Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

# Desenvolvimento de Jogos Indie

## *Sistemas e Desafios em 'Sculpted Fate'*

André Gustavo Nakagomi Lopez, Johnny da Silva Lima, Lincoln Yuji de Oliveira

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Profº. Drº. Ricardo Nakamura

São Paulo
2023

# Agradecimentos

*"Para o homem é impossível, mas para Deus todas as coisas são possíveis"*
— Jesus Cristo

Primeiramente, agraço ao meu Senhor e Salvador Jesus Cristo, sem Ele, nada disso seria possível. Agradeço a minha família por sempre me apoiar, aos meus amigos que tive o privilégio de conhecer e fazer este trabalho, e ao nosso orientador, que nos auxiliou no processo.

Johnny Da Silva Lima

Quero agradecer ao meu Senhor Jesus Cristo por me dar motivação e vida para continuar nos estudos, e também pelas amizades que fiz, que me ajudaram a completar esse trabalho. Quero agradecer minha família pelo apoio e pela paciência que eles têm por mim. Também quero agradecer o nosso orientador por nos guiar e nos ajudar durante todo o processo.

André Gustavo

Gostaria de agradecer à minha família, aos meus amigos e todos àqueles que sempre me apoiaram e nunca desistiram de mim, apesar de todas as falhas e momentos difíceis. Além disso, quero prestar agradecimentos ao nosso orientador, Ricardo Nakamura, que nos guiou e deu liberdade para o projeto acontecer.

Lincoln Yuji

# Resumo

André Gustavo Nakagomi Lopez, Johnny da Silva Lima, Lincoln Yuji de Oliveira . **Desenvolvimento de Jogos Indie: *Sistemas e Desafios em 'Sculpted Fate'***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

Este trabalho busca documentar o processo de desenvolvimento de um jogo com sistemas complexos, robustos, interligáveis e reutilizáveis. Além disso, procura expor os desafios enfrentados durante o processo de criação e analisar o feedback após o lançamento do jogo. Utilizando a orientação a objetos para a construção dos sistemas e métodos ágeis para a organização do projeto, foi criado o jogo de ação e plataforma chamado 'Sculpted Fate', que implementa mecânicas tradicionais de jogos desse gênero. O produto final é um jogo que apresenta sistemas expansíveis e a maioria das mecânicas planejadas inicialmente. Também destaca os desafios encontrados, como a falta de artistas no grupo, e mostra o impacto das opiniões dos jogadores na qualidade do jogo.

**Palavras-chave:** Jogo Indie. Desenvolvimento de jogos. Desenvolvimento de sistemas.

# Abstract

André Gustavo Nakagomi Lopez, Johnny da Silva Lima, Lincoln Yuji de Oliveira . **Indie Game Development: *Systems and Challenges in 'Sculpted Fate'***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

This paper aims to document the development process of a game with complex, robust, interconnected, and reusable systems. It also aims to highlight the challenges faced during the creation process and analyze the feedback received after the game's launch. By employing object-oriented programming for system construction and agile methods for project organization, we developed the action-platformer game 'Sculpted Fate,' implementing traditional platform game mechanics. The final product features expandable systems and incorporates a significant portion of the initially planned mechanics, while also addressing challenges such as a shortage of artists in the group. Additionally, the study explores the impact of player opinions on the game's quality.

**Keywords:** Indie Game. Game Development. Systems development.

# Lista de abreviaturas

| | |
|---|---|
| BT | *Behavior Tree* |
| FOSS | *Free and Open-Source Software* |
| FSM | *Finite State Machine* |
| GDD | *Game Design Document* |
| POO | *Programação Orientada a Objetos* |
| RPG | *Role-Playing Game* |

# Lista de figuras

# Lista de programas

# Sumário

# Introdução

Num cenário no qual os videogames desempenham um papel de destaque na cultura contemporânea, influenciando amplamente a vida cotidiana, os desenvolvedores independentes, ou "indies", emergem como agentes inovadores, desafiando convenções e proporcionando experiências únicas aos jogadores. Este projeto se insere nesse contexto, explorando os sistemas fundamentais que constituem a espinha dorsal dos jogos indie, com um enfoque específico na criação de um jogo de ação e plataforma - o "Sculpted Fate".

A motivação subjacente a este desenvolvimento reconhece que, para os desenvolvedores independentes, a criação de jogos transcende a expressão artística, demandando a construção de sistemas complexos que definem a essência do jogo. A jornada em direção à realização do "Sculpted Fate" não apenas abordou a necessidade de uma expressão artística autêntica, mas também a habilidade de traduzir conceitos abstratos em uma experiência interativa coesa.

Os obstáculos enfrentados durante o desenvolvimento desse jogo, desde limitações artísticas até restrições temporais, serviram como catalisadores para uma compreensão mais profunda da natureza do desenvolvimento de jogos. Adotando metodologias ágeis, o processo de desenvolvimento foi dividido em ciclos iterativos, permitindo adaptações contínuas às mudanças de requisitos e feedbacks recorrentes da equipe. A flexibilidade proporcionada pelos métodos ágeis foi crucial para superar desafios dinâmicos, garantindo uma abordagem adaptativa às complexidades do projeto.

Ao centrar a atenção nos sistemas essenciais, que abrangem programação orientada a objetos, design, arte e trilha sonora, este trabalho desvendará não apenas os desafios técnicos, mas também as dificuldades criativas que moldam a identidade de um jogo independente. A programação orientada a objetos, em particular, foi essencial para criar uma arquitetura de código modular e expansível, permitindo uma estrutura clara e eficiente para o desenvolvimento dos sistemas do jogo.

Ao longo dessa jornada, exploraremos a interseção entre a criatividade artística e a engenharia de sistemas, destacando como cada elemento contribui para a experiência do jogador no contexto específico de um jogo de ação e plataforma. Desde as primeiras ideias conceituais até os ajustes finais influenciados pelos feedbacks da comunidade.

# Capítulo 1

# Planejamento do projeto

Este capítulo contém os passos realizados para concretizar as ideias do jogo a serem implementadas. Inclui as técnicas utilizadas, como *Brainstorming*, qual o escopo decidido e os princípios de design utilizados.

O capítulo também explica as ferramentas utilizadas, tanto para a programação como para os quesitos organizacionais.

## 1.1 Brainstorming

Inicialmente, foi pensado em como o jogo deveria ser em termos de gênero e jogabilidade. A ideia inicial foi fazer um jogo do gênero Plataforma 2D, a qual foi mantida.

Esse período foi utilizado para pensar no estilo de jogo desejado, que foi um jogo com: ataques com espada, *crossbow*, *dash*, pulo, agachamento, etc. Eventualmente foi necessário escrever um Game Design Document (GDD) para formalizar, documentar todas as decisões de projeto e definir um escopo mínimo de funcionalidades.

## 1.2 Escopo Inicial

Para o escopo inicial, foi decidido a realização de um jogo de ação e plataforma, baseado em jogos como *Super Mario World* [1], *Castlevania Rondo of Blood* [2], *Cuphead*[3] e *Katana Zero*[4]. Cada um desses jogos serviu como inspiração para os principais aspectos e mecânicas do jogo.

[1] https://pt.wikipedia.org/wiki/Super_Mario_World

[2] https://pt.wikipedia.org/wiki/Castlevania:_Rondo_of_Blood

[3] https://cupheadgame.com

[4] https://www.katanazero.com


**Figura 1.1:** *Exemplo de action-platformer: Super Mario World. Disponível em: Wikipedia.* Super Mario World. *2023. URL: https://en.wikipedia.org/wiki/Super_Mario_World (acesso em 01/12/2023)*

O estilo das fases seria baseado em *Cuphead* e *Castlevania Rondo of Blood*, com fases mais horizontais e lineares, sem caminhos alternativos e poucas/nenhuma entrada secreta. Com desafios de parkour e plataformas como plataformas fixas, móveis, espinhos que dão dano por segundo e buracos, que caso o jogador caia morrerá instantaneamente.

O combate seria baseado em *Hollow Knight*, com o jogador possuindo quatro pontos de vida, uma barra de energia para a utilização de *PowerUps*, que seriam espalhados pela fase, duas armas: uma espada como principal e uma *crossbow* como secundária, com cada ataque com a espada restaurando um pouco de energia. Além disso, teria ataques direcionais (Cima, baixo, esquerda e direita) e uma *hitbox* de ataque pequena e precisa. A principal mecânica é um *hit-kill*, que quando uma determinada área de um inimigo for atingida, ele morre instantaneamente, independente da sua quantidade de vida.


**Figura 1.2:** *Ilustração da hitbox do ataque: Castlevania Rondo of Blood. Imagem elaborada pelos autores*

**Figura 1.3:** *Ilustração do ponto fraco. Imagem elaborada pelos autores*

Cada ataque bem-sucedido contra um inimigo concede um impulso na energia do jogador. Esse aumento de energia pode ser utilizado para ativar power-ups coletados ao longo da fase. Cada power-up é exclusivo para a fase em que é coletado, substituindo-se por outro ao ser adquirido. Vale ressaltar que os efeitos proporcionados por um power-up específico são restritos à fase em que foi coletado, não sendo transferíveis para outras fases do jogo.

A movimentação não conta com aceleração, sendo possível se mover, andar, correr para a direita e para a esquerda, pular, com a mecânica de pular mais alto conforme se segura o botão de pulo e uma esquiva, que deixa o jogador invulnerável por certo período e o movimenta em alta velocidade por um curto espaço.

O mapa do jogo seria como o do jogo *Super Mario World* começando com apenas uma fase desbloqueada. Conforme o jogador completa as fases, novos níveis são desbloqueados ao longo do mapa. O nível desbloqueado sempre está conectado com o anterior. O mapa representa visual e geograficamente o conteúdo encontrado dentro das fases. Além de manter coesão visual e contextual com os outros níveis, que seriam cinco.

Os efeitos sonoros seriam implementados para cada ação do jogador, como tomar dano, ataque, pulo, cair, andar e morrer, assim como os inimigos. Cada fase teria uma música própria, assim como cada chefe.

Para a arte e animação do jogo, o estilo decidido foi de 64 bits, com animações para cada ação do jogador e inimigos: ataques, movimentação, dano e morte.

Cada inimigos, apresenta características como: uma quantidade de vida, ataques em seus pontos não críticos tiram uma quantidade de dano, ataques ao ponto fraco de inimigos não-chefes dão *hit-kill*, ou sejam, matam em um ataque. Para os inimigos foram decididos três tipos:

- Simples: No mínimo cinco diferentes, com movimentação básica, limitada e independente das informações ao redor. Exemplo: Goomba e Koopa dos jogos da franquia *Mario*.

- Complexos: No mínimo cinco inimigos complexos distintos, com ataques e movimentação diferentes (utilizando *state machines*). Possuem um campo de visão e um estado *idle*, que, enquanto o jogador não estiver no campo de visão, ele continua no estado *idle*. Se algum inimigo morrer na tela, os outros inimigos complexos saem do estado idle. Exemplo: Inimigo com machado Castlevania Rondo of Blood [5].
- Chefes: No mínimo três. Basicamente são inimigos complexos, com mais estados e ataques diferentes. Para eles, ataques no ponto fraco tiram muita vida, além de os atordoar. Cada ponto fraco de um chefe só pode ser atingido uma vez.

Além disso, o jogo apresentaria: menu principal, configurações (Som, tamanho da tela, fullscreen), Tela de Pause, Sistema de save (salvar fases liberadas e salvar checkpoint) e suporte para teclado e *joystick*.

### 1.2.1 Design do Tutorial

O primeiro nível assume o papel de tutorial do jogo. Começando com uma *cutscene*, onde a jogabilidade se inicia após uma estátua ser atingida por um raio de luz, quebrando-se e dando início à exploração.

Localizado em um templo situado no coração de uma floresta densa, o jogador se depara com uma sala repleta de escombros e uma saída bloqueada, exigindo habilidades de parkour para superar os obstáculos e aprender as mecânicas de movimentação.

Ao avançar para o bosque, com poucas árvores e vegetação rasteira, o jogador enfrenta desafios iniciais para se habituar à movimentação e se depara com os primeiros inimigos, introduzindo as noções básicas de combate. Posteriormente, ao adentrar mais afundo na densa floresta, onde surgem mais plataformas e inimigos, instruindo o jogador em movimentos de ataque variados e técnicas específicas, como o *Bounce*, essencial para acessar um *power-up* escondido.

[5] https://youtu.be/nw1oexAUjlQ?si=flflD8JeHopwjwW9&t=68


**Figura 1.4:** *Primeira parte do tutorial. Imagem elaborada pelos autores*

Após essa etapa, o jogador atinge um checkpoint antes de enfrentar a sala do chefe. O confronto com o chefe ensina o jogador sobre esquiva e ataque nos pontos fracos, apresentando mecânicas específicas para lidar com os movimentos e ataques do chefe. Por exemplo, a necessidade de executar o *Bounce* de forma eficaz para aproveitar a altura do ponto fraco do chefe, localizado em sua nuca, oferece um desafio com recompensa arriscada.


**Figura 1.5:** *Segunda parte do tutorial. Imagem elaborada pelos autores*

## 1.3 Ferramentas

Foram escolhidas ferramentas já muito consolidadas e familiares para facilitar o desenvolvimento do projeto e evitar dificuldades desnecessárias logo no começo.

### 1.3.1 Git

Foi decidido utilizar o Git[6] para controlar o versionamento do projeto. Seria inviável para o grupo desenvolver esse jogo sem uma ferramenta para monitorar as mudanças que cada um fez e resolver possíveis conflitos de versões.

Com o Git, é possível para os integrantes trabalharem em paralelo, em uma *branch* separada. Também é possível separar as contribuições em *commits* pequenos, então é possível obter uma linha de tempo das funcionalidades implementadas, também sendo possível desfazer certas alterações com facilidade se algum bug for introduzido no código. E por fim, é possível reconciliar as alterações feitas separadamente em uma *branch* principal.

Em particular, a plataforma Github foi escolhida para criar o repositório, Pois os membros possuem mais familiaridade com ela

### 1.3.2 Jira

Optou-se por usar um Kanban[7] para registrar e distribuir as tarefas entre os membros. Um quadro Kanban não só facilita essa organização, mas também facilmente nos ajuda a definir prioridades, dependências e condição geral do projeto.

