da indústria brasileira de games 2022. *TIC CULTURA*, p. 113, 2022.

GORMANLEY, S. Audio immersion in games—a case study using an online game with background music and sound effects. *The Computer Games Journal*, Springer, v. 2, p. 103–124, 2013.

MINKKINEN, T. Basics of platform games. Kajaanin ammattikorkeakoulu, 2016.

SENARATH, U. S. Waterfall methodology, prototyping and agile development. *Tech. Rep.*, p. 1–16, 2021.

SHORT, T.; ADAMS, T. *Procedural generation in game design*. [S.l.]: CRC Press, 2017.

SICART, M. Defining game mechanics. *Game studies*, v. 8, n. 2, p. 1–14, 2008.

TADEU, V.; TORTELLA, T. Público gamer cresce e 3 em cada 4 brasileiros consomem jogos eletrônicos. *CNN*, Apr 2022.

TALBY, D. et al. Agile software testing in a large-scale project. *IEEE software*, IEEE, v. 23, n. 4, p. 30–37, 2006.

TOLEDO, L. A.; SHIAISHI, G. d. F. Estudo de caso em pesquisas exploratórias qualitativas: um ensaio para a proposta de protocolo do estudo de caso. *Revista da FAE*, v. 12, n. 1, nov. 2016. Disponível em: <https://revistafae.fae.edu/revistafae/article/view/288>.

VÁZQUEZ-INGELMO, A.; GARCÍA-HOLGADO, A.; GARCÍA-PEÑALVO, F. J. C4 model in a software engineering subject to ease the comprehension of uml and the software. p. 919–924, 2020.

VOLKWYN, T. S. et al. Learning to use cartesian coordinate systems to solve physics problems: The case of 'movability'. *European journal of physics*, IOP Publishing, v. 41, n. 4, p. 045701, 2020.