A plataforma online Jira[8] foi escolhida. Ela não somente possui um Kanban digital, mas também outras interfaces que ajudam a definir fases de desenvolvimento e datas de entrega. Com a plataforma também é possível organizar uma linha do tempo de entregas, como é possível observar a seguir:

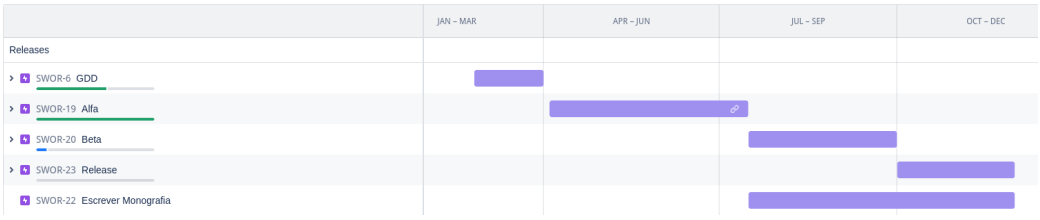



**Figura 1.6:** *Linha de tempo organizada no projeto. Imagem elaborada pelos autores*

Explicando mais detalhadamente o que é um Kanban: trata-se de um sistema de gestão originado no Japão e foi inicialmente desenvolvido pela Toyota nas décadas de 1940 e 1950. O sistema Kanban foi criado para melhorar a eficiência, controle de processos e otimização de fluxos de trabalho, algo com que muitas fábricas e empresas sofreram na época.

---

[6] https://pt.wikipedia.org/wiki/Git

[7] https://pt.wikipedia.org/wiki/Kanban

[8] https://swordhead.atlassian.net/jira/software/c/projects/SWOR/boards/1

A ideia de um Kanban é proporcionar visibilidade e controle sobre o fluxo de trabalho, permitindo que as equipes identifiquem gargalos, gerenciem o trabalho em progresso e otimizem a entrega de produtos ou serviços. O funcionamento do Kanban pode ser entendido com a seguinte lista:

- **Quadro Kanban:** o Kanban se utiliza de um quadro (físico ou digital) que contém colunas e cartões. Cada coluna representa uma etapa no processo, e os cartões representam tarefas ou unidades de trabalho.
- **Distribuição de tarefas:** a fim de evitar sobrecargas e gargalos em alguma parte do desenvolvimento, cada coluna do quadro tem um limite de WIP (Work in Progress), definindo um número máximo de cartões simultâneos.
- **Fluxo de Trabalho:** os cartões são movidos de uma coluna para outra à medida que o processo de desenvolvimento avança. Todos os membros da equipe podem ver facilmente em qual etapa cada tarefa está e como o desenvolvimento flui ao longo do tempo.
- **Priorização e atribuição de tarefas:** à medida que novas tarefas, ou cartões, entram no sistema, elas são priorizadas e atribuídas aos membros da equipe. Isso garante que o desenvolvimento ocorra de maneira ordenada e eficiente.

O Kanban, em particular, foi adotado pelo movimento de Métodos Ágeis e é largamente utilizado no mercado de desenvolvimento de software. A seguir é possível observar um exemplo do Kanban *board* desenvolvido para o projeto, utilizando a plataforma Jira:

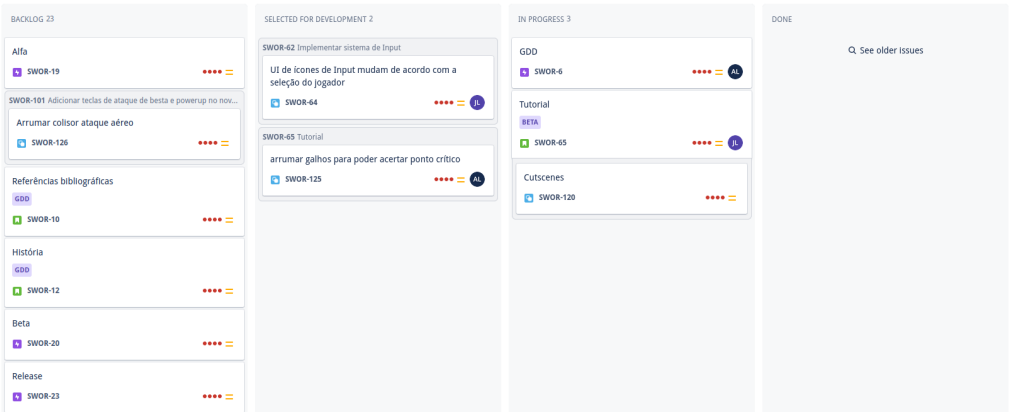



**Figura 1.7:** *Exemplo KanbanBoard. Imagem elaborada pelos autores*

### 1.3.3 Discord

O meio de comunicação utilizado foi o *Discord* [9]. Ele possui todas as funcionalidades básicas de outras ferramentas como Google Meet (chat, canal de áudio, compartilhamento de tela, etc.), mas nele também podemos criar um servidor customizável com múltiplos canais para separar diferentes informações e pautas das reuniões.

## 1.4 Game Engine

O framework de desenvolvimento escolhido foi Unity, um motor de jogos. (Unity, 2022). Trata-se de uma *engine* bem robusta e extremamente documentada, além de ser muito utilizada por pequenos e grandes estúdios de desenvolvimento há anos.

Todos os membros do grupo já tiveram experiências com a ferramenta, o que facilitou a decisão. Além disso, Hollow Knight, o jogo que mais inspirou o projeto, foi feito na Unity. Essa combinação de conveniências fez com que a escolha dessa ferramenta para o projeto fosse unânime.

### 1.4.1 Característica principais da Unity

Uma lista de alguns pontos importantes a respeito da Unity é:

- **Multi-Plataforma:** jogos feitos na Unity podem ser exportados para uma ampla variedade de plataformas, incluindo Windows, macOS, iOS, Android, consoles, web, etc. Isso facilita para os desenvolvedores a criação de jogos e aplicativos que podem ser executados em várias plataformas.
- **Gráficos:** a *engine* oferece gráficos de alta qualidade, incluindo *shaders* e outros diversos recursos avançados de renderização, o que faz dela uma boa opção para o desenvolvimento de jogos com visuais mais polidos ou estilizados. Em particular, a Unity possui diversas APIs já integradas em seu ambiente de desenvolvimento, incluindo Direct3D e OpenGL.
- **Recursos de desenvolvimento:** a Unity fornece um conjunto amplo de ferramentas de desenvolvimento, incluindo um editor visual para a criação de cenas e prefabs, diversas mecânicas e processos de física, suporte para animações e muito mais. Isso permite que os desenvolvedores criem jogos complexos rapidamente.
- **Linguagem de programação:** a principal linguagem utilizada no desenvolvimento com a Unity é o C# (VVA, 2023). Algumas características dessa linguagem são: orientada a objetos, tipada, coletor de lixo para fazer gerenciamento de memória, multiplataforma, nativamente integrada com diversos arcabouços de desenvolvimento (NET Framework, por exemplo), consolidada com um amplo e rico ecossistema de bibliotecas, etc.
- **Asset Store:** a Unity possui uma loja de extensões (Asset Store) que oferece uma ampla variedade de recursos, como modelos 3D, texturas, scripts e muito mais, que

[9] https://discord.com/

podem ser incorporados aos projetos. Boa parte desses recursos pode ser adquiridos gratuitamente, deixando o motor de jogos mais acessível para programadores iniciantes ou equipes com menos financiamento em seus projetos.

### 1.4.2 Interface da Unity

Tendo em mente que aprender a usar a interface da Unity é o primeiro desafio que os desenvolvedores enfrentam, a seguir segue uma lista dos principais elementos da interface do motor de jogos e suas funções:

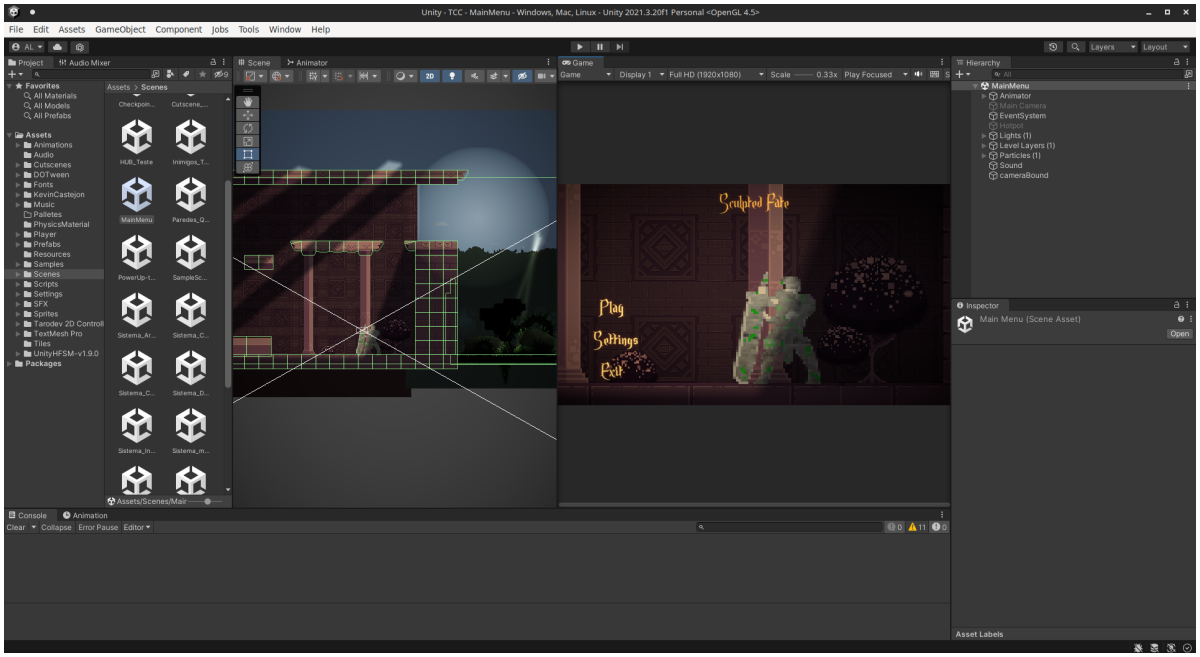



**Figura 1.8:** *Interface principal da Unity. Imagem elaborada pelos autores*

1. **Visualizador de Cenas:** o desenvolvedor consegue navegar e editar uma cena a partir dessa janela. Cenas podem conter diversos objetos e tudo pode ser renderizado com uma perspectiva 2D ou 3D, dependendo das configurações do projeto.

2. **Visualizador de Gameplay:** executa uma simulação usando os controles e gráficos finais do jogo, ou seja, a aparência e jogabilidade apresentadas são o produto final se o desenvolvedor decidir fazer a *build* do jogo nesse estado.

3. **Barra de Ferramentas:** o usuário pode acessar sua conta da Unity a partir dessa barra. Além disso, é possível editar o *layout* do editor e acessar todas as configurações do projeto por meio de menus localizados nesse elemento.

4. **Hierarquia de Objetos:** essa janela apresenta uma representação hierárquica de todo GameObject em uma cena. Um GameObject é o tipo de objeto fundamental da Unity e todas as suas funcionalidades são definidas a partir de diversos componentes associados ao mesmo. Nessa hierarquia, os objetos filhos dependem da execução do objeto pai, ou seja, se um objeto é desativado, então todos os seus filhos não serão executados.

5. **Inspetor de Objetos:** permite o usuário visualizar e editar qualquer propriedade de um GameObject. Através dessa janela, o desenvolvedor consegue associar diversos componentes ao objeto (colisores, câmeras, scripts, etc.) para definir todo o comportamento desse GameObject uma vez que a cena é inicializada.
6. **Janela do Projeto:** apresenta todos os arquivos, bibliotecas e *assets* disponíveis dentro do projeto que podem ser utilizados no desenvolvimento do jogo. Qualquer *asset* importado, seja da Asset Store ou externamente, irá aparecer nessa janela.
7. **Barra de Status:** fornece notificações a respeito de vários processos da Unity e permite rápido acesso às ferramentas relacionadas aos avisos, como, por exemplo, o *Console* ou o Gerenciador de Projeto, que também pode ser acessado a partir da barra de ferramentas.

### 1.4.3 Classe Monobehavior

Monobehavior é a classe mais importante que um desenvolvedor em Unity deve conhecer, tratando-se de uma classe especial com uma série de funções de ciclo de vida. Todos os métodos dessa classe são funções chamadas uma vez por *game-loop* ou via mensagens.

Os métodos mais importantes de um Monobehavior são:

- **Start:** executado no mesmo *frame* em que o script é ativado, antes de executar qualquer um dos métodos Update.
- **Update:** executado uma vez a cada *frame*.
- **FixedUpdate:** diferente do Update convencional, esse método é chamado em intervalos fixos de aproximadamente 0,02 segundos por padrão e independe da taxa de quadros do jogo. Seu propósito geral é permitir que o programador modifique elementos de física no jogo com mais precisão.
- **LateUpdate:** assim como o Update normal, esse método também é executado uma vez a cada *frame*. O LateUpdate é chamado após todos os Updates de todos os Monobehaviors ativados terminarem. Muito útil para definir uma ordem de comportamento entre diferentes Monobehaviors do jogo.
- **Destroy:** coloca um GameObject em uma fila especial para ser destruído. Ao final de todo *frame*, um sistema do jogo varre essa fila e apaga todos esses objetos da memória e todas as referências que apontavam para eles passam a ser nulas, efetivamente deletando-os da cena atual.

Uma lista com todos os métodos de um Monobehavior e quando são chamados pode ser encontrada na documentação da Unity [10].

[10] https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

### 1.4.4 Sistema de física

O sistema de física da Unity é altamente robusto, preciso e otimizado. De fato, um dos grandes atrativos da *engine*. Qualquer informação importante a respeito desse sistema pode ser encontrada na documentação, mas aqui segue uma lista com os principais elementos desse sistema e suas utilidades:

- **Rigidbody (Corpo Rígido):** simula um corpo no espaço e afeta o comportamento de um objeto com base em princípios de física básica. Possui propriedades como massa e gravidade, que determinam resistência de movimento, inércia, força gravitacional aplicada, impacto em colisões, etc. Forças e impulsos externos também podem ser aplicados nesses corpos, possibilitando a criação de simulações bem sofisticadas.
- **Colliders (Colisores):** esses elementos definem a forma física de um objeto. Diferente de um Rigidbody, um colisor não tem efeitos físicos próprios, mas são utilizados pela Unity para simular interações físicas entre diferentes objetos. É possível observar os colisores no editor da Unity através das linhas verdes indicadas, como é possível observar na imagem a seguir:

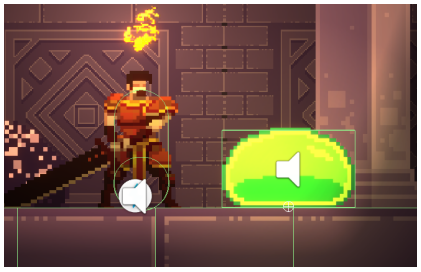


**Figura 1.9:** *Exemplo dos indicadores visuais de colisores. Imagem elaborada pelos autores*

- **Detecção de Colisão:** colisões são detectadas automaticamente entre objetos com colisores. Toda vez que uma colisão é detectada, uma mensagem é enviada e métodos especiais podem ser acionados para lidar com a colisão. Por exemplo, existem métodos do Monobehavior chamados toda vez que o objeto colide com alguma coisa.
- **Camadas e máscaras de colisão:** permitem que o desenvolvedor defina quais objetos podem colidir com outros. Alguns jogos cooperativos, por exemplo, não permitem que um jogador consiga causar dano aos outros. Isso pode ser facilmente implementado criando uma camada para todos os jogadores e garantindo que a máscara dos ataques de um jogador não consigam detectar colisão com objetos dessa camada. No projeto, por exemplo, foi separado os colisores de física de inimigos (*Enemy*) dos colisores de dano do inimigo (*EnemyDamage*), e foi feito que o jogador só possa colidir com os colisores de dano de inimigos, para permitir que o jogador atravesse inimigos, como se pode observar a seguir:

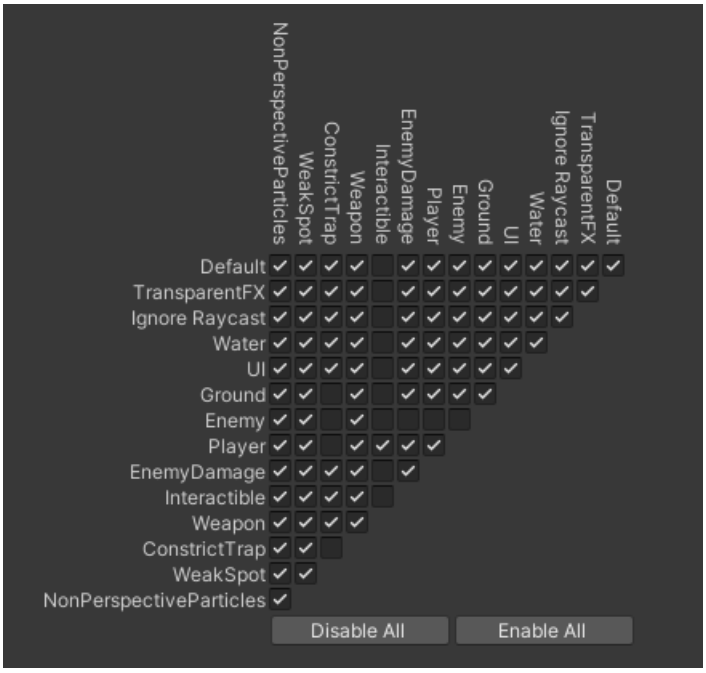



**Figura 1.10:** *Camadas de colisão. Imagem elaborada pelos autores*

- **Raycasting:** útil para detectar colisões em uma linha. Geralmente utilizado para calcular colisões em pontos futuros, como projéteis, ou para detectar outros objetos a uma certa distância de sua origem.
- **Ambiente 2D:** apesar de todos os elementos citados anteriormente serem implementados em um sistema 3D, a Unity também oferece um ambiente de desenvolvimento 2D para a criação desse gênero de jogo. Devido à ausência de uma terceira dimensão, muito cálculos podem ser otimizados. Em geral, os elementos de física 2D funcionam de maneira semelhante aos de física 3D, mas possuem essas otimizações adicionais.

### 1.4.5 Sistema de animação

Como já foi dito anteriormente, a Unity possui diversas ferramentas que permitem os desenvolvedores criarem animações complexas facilmente. Esta seção contém uma breve introdução aos principais elementos do sistema de animação da Unity.

- **Animator:** responsável por gerenciar o comportamento de uma animação, como as transições e tempo de execuções.
- **Animator Controller:** trata-se de um arquivo especial que define como um Animator irá gerenciar as animações. Ele contém uma máquina de estados, onde cada estado representa uma animação e cada aresta representa uma transição. As transições podem ser acionadas por diversas condições, como tempo de animação, mensagens ou detecção de eventos. Essa ferramenta foi utilizada para construir a lógica de animação do jogador e dos inimigos do projeto, é possível observar um exemplo de *animator* do inimigo chefe do tutorial a seguir:

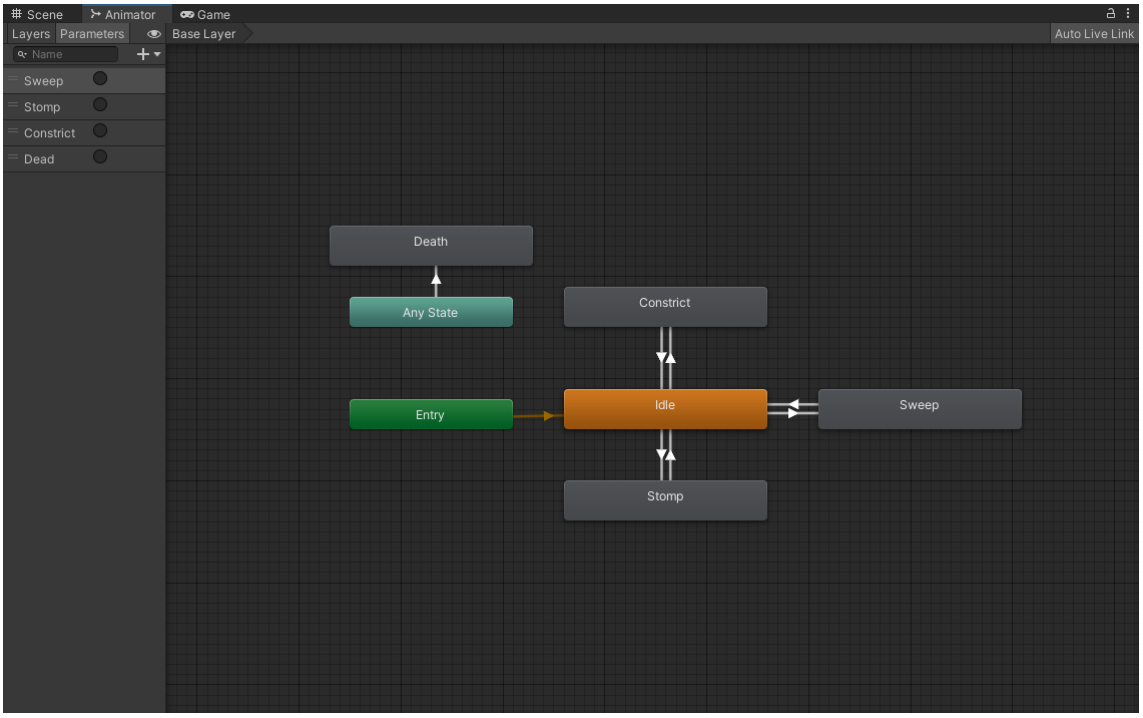



**Figura 1.11:** *Exemplo Animator inimigo chefe. Imagem elaborada pelos autores*

- **Parâmetros do Animator:** são variáveis que podem ser alteradas pela interface da *engine* ou controladas via código com scripts. Elas podem ser usadas para acelerar, desacelerar, forçar ou cancelar animações de um Animator entre outras coisas.

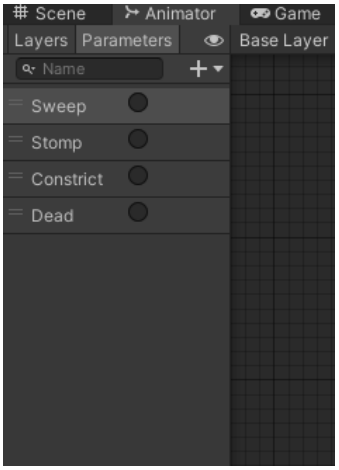



**Figura 1.12:** *Exemplo: parâmetros do Animator do inimigo chefe. Imagem elaborada pelos autores*

- **Eventos de animação:** é possível adicionar eventos em uma animação na Unity. Por exemplo, quando a animação de ataque é tocada podemos criar um evento que aciona a *hitbox* do ataque em algum momento e desliga essa *hitbox* em outro evento ao final da animação. Esses eventos são muito úteis para o desenvolvedor sincronizar outros elementos e comportamentos do jogo com as animações de maneira extremamente precisa. É possível observar o evento que liga o *hitbox* de ataque, no ataque normal do jogador na figura a seguir:

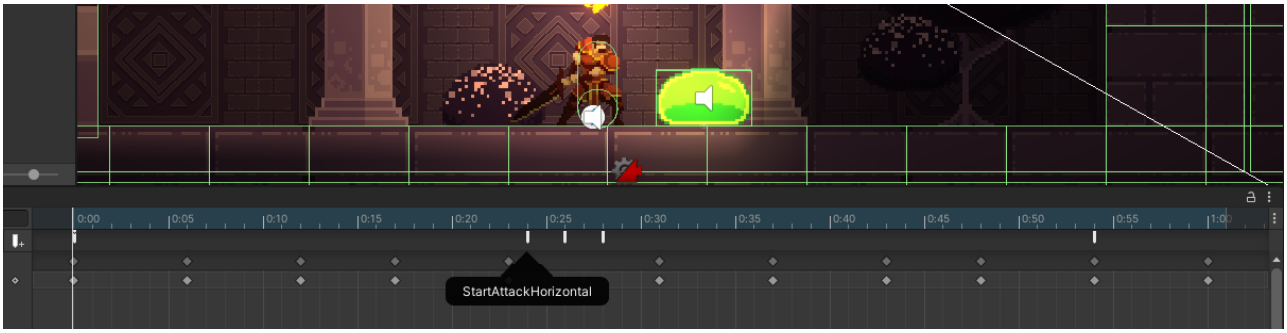



**Figura 1.13:** *Exemplo evento de animação do ataque do jogador. Imagem elaborada pelos autores*

Mais informações sobre o sistema de animação da Unity podem ser encontradas na documentação oficial.

### 1.4.6 Componentes

A classe Component se refere a qualquer elemento que pode ser associado a um GameObject. A classe Monobehavior, por exemplo, é um Component e, portanto, um script da mesma deve estar associado a um GameObject para ser ativado e executado. Utilizando esse padrão de projeto, é possível criar objetos muito complexos com diversos comportamentos associados. Em particular, alguns elementos já citados anteriormente como o Animator, Rigidbody e Colliders são componentes.

Componentes podem associados a um GameObject de duas formas:

- **Script:** GameObjects possuem um método AddComponent, que associa um novo componente à estrutura do objeto.
- **Inspetor:** novos componentes podem ser adicionados à estrutura de um objeto via inspetor (Figura 1.14), como explicado no capítulo anterior explicando a interface principal da Unity.

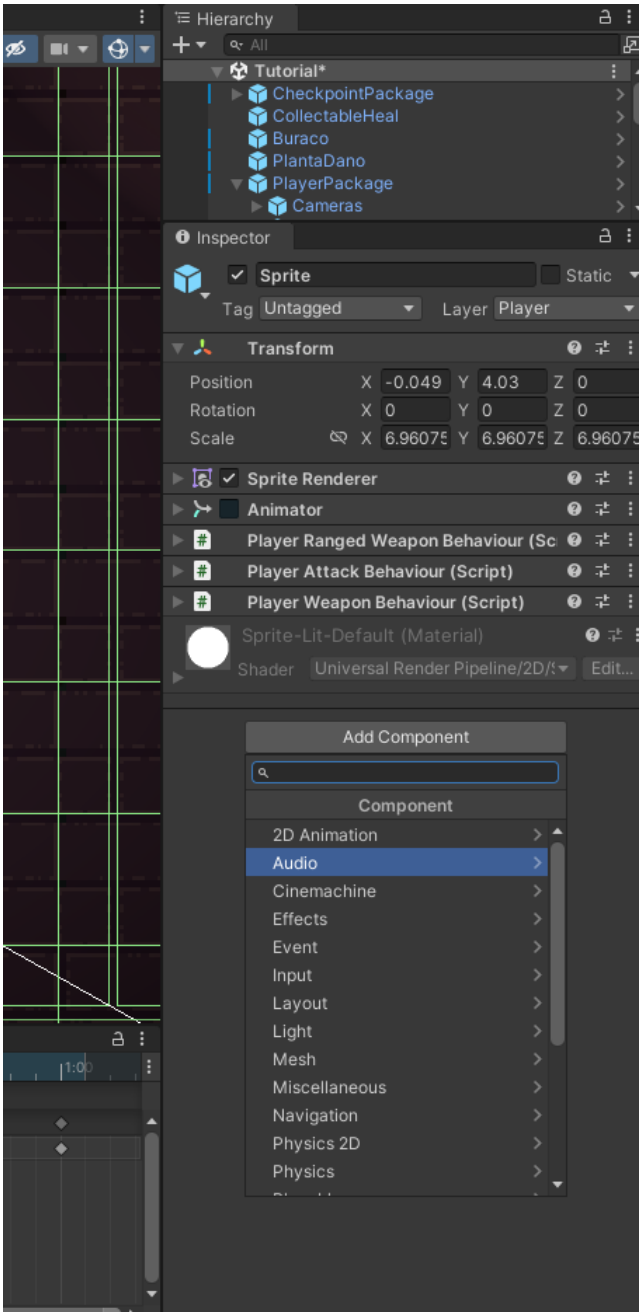



**Figura 1.14:** *Exemplo adicionar componente por* Inspector. *Imagem elaborada pelos autores*

Outro ponto importante a respeito de componentes na Unity é que qualquer atributo público da classe aparece no inspetor. No entanto, o desenvolvedor pode desejar que um atributo apareça no inspetor, mas não seja público por questões de encapsulamento de código. Para isso, a Unity oferece uma anotação que, independente do nível de proteção, expõe um atributo para o inspetor da *engine*, o **SerializeField**.

### 1.4.7 Prefabs de objetos

Um Prefab é um objeto ou uma hierarquia de objetos reutilizável em múltiplas cenas. Trata-se de um modelo de objeto definido pelo desenvolvedor que pode ser instanciado diversas vezes em qualquer cena do jogo.

O grande benefício de se usar Prefabs está na sua reutilização. Várias instâncias de um mesmo Prefab podem ser criadas em uma cena ou em cenas diferentes. Quando o algum é alterado, todas as suas instâncias são automaticamente atualizadas, facilitando o processo de criação e manutenção de objetos do jogo.

Além disso, novos Prefabs podem ser criados a partir de outros e então alguns elementos podem ser adicionados ou alterados, assim como é possível remover ou adicionar objetos ao novo Prefab, permitindo o desenvolvedor rapidamente criar uma variação do antigo.

Muitas características de Prefabs são semelhantes às dos GameObjects. Por exemplo, eles podem ser instanciados via código e podem ser organizado de maneira hierárquica, ou seja, é possível criar um Prefab composto de outros através de uma organização hierárquica. Essa forma de organizar o projeto permite que a arquitetura do jogo seja altamente modulável e flexível, melhorando a qualidade e manutenibilidade do projeto.

### 1.4.8 Bibliotecas adicionais

Para o projeto, foram utilizadas duas bibliotecas adicionais que não estão presentes na versão básica da Unity, isto é, ferramentas desenvolvidas por outros desenvolvedores para a game *engine*:

- **Ultimate 2D Controller:** trata-se de uma biblioteca desenvolvida pelo Tarodev [11] com diversas mecânicas de movimentação 2D já implementadas e múltiplas cenas de teste com plataformas para testar todas as diferenças mecânicas. Decidimos utilizar uma ferramenta já pronta para essa tarefa a fim de economizar tempo, visto que jogos de plataforma 2D são bem tradicionais e não há necessidade de reinventar a roda para implementar mecânicas de nível tão baixo. A biblioteca inclui *features* como pulo com altura variável, *buffering* de pulo, *dash*, escalada e muito mais. O funcionamento dessas mecânicas será explicado em detalhes no capítulo de **Fundamentação Teórica**.
- **DOTween:** essa é uma biblioteca com diversas funções de interpolação de valores entre *game-loops*. Muitas vezes o desenvolvedor quer que um determinado elemento do jogo seja modificado, porém, não instantaneamente. O DOTween oferece várias funções que criam co-rotinas e alteram o valor das variáveis se baseando em curvas já pré-definidas pela biblioteca. Essas curvas podem ser lineares, exponenciais, senoides, etc. Uma ferramenta dessas é bem útil para fazer animações e criar efeitos de elasticidade, impressão de *bouncing* ou vibração. Programadores experientes certamente poderiam escrever códigos do zero a fim de replicar esse comportamento. Porém, pelo mesmo motivo do uso da biblioteca do Tarodev, não vimos necessidade em ter o trabalho de criar um sistema possivelmente menos preciso ou eficiente quando

[11] https://www.youtube.com/c/Tarodev

comparado ao de uma biblioteca já existente, exaustivamente testado, refatorado e otimizado.

- **FluidBT:** oferece uma estrutura de dados chamada Árvore de Comportamento (*Behavior Tree*), comumente utilizada para programar IA (Inteligência Artificial) em jogos. Inimigos Chefes geralmente requerem comportamentos mais complexos, devem ser capazes de tomar algumas decisões ao longo do combate e poder diversas fases. Árvores de Comportamento são muito úteis para criar esses tipos de entidades em jogos e decidimos usar essa biblioteca por estar bem documentada e ser FOSS *free open-source software*. Essa estrutura de dados é explicada detalhadamente em capítulos futuros.
- **Hierarchical Finite State Machine (HFSM):** alguns inimigos possuem comportamentos mais simples e não precisam ser muito inteligentes, nesses casos usar Máquinas de Estado é uma boa opção gerando códigos mais limpos e fáceis de entender. Nesse tipo de estrutura, temos mais liberdade para mudar de qualquer estado para outros sem muitas restrições. Isso será explicado com mais detalhes em capítulos seguintes. Essa biblioteca oferece uma API bem simples para criar máquinas de estado rapidamente, e facilmente executá-las durante o game-loop.
- **Cinemachine:** jogos de plataforma 2D muitas vezes irão requerer um sistema de câmeras quando o nível inteiro não é exibido na tela. Apesar da Unity possuir um objeto de câmera nativo, criar comportamentos como transições, *clamp boundaries*, *shaking* requer mais tempo de desenvolvimento, envolvendo implementação dessas funcionalidades e possíveis otimizações. Decidimos usar essa biblioteca por já ser muito conhecida entre desenvolvedores em Unity e ser bem rica em funcionalidades.
- **New Input System:** os jogos são interativos, e modo pelo qual eles podem interagir é através dos *inputs* do jogador. Por isso, a eficiência de um sistema dedicado a isso, impacta diretamente na jogabilidade e na experiência do usuário. O *New Input System* oferece maior consistência na interpretação das entradas, de uma forma mais simples, rápida, flexível e eficiente em comparação ao sistema legado. o *New Input System*, apresenta um sistema de entrada baseado em eventos, permitindo o mapeamento de diferentes dispositivos de entrada, como teclados, mouse, controladores e joysticks, de maneira intuitiva e consistente. Além disso, o sistema oferece funcionalidades avançadas, como entrada com base em ações, *rebinds* de teclas e suporte embutido para controles de toque e gestos.

## 1.5 Processo de desenvolvimento

O processo de desenvolvimento foi inicialmente dividido em: elaboração do game design document (GDD), desenvolvimento de todos os sistemas, elaboração dos níveis e arte, coleta de feedbacks e por fim, lançamento do jogo. A divisão do tempo das tarefas foi:

- Elaboração do GDD: 4 semanas.
- Desenvolvimento dos sistemas: 13 semanas.

- Elaboração dos níveis e arte: 10 semanas.
- Coleta de feedbacks: 4 semanas.
- Lançamento: 4 semanas.

O GDD (Game Design Document), foi usado para a organização das ideias para o jogo, como: análise geral do jogo (gênero, audiência alvo, plataformas de lançamento), história do jogo e personagens, *gameplay* (*overview* do jogo, experiência do jogador, diretrizes de gameplay, objetivos e recompensas, mecânicas, inimigos, níveis) e controles. Para mais informações, verificar Apêndice C - GDD.

Para o desenvolvimento dos sistemas do jogo foi separada a maior fatia de tempo, pois foram baseados na programação orientada a objetos, de modo que atendesse os princípios de herança, polimorfismo, encapsulamento e abstração. Assim, os códigos seriam limpos, apresentáveis, facilmente expansíveis e modularizados.

A elaboração dos níveis foi feita a partir dos conceitos decididos no GDD, como, por exemplo, o planejamento do tutorial. Para a construção dos níveis foram utilizadas as ferramentas da Unity, como *tilemaps*.

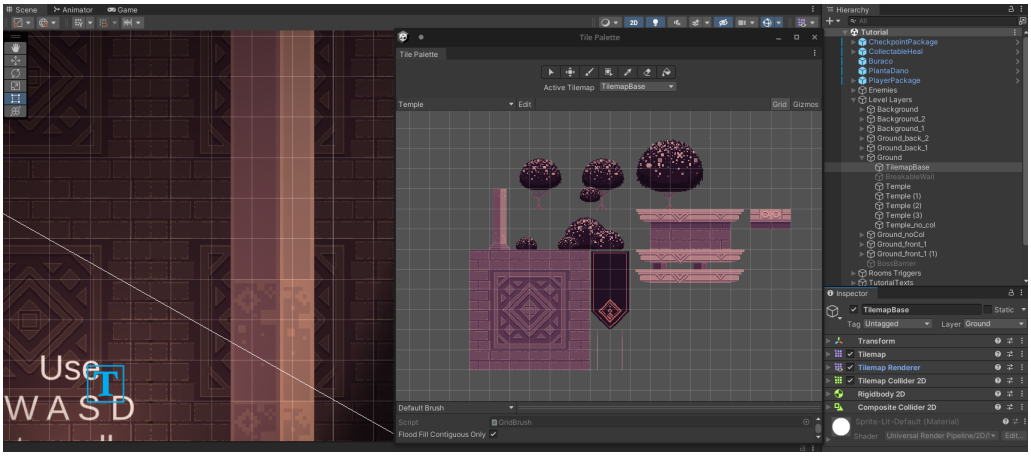



**Figura 1.15:** *Editor de tilemap na Unity. Imagem elaborada pelos autores*

Para a arte do jogo, diversos pacotes de licença gratuita foram baixados, sendo utilizados tanto na arte dos níveis, quanto no jogador, inimigos e menus. Para a escolha da arte, foi escolhido um estilo mais *retrô*, baseados nos antigos jogos de 64bits como Castlevania. Como diversos pacotes de artistas diferentes foram baixados, a coesão para que não houvesse uma grande diferença entre as artes foi importante. O estilo almejado é o estilo da figura abaixo:


**Figura 1.16:** *Estilo almejado para o projeto do jogo. Disponível em:* Jeremy Parish. How Can I Play It? Rondo of Blood and Symphony of the Night. *2017.* URL: *https://retronauts.com/article/608/how-can-i-play-it-rondo-of-blood-and-symphony-of-the-night (acesso em 01/12/2023)*

Para definir e distribuir tarefas para os membros do grupo foi utilizado um Kanban virtual, tendo como uma diferença o fato de não haver um *coach*, dado que apesar de fornecer algumas sugestões e feedbacks, o orientador não participou diretamente do desenvolvimento do código ou *assets* do jogo.

Outra prática dos métodos ágeis adotada foi a dos ciclos de desenvolvimento, mas sem prazos definidos para entregas, contanto que elas ficassem prontas a uma velocidade considerada razoável pelo grupo e fosse possível mostrar progresso para o orientador.

Foram feitas reuniões semanais pelo *Discord* para discutir o que foi possível, ou não, desenvolver durante o ciclo, explicar soluções ou dificuldades e fazer reuniões de desenvolvimento utilizando *mob-programming*.

As reuniões com o orientador foram feitas a cada duas semanas, onde o progresso de cada ciclo foi mostrado, e dúvidas, sugestões e problemas de programação, como *bugs* e mecânicas mais complexas, foram discutidas.

A coleta de feedback utilizou de dois recursos principais: um formulário e reviews do jogo no site itch.io [12], e foi realizada paralelamente com o lançamento do jogo. A página do jogo desenvolvido no site *itch.io* pode ser vista na figura abaixo:

[12] https://zubumafu67.itch.io/sculpted-fate

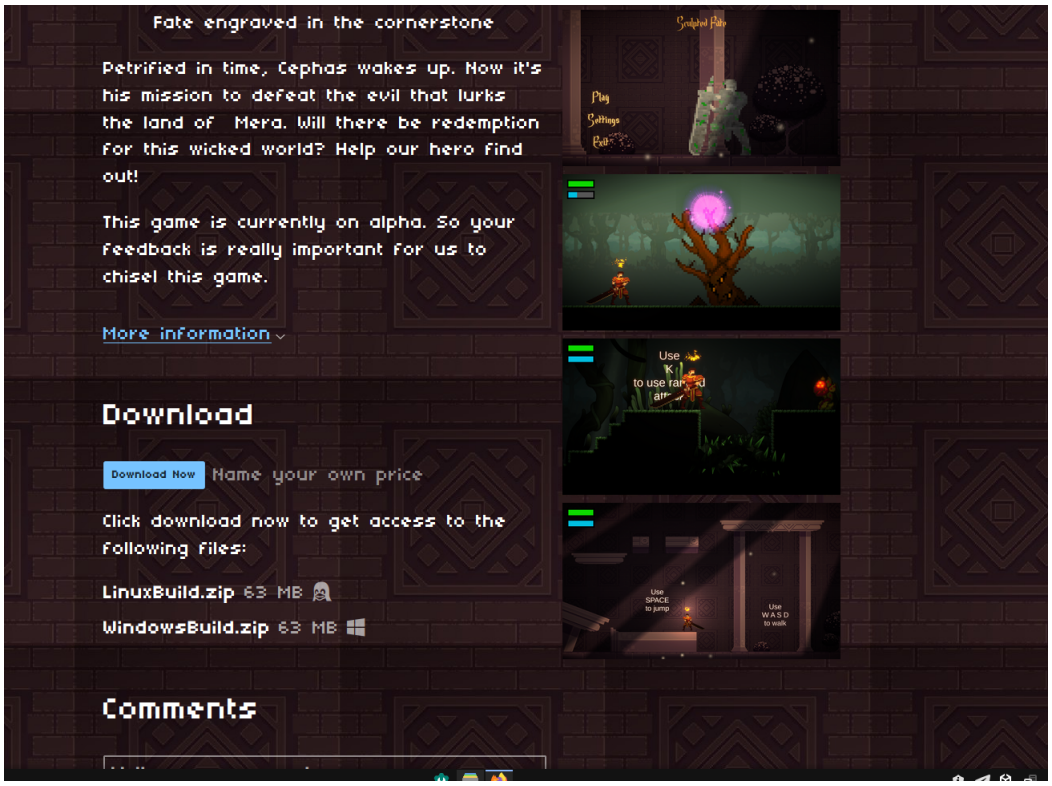



**Figura 1.17:** *Página do jogo no site itch.io. Imagem elaborada pelos autores*

# Capítulo 2

# Fundamentação Teórica

Neste capítulo são exploradas algumas das bases fundamentais que serviram como alicerce para a produção deste jogo eletrônico. O entendimento conjunto dos princípios de game-design e das técnicas de programação e organização é crucial para alcançar um jogo de qualidade, que seja apreciado pelos jogadores e eficazmente desenvolvido por sua equipe de colaboradores.

A importância de uma fundamentação bem estruturada reside na sua capacidade de orientar o game-design, garantindo que o jogo seja atraente e envolvente para o público-alvo. Ela fornece as diretrizes necessárias para criar mecânicas de jogabilidade sólidas, desafios envolventes e uma narrativa coerente. Além disso, a fundamentação teórica permite que os desenvolvedores compreendam as expectativas dos jogadores, levando a uma experiência de jogo mais gratificante.

Por outro lado, a sólida base teórica também é essencial para as técnicas de programação e organização do projeto. Ela ajuda a definir uma arquitetura de software eficiente e coesa, facilitando a implementação de sistemas complexos, como física de personagens, inteligência artificial e interações do jogador. Isso resulta em um desenvolvimento mais eficiente, organizado e facilmente expansível.

## 2.1 Action Platformer

Um *Action Platformer*, também conhecido como jogo de ação e plataforma, é um subgênero de jogos eletrônicos que combina elementos de ação e plataforma em um único jogo. O gênero de ação se concentra em combates em tempo real, ação rápida e interações intensas, abrangendo uma variedade de temas, desde tiroteios até lutas corpo a corpo. Por outro lado, o gênero de plataforma envolve a navegação por cenários cheios de obstáculos, com desafios de pulo, armadilhas e exploração do ambiente, frequentemente incorporando elementos de quebra-cabeças e coleta de itens. Ambos os gêneros oferecem experiências de jogo únicas e amplamente apreciadas.


**Figura 2.1:** *New Super Mario Bros. U Deluxe. Disponível em: NINTENDO.* New Super Mario Bros.™ U Deluxe. *2019.* URL: *https://www.nintendo.com/pt-br/store/products/new-super-mario-bros-u-deluxe-switch/ (acesso em 01/12/2023)*

Sendo assim, o objetivo principal de um jogo de ação e plataforma, é mover o jogador entre diferentes pontos do ambiente. Muitos jogos deste gênero também apresentam mecânicas de movimentação avançada, como: correr em paredes, se agarrar em cordas, diferentes tipos de pulos, etc.

### 2.1.1 Características

As principais características de um *Action Platformer* incluem:

- Movimentação Precisa: Os jogadores geralmente controlam um personagem ágil que pode realizar saltos precisos e manobras de plataforma para superar obstáculos e navegar pelos níveis.
- Inércia e Física: A aplicação de física realista, incluindo inércia, é vital para criar uma sensação de peso e realismo no movimento do personagem. Cálculos são utilizados para simular a velocidade, aceleração e fricção, resultando em movimentos mais naturais e dinâmicos, tornando a jogabilidade mais envolvente.
- Invencibilidade Temporária: A mecânica de invencibilidade temporária é implementada com cuidado para gerenciar o tempo após o personagem ser atingido por um inimigo. Durante esse período, o personagem é protegido contra ataques inimigos, proporcionando uma oportunidade estratégica para escapar de situações perigosas.
- Ambientes organizados em plataformas: O jogo é ambientado em níveis repletos de plataformas, escadas, abismos e outros elementos interativos que os jogadores devem explorar e superar.
- Combate: Muitos *Action Platformers* incorporam elementos de combate, permitindo que os jogadores enfrentem inimigos enquanto avançam pelos níveis. Isso pode

incluir o uso de armas, habilidades especiais ou simplesmente golpear inimigos fisicamente.

- Quebra-Cabeças: Alguns jogos desse gênero também incluem elementos de quebra-cabeças, nos quais os jogadores devem resolver desafios lógicos para progredir.
- Progressão de Níveis: Geralmente, os jogadores avançam por uma série de níveis, cada um com seus próprios desafios, inimigos e obstáculos. O jogo pode culminar em um chefe final desafiador.
- Dificuldade Crescente: A dificuldade tende a aumentar à medida que o jogador avança no jogo, oferecendo desafios cada vez mais complexos.

### 2.1.2 Técnicas

Existem certas técnicas utilizadas neste tipo de jogo para o jogador sentir que tem um maior controle sobre seu personagem, deixando a jogabilidade fluída e responsiva. As mais comuns são:

1. *Coyote Time*: Em jogos *Action Platformer*, o "*Coyote Time*"é essencial para proporcionar aos jogadores uma sensação de controle ágil. Quando o jogador deixa de pressionar o botão de salto enquanto está no ar ou à beira de uma plataforma, o "*Coyote Time*"concede um breve período (geralmente alguns milissegundos) durante o qual o jogo ainda registra o pulo. Isso permite que o jogador execute o pulo mesmo que tenha saído tecnicamente da plataforma, evitando a frustração de quedas inesperadas e mantendo a jogabilidade fluida. O *Coyote Time* pode ser visto aplicado no projeto na imagem a seguir:

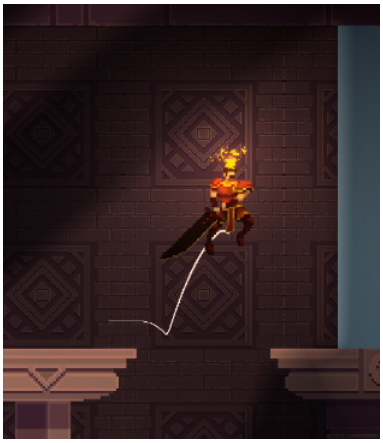


**Figura 2.2:** *Exemplo* Coyote Time. *Imagem elaborada pelos autores*

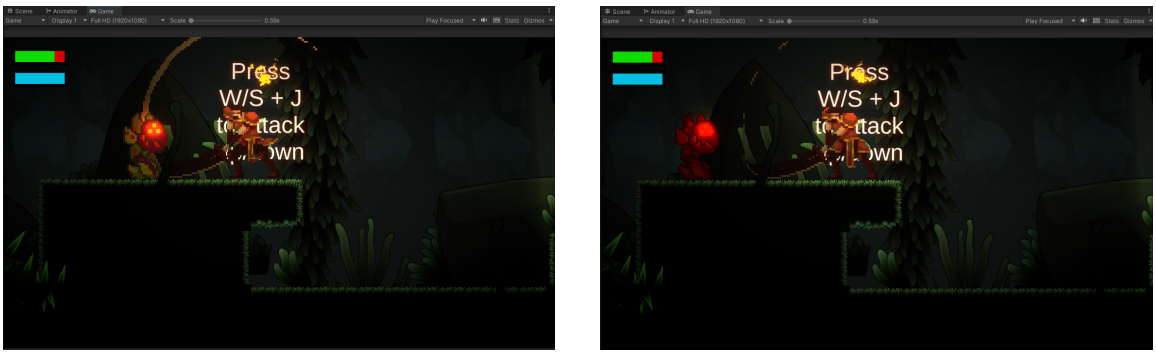
2. *Input Buffering*: A técnica de "*Input Buffering*"é utilizada para armazenar temporariamente os comandos de ação do jogador em uma fila. Essa fila é processada em uma ordem pré-definida, mesmo que os comandos tenham sido inseridos um pouco antes de serem necessários. Isso garante que os comandos do jogador sejam registrados e executados com precisão, independentemente do timing exato. Um exemplo de como isso é utilizado é o *Jump Buffering* onde o comando de pulo é pressionado antes de tocar o chão. Isso garante que o pulo ocorra de maneira instantânea assim

que o personagem aterrissa, permitindo que os jogadores realizem sequências de saltos e manobras com rapidez e precisão, o que é crucial para superar desafios de plataformas complexos.

3. *Respawn e Checkpoints*: Os pontos de *respawn* e os *checkpoints* são recursos críticos, por serem usados para reposicionar o jogador em coordenadas específicas do nível após a sua morte. Isso não apenas mantém o ritmo do jogo, permitindo que os jogadores continuem a explorar e enfrentar desafios, mas também garante que o progresso seja mantido, como a recuperação da saúde e a manutenção dos itens coletados. Um exemplo de checkpoint implementado no jogo pode ser visto a seguir:

4. Controles de Precisão: Em jogos *Action Platformer*, os controles precisos são fundamentais para permitir que os jogadores realizem saltos complexos, manobras e desvios com suavidade. Isso é alcançado através da detecção precisa da entrada do jogador, considerando a sensibilidade dos dispositivos de controle, como joystick ou teclado. Os ajustes no código do jogo são feitos para fornecer movimentação suave e responsiva.

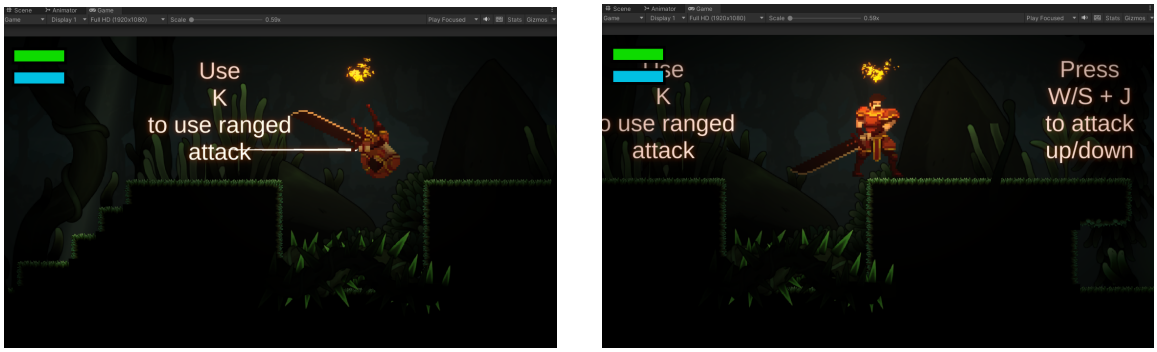
5. *Knockback*: A técnica de "*knockback*"é utilizada para criar uma sensação de impacto. Quando um personagem é atingido por um ataque, ele é empurrado para trás. Isso pode ser explorado para deslocar o personagem para longe ou perto de perigos inimigos, aumentando o dinamismo das situações de combate. Um exemplo do *knockback* implementado no jogo pode ser visto na figura a seguir:



**(a)** *Antes do ataque* **(b)** *Depois do ataque*

**Figura 2.3:** *Exemplo Knockback. Imagem elaborada pelos autores*

6. *Dash*: A habilidade de *Dash* permite que o jogador realize um movimento rápido e controlado em uma direção específica, adicionando um elemento estratégico à jogabilidade. Isso é frequentemente usado para evitar obstáculos, alcançar áreas distantes ou aprimorar as capacidades de combate. A mecânica de *Dash* pode ser especialmente útil em situações de ação intensa e plataformas desafiadoras. Um exemplo de *dash* sendo utilizado para rapidamente atravessar um buraco pode ser visto nas figuras a seguir:




**(a)** *Durante o dash* **(b)** *Depois do dash*

**Figura 2.4:** *Exemplo de Dash. Imagem elaborada pelos autores*

7. *Bounce*: é uma mecânica na qual o personagem do jogador é impulsionado para cima quando entra em contato com um objeto, inimigo ou superfície que normalmente o empurraria para baixo, como um inimigo ou um obstáculo. Esse movimento ascendente, muitas vezes, é usado para dar ao jogador a oportunidade de superar obstáculos, alcançar áreas mais altas ou realizar ataques aéreos. Um exemplo da mecânica de *Bounce* implementada no jogo pode ser vista na figura a seguir, em que o *bounce* ocorre após o jogador atacar para baixo um inimigo:

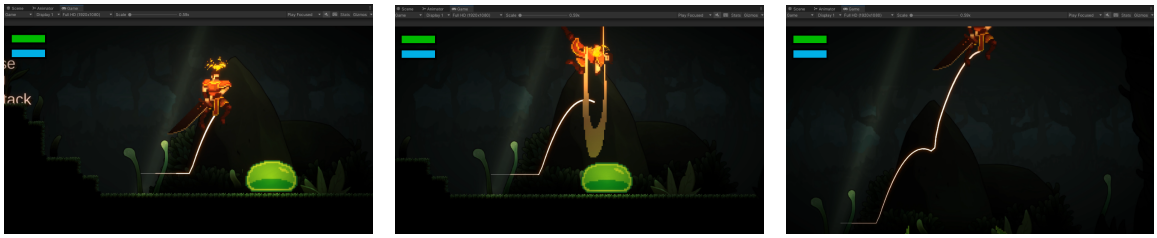


**Figura 2.5:** *Exemplo de Bounce. Imagem elaborada pelos autores*

8. *Wall climbing*: *Wall climbing* é uma técnica que permite que os jogadores subam paredes verticalmente. Isso adiciona uma dimensão vertical à jogabilidade e permite explorar novas áreas ou evitar obstáculos. A implementação de *wall climbing* envolve cálculos precisos de colisão e movimentação suave ao longo das superfícies das paredes.
9. *Crouching* e *slide*: são movimentos que permitem que o jogador se agache e deslize pelo chão. Esses movimentos são frequentemente usados para passar por espaços estreitos, escapar de inimigos ou executar manobras evasivas. A técnica requer animações de personagem realistas e detecção precisa de colisão.

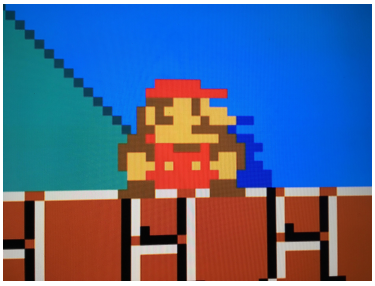
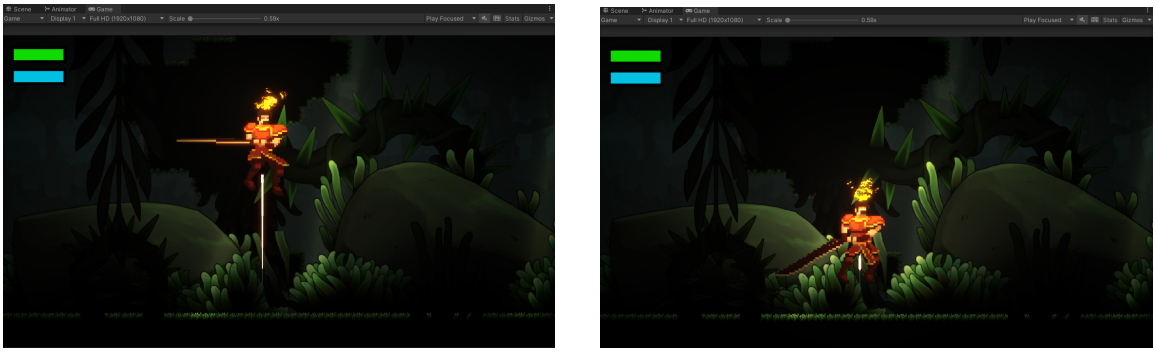


**Figura 2.6:** *Mario Crouching. Disponível em* SlimCK. 8-Bit Small Mario Crouching. *2019.* URL: *https://www.reddit.com/r/MarioMaker2/comments/c7aiuw/8bit_small_mario_crouching (acesso em 01/12/2023)*

10. Pulo duplo: O pulo duplo é uma manobra que permite ao jogador pular no ar uma segunda vez após o primeiro salto. Essa técnica é amplamente utilizada em jogos de plataforma para alcançar áreas distantes ou evitar quedas. Requer o gerenciamento adequado da altura e do tempo de pulo.
11. Controle de altura do pulo: é uma técnica usada para que dependendo do quanto o botão de pulo for segurado, mais alto o jogador irá pular. Isto influencia em pulos mais precisos e rápidos, visto que não há necessidade de se pular a altura máxima em todos os momentos. Um exemplo de controle de pulo implementando no jogo pode ser visto na figura a seguir:



**(a)** *Pulo com altura máxima* **(b)** *Pulo com altura mínima*

**Figura 2.7:** *Exemplo de controle de altura de pulo. Imagem elaborada pelos autores*

Essas técnicas são exemplos de como a programação e o design de jogos de ação e plataforma exigem atenção cuidadosa aos detalhes técnicos para criar uma experiência de jogabilidade suave e satisfatória. Cada uma delas contribui para a sensação única e responsiva desse gênero de jogo.

## 2.2 Princípios do Game Design

Neste capítulo explicaremos os princípios do Game Design. Antes de começar a enumerá-los, é importante que duas coisas estejam bem claras:

- O que é um princípio de Game Design?

  Um princípio de game design é uma regra ou boa prática que desenvolvedores seguem à medida que produzem um jogo. O objetivo é engajar o jogador e se certificar que a experiência do mesmo com a gameplay seja agradável.

- Por que esses princípios são importantes?

  Segundo a Juego Studios (Juego, 2023), estúdio especializado em design de jogos, os princípios do Game Design são importantes porque eles servem de manual para a criação de jogos bem estruturados, imersivos, divertidos e recompensadores. A experiência do jogador não pode ser prejudicada pelas visões dos desenvolvedores, ao mesmo tempo que um desenvolvedor não deve abandonar suas ideias por medo de estar saindo muito da curva do que é considerado mais tradicional.

Com isso e mente, é possível começar a explicar esses princípios.

### 2.2.1 Estabelecendo objetivos

Objetivos são extremamente importantes em jogos. Sem eles, jogadores podem ficar desinteressados e abandonar o jogo antes de terminá-lo. Tais objetivos podem ser complexos e quebrados em diversos sub-objetivos.

Até mesmo jogos clássicos e bem simples que fizeram muito sucesso seguem esse princípio. Tetris, por exemplo, possui um conjunto bem pequeno e definido de regras e operações. O objetivo é bem claro, conseguir se manter jogando, sem perder, o maior tempo possível e acumular o máximo de pontos. Para atingir esse objetivo, o jogador deve completar alguns sub-objetivos, sendo estas coisas extremamente básicas como: conhecer a forma de todas as peças, praticar rotações e translações das mesmas peças e entender como maximizar a quantidade pontos ganhos por linha completada.

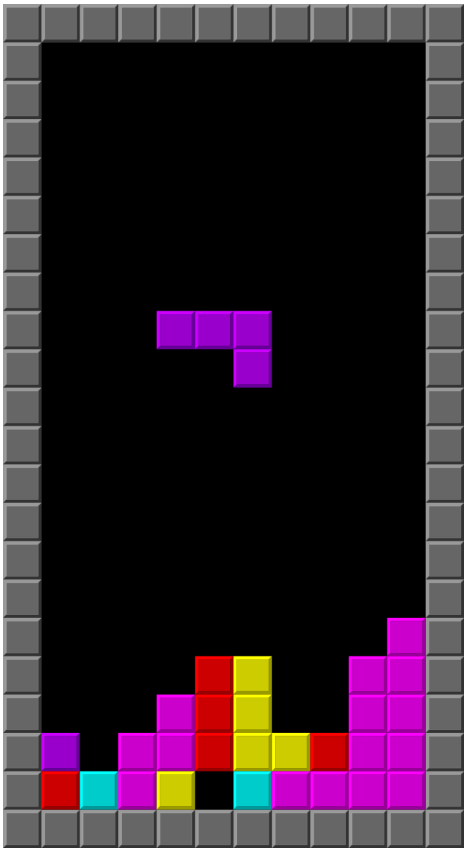


**Figura 2.8:** *Tetris, onde o objetivo é empilhar as peças de maneira que formem linhas horizontais. Disponível em:* WIKIPEDIA. Tetris. *2023.* URL: https://pt.wikipedia.org/wiki/Tetris *(acesso em 28/11/2023)*

Esses conjuntos de sub-objetivos em jogos geralmente requerem que o jogador passe por algum processo metódico para se acostumar com as regras e controles. Quando o objetivo maior é finalmente alcançado, o jogador se sente recompensado e orgulhoso.

### 2.2.2 Engajamento com mecânicas básicas

Mecânicas de gameplay são o esqueleto de um jogo. É importante que o jogador consiga interagir com elas de maneira simples e fluida.

Continuando no mesmo exemplo de Tetris, toda ação do jogo pode ser executada apertando algum botão e de maneira intuitiva, no caso, girar a peça. Com isso, um jogador não precisa ficar perdendo tempo raciocinando como executar uma ação específica durante a jogabilidade, ele apenas a executa sem sequer pensar nos passos necessários para isso.

O que pode acontecer é do jogo ter múltiplas mecânicas complicadas que potencialmente irão confundir os jogadores. Se as mecânicas mais básicas são difíceis de entender ou executar, é alta a chance do jogador desistir devido à frustração.

### 2.2.3 Mantendo um bom fluxo de gameplay

Enquanto o jogador joga, é importante que ele não seja interrompido ou distraído por questões não relacionadas aos seus objetivos. Em outras palavras, é importante que a interface do jogo seja fácil de entender. Identificar e interagir com elementos, controles e menus deve ser algo que o jogador consiga fazer sem precisar perder tempo procurando e adivinhando o que cada coisa faz. Isso é algo que quebra completamente a imersão e o fluxo e, consequentemente, pode arruinar a experiência do jogador.

Também é importante se utilizar de elementos para guiar o jogador na direção correta. Por exemplo, em diversos jogos é comum paredes que se destacam do resto ambiente indicarem uma passagem secreta. Alguns jogos explicitamente vão usar caminhos ou trilhas no chão para guiar os jogadores até alguma região, outros podem esconder essas informações em diálogos de NPCS, padrões na paisagem, etc. O importante é que o jogo ofereça essas informações para fazer com que o jogador não fique preso.


**Figura 2.9:** *No jogo Metal Slug, uma seta aponta a direção que o jogador deve seguir. Disponível em* Miggoh. Branching Paths (Spoiler free). *2014.* URL: *https://steamcommunity.com/sharedfiles/filedetails/?l=brazilian%5C&id=224590222 (acesso em 01/12/2023)*

### 2.2.4 Balanceando o jogo

Jogadores se sentem recompensados e motivados quando eles conseguem superar desafios impostos pelo jogo e atingem seus objetivos. No entanto, alguns desenvolvedores exageram nesse aspecto e acabam criando jogos extremamente difíceis e frustrantes.

Um exemplo de jogo difícil, porém balanceado, é Dark Souls 3. Nesse jogo, alguém inexperiente provavelmente morrerá inúmeras vezes no mesmo local da morte anterior. Trata-se de um jogo que pune o jogador que corre pelo mapa sem prestar atenção e sem se preocupar com outros inimigos, possui um combate bem metódico, exigindo tempo de reação e paciência. No entanto, é um jogo onde as mecânicas e controles são bem consistentes e, uma vez que o jogador se acostuma, o jogo dá impressão de ter ficado fácil. Se os ataques inimigos fossem imprevisíveis ou impossíveis de evitar, isso poderia caracterizar o jogo como injusto.


**Figura 2.10:** *Chefe em Dark Souls 3. Disponível em* Oliver Cragg. Dark Souls 3 video guide: How to beat first boss Iudex Gundyr. *2016.* URL: *https://www.ibtimes.co.uk/dark-souls-3-video-guide-how-beat-first-boss-iudex-gundyr-1554218 (acesso em 01/12/2023)*

Não há problema em um jogo ser difícil, mas jogos injustos vão apenas frustrar e afastar seus jogadores.

### 2.2.5 Oferecendo recompensas

Recompensas e feedbacks são uma ótima forma de manter o jogador motivado. Obviamente, tais recompensas e feedbacks devem combinar com o tema e estilo do jogo. Por exemplo, é comum em RPGs, personagens desbloquearem novos itens ou habilidades à medida que os jogadores sobem de nível, motivando-os a continuar jogando e desbloqueando mais coisas. Alguns games oferecem selos de conquista ou troféus, indicando que aquele jogador realizou algo importante ou icônico no jogo.

### 2.2.6 Testando o jogo

Uma das partes mais importantes do desenvolvimento de qualquer jogo é o *Playtesting*[1]. Ele permite verificar se um jogo é divertido e se isso é algo que se mantém durante todo o *playthrough*[2]. Diversos jogos começam muito divertidos, mas começam rapidamente a ficarem tediosos ou enjoativos. Alguns sofrem do contrário: começam muito monótonos e chatos, mas ficam mais interessantes no final. Isso acontece com muita frequência porque, inicialmente, os desenvolvedores tinham mais ideias e gastaram mais tempo com conceitos e mecânicas para uma etapa específica do jogo, à medida que as outras partes foram vistas como secundárias ou menos relevantes.

Testar um jogo é importante não só para encontrar possíveis falhas no design do jogo, mas também garantir que o alto nível de qualidade se mantenha do início ao fim.

Não existe um único intervalo na linha do tempo para se testar um jogo. Isso deve ser feito regularmente desde o início do desenvolvimento, quando as principais mecânicas

[1] Um playtest é uma jogatina com intuito de achar falhas de design e bugs em um jogo antes de lançá-lo ao mercado.

[2] Um playthrough é jogar o jogo até o fim.

básicas estão surgindo, até o final, quando o jogo já está praticamente pronto para ser publicado (final da fase beta).

A única maneira de testar um jogo é colocar humanos para jogar e coletar feedback. No fim das contas, o principal objetivo de um jogo é que ele seja divertido. Todo o processo criativo e esforço por trás do desenvolvimento podem perder completamente o valor se ele não foi devidamente testado.

## 2.3 Arquiteturas de jogos orientadas a objetos

A programação orientada a objetos (POO) é um modelo de programação amplamente utilizado, que se baseia na criação de classes, as quais representam modelos para a construção de objetos do mundo real. Cada classe possui características que definem atributos e métodos específicos para objetos. Esses métodos descrevem o comportamento dos objetos, enquanto os atributos determinam seus estados. Herdada de paradigmas anteriores, a POO se destaca por suas principais características, como herança, polimorfismo, encapsulamento e abstração, essenciais para a construção de sistemas complexos.

Para ilustrar a importância da POO e suas características, é relevante mencionar o livro "Design Patterns: Elements of Reusable Object-Oriented Software", escrito por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Este livro seminal explora princípios fundamentais de POO e descreve padrões de projeto que promovem a reutilização de código, a flexibilidade e a manutenção eficiente de sistemas de software. Os princípios mencionados anteriormente são a base para a implementação bem sucedida desses padrões.

No contexto dos jogos, a aplicação da POO é evidente na construção de personagens, inimigos, sistemas de habilidades e interações entre elementos do jogo. A herança permite compartilhar comportamentos comuns, o polimorfismo permite que objetos de diferentes classes respondam de maneiras distintas aos mesmos métodos, o encapsulamento protege informações sensíveis e a abstração simplifica a representação de entidades do jogo, tornando o código mais organizado e modular. Além disso, classes e objetos desempenham um papel central na modelagem de elementos do jogo, como personagens, inimigos, itens e ambientes, tornando a POO uma abordagem essencial na criação de jogos complexos e interativos.

### 2.3.1 Delegação

No desenvolvimento de jogos, a capacidade de controlar eventos e ações de forma eficaz e flexível é fundamental para a criação de experiências interativas envolventes. Uma técnica de programação chamada “Delegação” desempenha um papel importante neste processo, permitindo que objetos respondam a eventos e executem ações de forma dinâmica.

#### O que é

A Delegação é um padrão de design e uma técnica de programação que envolve a passagem de responsabilidades de um objeto para outro. Em essência, um objeto delega

uma tarefa ou ação a outro objeto especializado em lidar com ela. Esse processo permite uma separação eficaz de preocupações e torna o código mais flexível e modular. Abaixo segue um exemplo de aplicação de delegação em uma linguagem orientada a objetos genérica.

**Programa 2.1** Exemplo de delegação em programação orientada a objetos

```
1  class A {
2      void foo() {
3          this.bar();
4      }
5
6      void bar() {
7          print("a.bar");
8      }
9  }
10
11 class B {
12     private delegate A a; // delegation link
13
14     public B(A a) {
15         this.a = a;
16     }
17
18     void foo() {
19         a.foo(); // call foo() on the a-instance
20     }
21
22     void bar() {
23         print("b.bar");
24     }
25 }
26
27 a = new A();
28 b = new B(a); // establish delegation between two objects
```

Como é possível observar, a classe B possui um objeto da classe A como variável, e a função *foo* da classe B delega sua funcionalidade para o objeto da classe A.

### Aplicação da Delegação em Jogos

A Delegação é amplamente aplicada em jogos para gerenciar comportamentos de personagens, sistemas de interação e eventos do jogo. Alguns exemplos de como a Delegação pode ser usada neste contexto:

1. Controle de Personagens

Suponha que estamos desenvolvendo um jogo onde o personagem pode se ter diversas habilidades especiais. Cada habilidade pode ser implementada como um objeto delegado. Quando o jogador utilizar uma habilidade específica, o objeto delegador (personagem), pode delegar a execução dessa habilidade ao objeto delegado responsável por ela. Isso permite que o jogador em si não precisa lidar com todas as execuções de todas as habilidades possíveis, deixando uma organização mais limpa do código e facilitando a expansão e reutilização de habilidades ao longo do jogo.

2. Sistema de Colisão

Jogos precisam lidar com físicas e como as colisões interagem. A delegação pode ser usada para tratar as diferentes formas que uma colisão deve acontecer. Um exemplo é: suponha que o jogador tenha a habilidade de soltar uma bola de fogo, porém a bola de fogo não é eficaz contra inimigos na água. Neste caso, o objeto delegador, como um detector de colisão, pode delegar a tarefa de lidar com uma colisão específica a um objeto delegado (o inimigo), para lidar como a colisão deve ser tratada. Isso permite que diferentes objetos reajam de maneira personalizada às colisões, sem a necessidade de uma lógica de colisão complexa em cada objeto.

3. Eventos do Jogo

Eventos em jogos, como o início de uma missão, a conclusão de um nível ou a interação do jogador com um NPC, podem ser gerenciados por meio de Delegação de Eventos. Um objeto delegador (por exemplo, um sistema de missões) pode delegar a responsabilidade de lidar com eventos específicos a objetos delegados (como missões individuais). Isso torna o sistema de eventos escalável e flexível.

### Vantagens da Delegação

A Delegação oferece várias vantagens na programação de jogos:

- Modularidade: A Delegação permite que os sistemas de jogos sejam compostos por objetos independentes e especializados, facilitando a criação e a manutenção do código.
- Reutilização de Código: Os objetos delegados podem ser reutilizados em diferentes contextos do jogo, economizando tempo e esforço de desenvolvimento.
- Extensibilidade: A adição de novas funcionalidades ou comportamentos ao jogo é facilitada pela introdução de novos objetos delegados.

A Delegação é uma técnica valiosa na programação de jogos que permite a alocação eficiente de responsabilidades entre objetos. Através da delegação é possível criar jogos

mais modulares, extensíveis e reutilizáveis. Ao compreender e aplicar a descentralização adequadamente, os desenvolvedores podem simplificar o desenvolvimento de jogos complexos e interativos.

### 2.3.2 Classes Abstratas

A programação de jogos é um campo que frequentemente lida com hierarquias complexas de objetos e comportamentos. As classes abstratas são uma técnica importante na programação orientada a objetos, fornecendo uma maneira de definir estruturas e comportamentos comuns entre diferentes objetos.

#### O que é

Uma classe abstrata é uma classe que não pode ser usada diretamente, mas serve como base para classes derivadas dela. Ela contém métodos e propriedades compartilhados por todas as classes derivadas. As classes derivadas implementam detalhes específicos das funções da classe abstrata herdada, podendo ter diferentes implementações dependendo da classe derivada.

#### Características de uma classe abstrata

- Não pode ser instanciada diretamente: um objeto não pode ser instanciado diretamente de uma classe abstrata. Ela é destinada a ser usado como uma classe base.
- Pode Conter Métodos Abstratos: Uma classe abstrata pode conter métodos abstratos, os quais são métodos que não têm uma implementação definida na classe base, mas devem ser implementados por classes derivadas.
- Pode Conter Métodos Concretos: Além de métodos abstratos, uma classe abstrata pode conter métodos com implementações concretas que podem ser usados pelas classes derivadas.

#### Aplicação de Classes Abstratas em Jogos

As classes abstratas são amplamente utilizadas na programação de jogos para criar hierarquias de objetos e comportamentos compartilhados. Alguns exemplos de como podem ser utilizadas:

1. Movimentação de Personagens

Em um jogo, diferentes personagens, como guerreiros, magos e arqueiros, podem compartilhar características comuns, como movimentação. Uma classe abstrata "Movimentação de Personagem"pode definir essas propriedades e métodos comuns, enquanto as classes derivadas (como "Guerreiro"e "Mago") implementam os detalhes específicos da movimentação de cada personagem.

2. Objetos Interativos Em jogos, objetos interativos, como portas, alavancas e baús, frequentemente compartilham funcionalidades comuns, como interação com o jogador. Uma classe abstrata "ObjetoInterativo"pode definir os métodos para interação (por exemplo,

"Abrir"e "Fechar"), enquanto as classes derivadas (como "Porta"e "Baú") implementam o comportamento específico de cada objeto.

3. Efeitos e Magias Em jogos que envolvem feitiços e efeitos especiais, muitos feitiços podem ter comportamentos comuns, como causar dano ou aplicar efeitos de condição. Uma classe abstrata "Magia"pode definir métodos que aplicam esses efeitos gerais, enquanto classes derivadas (como "BolaDeFogo"e "Congelar") implementam os efeitos específicos de cada feitiço.

**Vantagens das Classes Abstratas**

As classes abstratas oferecem várias vantagens na programação de jogos:

- Reutilização de código: classes abstratas permitem definir um comportamento comum uma vez e reutilizá-lo em várias partes do sistema.
- Hierarquia: Hierarquias de classes abstratas ajudam a organizar objetos e comportamentos em uma hierarquia lógica.
- Manutenção simplificada: Mudanças gerais de comportamento podem ser feitas em classes abstratas, afetando todas as classes que derivam delas.

Ao utilizar classes abstratas apropriadamente, o desenvolvimento de um jogo pode ser feito de maneira efetiva, melhorando a organização do código, além de facilitar a interação de diferentes objetos via métodos da classe abstrata, sem necessidade de se saber o comportamento específico destes.

### 2.3.3 Interfaces

No desenvolvimento de um jogo, muitas vezes é necessário especificar um conjunto de métodos que um grupo de classes deverá implementar. As interfaces, consideradas entidades, servem para isso.

**O que é**

Uma interface define um “contrato” a ser seguido por outras classes. Este contrato é composto por cláusulas, que descrevem determinados comportamentos que este grupo de classes deverá, obrigatoriamente, seguir. No entanto, as interfaces não contêm implementações reais dos métodos. Isso permite que objetos de diferentes classes compartilhem funcionalidades comuns por meio da implementação de uma ou mais interfaces. A seguir é possível ver um exemplo de utilização de interface em programação orientada a objetos, onde as classes *car*, *truck* e *boat* implementam duas interfaces, a interface *Breakable* e *Driveable*.

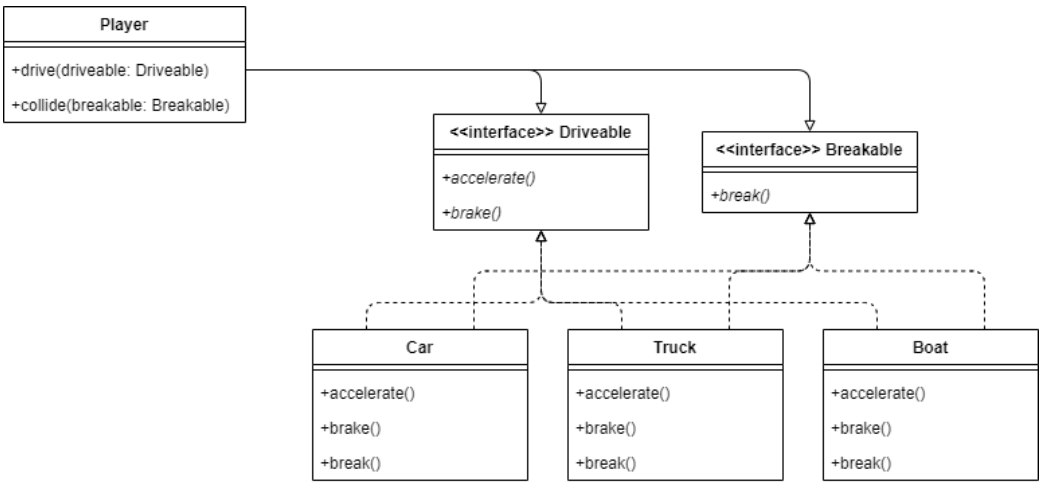



**Figura 2.11:** *Exemplo de aplicação de interface. Disponível em:* ATTILA FEJÉR. What Does It Mean to Program to Interfaces? *2023.* URL: *https://www.baeldung.com/cs/program-to-interface (acesso em 24/11/2023)*

Dessa forma, no exemplo acima, a classe jogador (*Player*) pode referenciar tanto o carro, quanto o caminhão quanto o barco apenas utilizando as interfaces *Breakable* e *Driveable* definidas.

### Características de uma Interface

- Não pode ser instanciada diretamente: um objeto não pode ser instanciado diretamente de uma interface.
- Declaração de Métodos: As interfaces definem métodos e propriedades sem fornecer a implementação real desses métodos.
- Implementação Obrigatória: Qualquer classe que implementa uma interface deve fornecer uma implementação para todos os métodos definidos na interface.
- Múltiplas Interfaces: Uma classe pode implementar várias interfaces. Isso implica que ela deve cumprir múltiplos contratos diferentes.

### Aplicação de Interfaces em Jogos

As classes abstratas são amplamente utilizadas na programação de jogos para criar hierarquias de objetos e comportamentos compartilhados. Alguns exemplos de como podem ser utilizadas:

1. Gerenciamento de missões

Em jogos de mundo aberto, geralmente existe um sistema de missões, dado por um personagem ou mediante um certo evento. Um interface "IMissao"pode implementar métodos como "Checkprogress"e "CompleteMission". Cada missão implementa esta interface para fornecer uma lógica específica para aquele tipo de missão.

2. Sistema de Diálogo

Diversos jogos implementam diálogo entre personagens. Uma interface chamada "IDialogo"pode definir métodos como "IniciarDialogo"e "AvancarFala". Isso permite que diferentes personagens no jogo, implementem comportamentos de diálogo únicos.

**Vantagens das Interfaces em Jogos**

As interfaces oferecem várias vantagens na programação de jogos:

Conclusão

- Obrigatoriedade de Contratos: As interfaces garantem que os objetos cumpram contratos específicos, tornando os contratos padronizados.
- Flexibilidade: Os objetos podem implementar várias interfaces, permitindo que sejam usados em diferentes contextos e sistemas do jogo.
- Desacoplamento: Interfaces facilitam o desacoplamento, permitindo que objetos se comuniquem por meio de contratos, sem depender de suas implementações específicas.

### 2.3.4 Diferença entre Interfaces e Classes Abstratas

1. Implementação de Métodos

- Interfaces: As interfaces declaram métodos que devem ser implementados por classes que as utilizam, mas não fornecem a implementação real desses métodos. A implementação é obrigatória nas classes que implementam a interface.
- Classes Abstratas: Classes abstratas podem conter métodos abstratos (sem implementação) e métodos concretos (com implementação). As classes derivadas de uma classe abstrata devem fornecer uma implementação para os métodos abstratos, mas podem optar por substituir ou estender os métodos concretos.

2. Herança

- Interfaces: As classes podem implementar múltiplas interfaces, permitindo que cumpram vários contratos diferentes. Não há herança direta de implementações em interfaces.
- Classes Abstratas: Uma classe abstrata é usada como uma classe base que pode ser herdada por classes derivadas. As classes derivadas herdam a implementação dos métodos concretos da classe abstrata.

3. Flexibilidade

- Interfaces: As interfaces fornecem maior flexibilidade, pois as classes não estão vinculadas a uma única hierarquia de herança. Isso permite que objetos de classes diferentes cumpram contratos comuns.
- Classes Abstratas: Classes abstratas são úteis quando há uma necessidade de compartilhar código e implementação comum entre classes derivadas, criando uma hierarquia de herança.

As interfaces desempenham um papel fundamental na programação de jogos, permitindo que os desenvolvedores estabeleçam contratos e diretrizes que os objetos devem atender. Quando empregadas corretamente, as interfaces auxiliam na uniformidade, adaptabilidade e redução de acoplamento nos jogos, simplificando o processo de desenvolvimento e manutenção.

## 2.4 Técnicas para representação de comportamentos em jogos

No desenvolvimento de jogos, técnicas para representar comportamentos desempenham um papel fundamental, por serem elas que dão vida aos personagens e controlam suas ações em resposta às interações dos jogadores. Duas dessas técnicas são as Behavior Trees (Árvores de Comportamento) e as State Machines (Máquinas de Estados), que proporcionam estruturas organizadas e eficazes para definir como os personagens virtuais agem e reagem no mundo do jogo. A importância dessas técnicas reside na capacidade de criar comportamentos complexos, realistas e responsivos, proporcionando uma experiência envolvente aos jogadores.

### 2.4.1 Behavior Tree

#### O que são

As Behavior Trees são uma estrutura de controle de fluxo que modela o comportamento de agentes autônomos, como personagens de jogos, em um formato hierárquico de árvore. Elas foram originalmente desenvolvidas para a indústria de robótica, mas foram amplamente adotadas na indústria de desenvolvimento de jogos devido à sua flexibilidade e facilidade de compreensão. Abaixo segue um exemplo de uma BT onde o "robô" vasculha $N$ locais até encontrar um objeto. O funcionamento de cada nó e do fluxo de execução serão explicados nas seções seguintes.

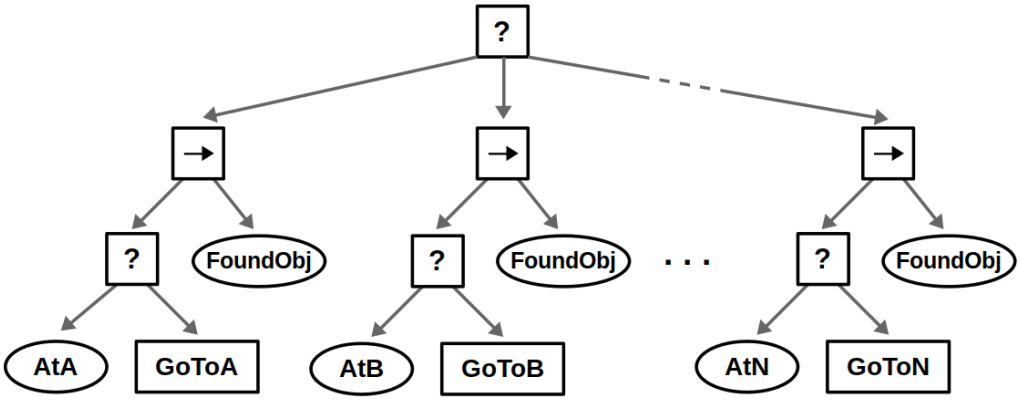



**Figura 2.12:** *Exemplo de uma BT. Disponível em:* Sebastian Castro. Introduction to Behavior Trees. *2021.* URL: https://robohub.org/introduction-to-behavior-trees/ *(acesso em 28/11/2023)*

### Componentes de uma Behavior Tree

Uma Behavior Tree é composta por uma série de nós interconectados que representam ações e condições. Os três tipos principais de nós em uma Behavior Tree são:

- Nós de Comportamento (Behavior Nodes): Esses nós representam ações que o personagem pode realizar, como atacar, mover-se, interagir com objetos, etc.
- Nós de Controle (Control Nodes): Esses nós controlam a execução dos nós de comportamento e condição e podem incluir sequenciadores, seletores e decoradores.
- Nós de Condição (Condition Nodes): Esses nós verificam as condições do ambiente ou do personagem, como verificar se o inimigo está próximo, se o personagem está com pouca vida, etc.

Os símbolos mais comumente utilizados para esses tipos de nós podem ser vistos na figura a seguir:

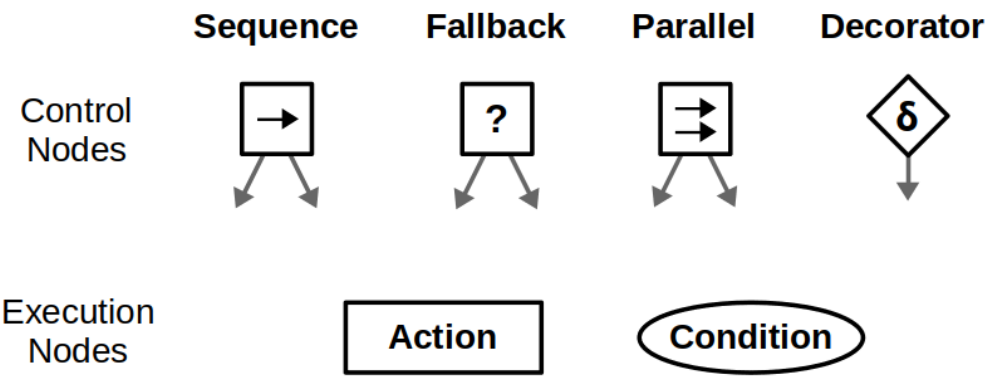



**Figura 2.13:** *Tipos de nós em Behavior Trees. Disponível em: Sebastian Castro.* Introduction to Behavior Trees. *2021. URL: https://robohub.org/introduction-to-behavior-trees/ (acesso em 28/11/2023)*

### Estrutura Hierárquica

A estrutura hierárquica das Behavior Trees permite que os desenvolvedores organizem o comportamento dos personagens de forma clara e lógica. A árvore é geralmente dividida em várias camadas ou níveis, onde os nós de controle no nível superior determinam o comportamento geral, enquanto os nós de comportamento mais específicos estão no nível inferior.

Por exemplo, um personagem pode ter um nó de controle "Seleção" no nível superior que decide entre atacar um inimigo ou fugir. Se o personagem decidir atacar, a árvore pode se ramificar para incluir nós de comportamento como "Ataque Corpo a Corpo" ou "Ataque à Distância".

### Tomada de Decisões com Behavior Trees

A execução de uma Behavior Tree envolve a travessia da árvore a partir do nó raiz até os nós de comportamento relevantes. Durante essa travessia, os nós de controle determinam o fluxo de decisões com base nas condições definidas. Este são:

- Seletores (Selectors/Fallback): Um seletor avalia seus filhos em ordem até encontrar um nó de comportamento que pode ser executado com sucesso. Isso permite que o personagem escolha entre várias ações possíveis, selecionando a primeira que seja viável.

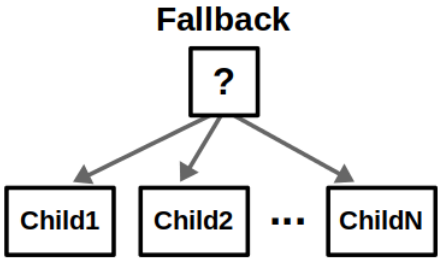



**Figura 2.14:** *Nó Selector em BTs. Disponível em: Sebastian Castro.* Introduction to Behavior Trees. *2021. URL: https://robohub.org/introduction-to-behavior-trees/ (acesso em 28/11/2023)*

- Sequenciadores (Sequencers): Um sequenciador executa seus filhos em ordem, apenas avançando para o próximo nó quando o anterior for bem-sucedido. Isso permite definir uma sequência específica de ações.

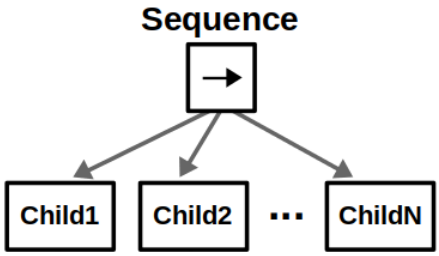



**Figura 2.15:** *Nó Sequence em BTs. Disponível em: Sebastian Castro.* Introduction to Behavior Trees. *2021. URL: https://robohub.org/introduction-to-behavior-trees/ (acesso em 28/11/2023)*

- Decoradores (Decorators): Os decoradores modificam o comportamento de um nó filho, adicionando condições extras ou limitações.

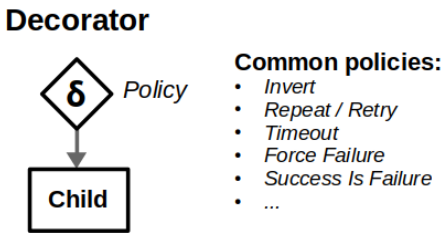



**Figura 2.16:** *Nó Decorator em BTs. Disponível em:* Sebastian Castro. Introduction to Behavior Trees. *2021.* URL: *https://robohub.org/introduction-to-behavior-trees/ (acesso em 28/11/2023)*

### Processo de Tomada de Decisão

O processo de tomada de decisão em uma Behavior Tree geralmente segue este fluxo:

1. Começando pelo nó raiz, a árvore é percorrida de cima para baixo. Os nós de controle (seletores, sequenciadores) avaliam seus filhos, decidindo se a execução continua ou muda de direção.

2. Os nós de comportamento executam ações específicas com base nas condições estabelecidas.

3. Os resultados da execução são propagados de volta à raiz, afetando as decisões subsequentes.

### Vantagens das Behavior Trees

As BTs oferecem várias vantagens na programação de jogos:

- Facilidade de Visualização: A estrutura em árvore é fácil de compreender e visualizar, tornando o desenvolvimento e a depuração mais eficiente.

- Flexibilidade: As árvores podem ser facilmente adaptadas e modificadas para diferentes comportamentos sem a necessidade de reescrever o código principal.

- Lógica Clara: A divisão hierárquica de comportamentos facilita a organização da lógica de tomada de decisões.

As BTs são uma técnica valiosa na programação de jogos, permitindo que os desenvolvedores controlem o comportamento de personagens virtuais de maneira eficaz e compreensível. Com sua estrutura hierárquica e diversos tipos de nós, as BTs oferecem flexibilidade, clareza e eficiência na criação de comportamentos envolventes e dinâmicos para jogos.

### 2.4.2 State Machines

#### O que são

Uma máquina de estados é uma técnica de programação que modela o comportamento de um objeto ou personagem como um conjunto finito de estados, cada um com suas próprias ações e transições. Cada estado representa uma situação específica em que o objeto pode se encontrar e define como ele reage aos eventos e ações do jogador ou do ambiente.

#### Componentes de uma State Machine

Uma Máquina de Estados é composta por três elementos principais:

- Estados (States): Um estado representa uma situação na qual um objeto pode existir. Cada estado define um conjunto de ações que um objeto pode executar enquanto estiver nesse estado.
- Transições (Transitions): A transição é uma condição que determina quando um objeto passará de um estado para outro. Eles estão vinculados a eventos ou condições específicas que ocorrem no jogo.
- Controlador de Estados (State Controller): O controlador de estados gerencia o estado atual do objeto e as transições entre estados. Ele decide qual estado está ativo em um determinado momento e quando as transições devem ocorrer.

Como exemplo segue uma máquina de estados da lógica de uma catraca que abre ao receber uma moeda:

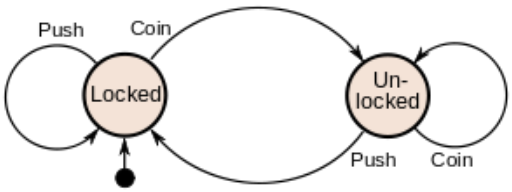



**Figura 2.17:** *Lógica de uma catraca utilizando FSM. Disponível em: Wikipedia.* Finite-state machine. *2023. URL: https://en.wikipedia.org/wiki/Finite-state_machine (acesso em 28/11/2023)*

#### Estrutura Hierárquica

Muitos jogos exigem uma hierarquia de máquinas de estado devido à complexidade do comportamento dos personagens. Máquinas de estado são frequentemente usadas em conjunto com árvores de decisão.

Digamos que você esteja desenvolvendo um RPG onde seus personagens podem entrar em estados de “exploração” ou “combate”. No estado de “combate”, existem estruturas hierárquicas como “ataque” e “defesa”. Isso permite que os personagens reajam de maneira diferente durante o combate, dependendo das ações do jogador.

### Tomada de Decisões com State Machines

A tomada de decisões em uma Máquina de Estados é controlada pelos próprios estados e pelas transições entre eles. Cada estado determina suas ações com base nas condições internas e nos eventos externos que ocorrem no jogo.

### Processo de Tomada de Decisão

O processo de tomada de decisão em uma Máquina de Estados geralmente segue este fluxo:

1. Inicialização: O objeto é inicializado em um estado inicial predefinido.
2. Execução Contínua: O objeto permanece no estado atual e executa as ações associadas a esse estado até que uma transição seja acionada.
3. Transição de Estados: Quando uma condição de transição é atendida (por exemplo, um evento específico ocorre), o objeto muda para um novo estado conforme as regras definidas.
4. Ações no Novo Estado: O objeto executa as ações associadas ao novo estado, e o ciclo continua.

### Vantagens das State Machines

As State Machines oferecem várias vantagens na programação de jogos:

- Organização Estruturada: Os estados e transições fornecem uma estrutura organizada e clara para o comportamento do objeto.
- Facilidade de Manutenção: Alterar ou adicionar comportamentos é mais simples, pois as mudanças ocorrem em estados e transições específicos.
- Flexibilidade: É fácil criar personagens com comportamentos complexos, pois cada estado é uma unidade independente de funcionalidade.

As máquinas de estado são uma técnica poderosa que pode ser usada na programação de jogos, permitindo aos desenvolvedores controlar o comportamento de personagens e objetos de forma estruturada e eficaz. Estados, transições e controladores de estado apropriados permitem criar um comportamento responsivo, simples e dinâmico para personagens em seus jogos.

# Capítulo 3

# Desenvolvimento do projeto

## 3.1 Arquitetura de classes

Para programar o projeto, foi utilizado C#, uma linguagem orientada a objetos, então técnicas típicas desses tipos de linguagem foram colocadas em prática. Neste capítulo serão explorados alguns conceitos usados na arquitetura geral do projeto.

### 3.1.1 Classes-Base

As classes-base são as classes de entidade, que implementam funcionalidades que tanto o Player quanto os inimigos compartilham, por isso a funcionalidade de algumas dessas classes são especializadas utilizando herança. Essas funcionalidades são:

- Controle de vida (HealthManager): Essa classe é responsável por administrar os pontos de vida da entidade, e possui funções para tomar dano, curar vida, verificar a situação da entidade (morta ou viva) e controlar o intervalo de tempo em que a entidade pode tomar dano (invencibilidade temporária), como é possível observar no Programa A.1.
- Controle de animações executadas em certos eventos (EntityAnimator e EnemyAnimator): A classe EntityAnimator é uma classe abstrata que define as funções de animação de tomar dano, tomar dano crítico e morte. Já a classe EnemyAnimator é uma classe para inimigos genéricos, que não possuem animações próprias de morte ou dano. Ela é responsável por implementar como as animações serão tocadas, utilizando a biblioteca DOTween e o Animator nativo da Unity (ver Programa A.2 e Programa A.3).
- Controle do movimento da entidade (CharacterController): Responsável por administrar as velocidades de movimentação, aceleração e desaceleração do objeto. Então, através dessa classe é possível adicionar e remover velocidades horizontais e verticais e também implementar comportamento de gravidade e inércia (ver Programa A.4).
- Controle da funcionalidade de Knockback (KnockbackBehavior): Responsável por administrar e aplicar a força de knockback, e quanto tempo a mesma deve durar

(Ver Programa A.5).

- Controle de receber ataques e tomar dano (HittableBehaviour): Implementa a lógica completa de um objeto de jogo tomar um ataque, então determina as animações e sons que devem ser tocados e todos os eventos que sucedem um ataque, como o *knockback*, perda de vida, dano crítico, se deve aplicar o *bounce*, aplicar efeitos de câmera, etc. A classe é responsável por administrar e delegar quando as classes-base anteriores devem ser chamadas (ver Programa A.6 e Programa A.7).

### 3.1.2 Player

O Player compartilha diversas funcionalidades com os inimigos, resultando na especialização de classes fundamentais, incluindo HealthManager, EntityAnimator e CharacterController. No que diz respeito ao controle de vida do jogador, é necessário também atualizar a barra de vida na interface visual conforme a condição do jogador. Para abordar essa necessidade, foi derivada a classe PlayerHealthbarManager a partir da HealthManager (ver Programa A.8).

Além disso, o Player compartilha funcionalidades de animação com outras entidades, como a animação de tomar dano e de morte. Por esse motivo, a classe PlayerAnimator foi derivada da EntityAnimator. Para atender às especificidades do Player, foram adicionadas novas funções de animação exclusivas para o jogador, como animações de ataque direcional (ver Programa A.9 e Programa A.10).

A classe de controle do jogador foi desenvolvida com base no pacote "Tarodev 2D Controller". Ela é responsável por receber os inputs do jogador, controlar os ataques, lidar com colisões e movimentar o objeto Player. A movimentação do jogador se destaca em termos de complexidade e precisão em comparação com outras entidades. Essa classe de movimentação do jogador incorpora recursos de física adicionais, como *bounce*, *crouch*, *dash*, *wall-jump*, interrupção de pulo, *buffer* de input, *buffer* de pulo e o *coyote*.

#### Comportamento de ataque

O controle dos ataques que o jogador pode realizar é feito na classe *PlayerAttackBehaviour*, que possui as funções que iniciam tanto o ataque corpo a corpo como o ataque de longo alcance. A classe utiliza delegação para separar os detalhes da implementação de cada ataque. Os detalhes estão definidos nas classes de comportamento de armas. Além disso, a classe possui funções chamadas pelo *Animator* do jogador responsáveis por ajustar o comportamento dos ataques (Exemplo: ao executar um ataque no chão, o jogador não deve se movimentar até que a animação acabe). Como é possível observar no Programa A.11

#### Arma Corpo a corpo

Os detalhes da implementação da arma corpo a corpo são feitos no Script *PlayerWeaponBehaviour*. São definidas chamadas para ativar as animações de ataque específicas de cada arma, isso foi feito de modo com que as animações do jogador e do ataque das armas fossem animadas separadamente. Porém, no projeto implementado atualmente a única

arma que o jogador possui já está integrada na sua animação (*PlayerAttackBehaviour*), deixando a chamada do *PlayerWeaponBehaviour* como uma animação vazia.

De resto, a classe *PlayerWeaponBehaviour* especifica o comportamento de contato da arma com os inimigos, sendo eles: verificar se um ponto fraco foi atingido, ativação/desativação de colisores de dano dependendo da direção que o ataque foi feito, aplicar o dano em entidades colididas e verificar se o jogador pode realizar o *Bounce*. Como é possível observar em Programa A.12 e Programa A.13.

#### Arma de longo alcance

O comportamento da arma de longo alcance está definido no script PlayerRangedWeaponBehaviour, que define apenas uma função que instancia e configura o projétil a ser lançado. Da mesma que o script PlayerWeaponBehaviour, as animações da arma e do player estão separadas por código. Vale ressaltar que por se tratar de uma arma de longo alcance, o projétil se move a uma velocidade constante, atualizada a cada *frame*, por causa disso, a flecha não passa por todos os pixeis e pode acabar não atingindo inimigos que deveria atingir. Para resolver este problema, um *raycast* é traçado entre a posição anterior e a posição atual do objeto, e caso colida com algum inimigo, a função TakeDamage é chamada. Como é possível observar em Programa A.14 e em Programa A.15.

### 3.1.3 Inimigos

Assim como o jogador, os inimigos têm comportamentos que utilizam classes-base, como HealthManager, EntityAnimator e CharacterController. Estes comportamentos são similares ao do jogador, não havendo mudança em suas funcionalidades, com exceção de HittableBehavior, que implementa a criação de um novo tipo de colisão: o ponto fraco.

A principal diferença dos inimigos são como seus comportamentos são implementados, esses comportamentos variam segundo o tipo do inimigo. Como explicado anteriormente, os inimigos são divididos em simples, complexos e chefes. Inimigos simples implementam comportamentos singulares, sem resposta ao estado do jogador. Inimigos complexos envolvem comportamentos variados consoante o estado do jogador, ou seja, inimigos complexos reagem ao jogador, por exemplo: verificar se o jogador está a uma certa distância antes de o atacar. Chefes apresentam os comportamentos mais complexos, tendo diversos tipos de ataques que podem ou não variar conforme o comportamento do jogador.

Para que os mais diversos tipos de comportamento sejam possíveis, são usadas máquinas de estado e árvores de comportamento, e combinações entre elas. Ao combinar máquinas de estados e árvores de comportamento é possível utilizar os pontos fortes de ambas estruturas.

Dentre os inimigos do jogo, temos:

- Slime: Inimigo simples, seu comportamento é andar em uma direção fixa até colidir com uma parede, quando troca a direção. Foi implementado utilizando um simples script de movimentação que implementa as funções de uma classe fundamental AbstractMovement, chamado *SlimeMovement* (ver Programa A.16), e utiliza a classe base CharacterController. O inimigo implementado pode ser visto na figura a seguir:

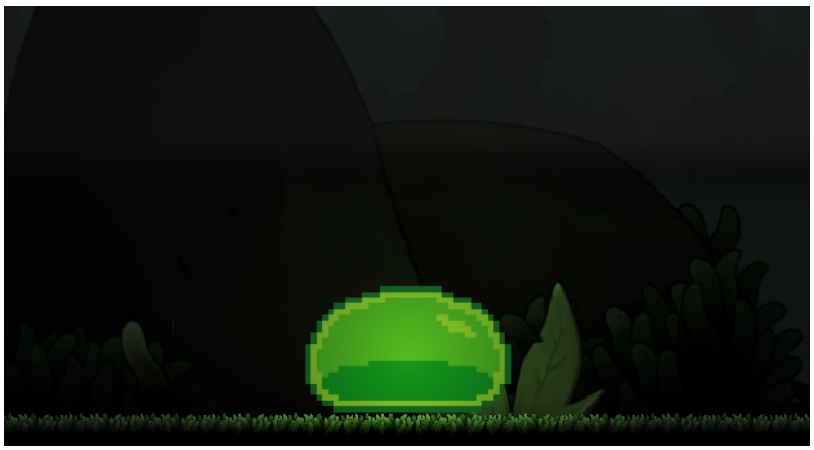


**Figura 3.1:** *Inimigo Slime implementado. Imagem elaborada pelos autores*

- Inimigo voador: inimigo simples, seu comportamento é seguir pontos especificados no mapa e fazer uma patrulha entre eles. Assim como o *slime*, foi implementado com um script "TwoPointMovement" (ver Programa A.19) que utiliza a classe base CharacterController, seguindo a direção dos pontos especificados. O inimigo implementado pode ser visto na figura a seguir:

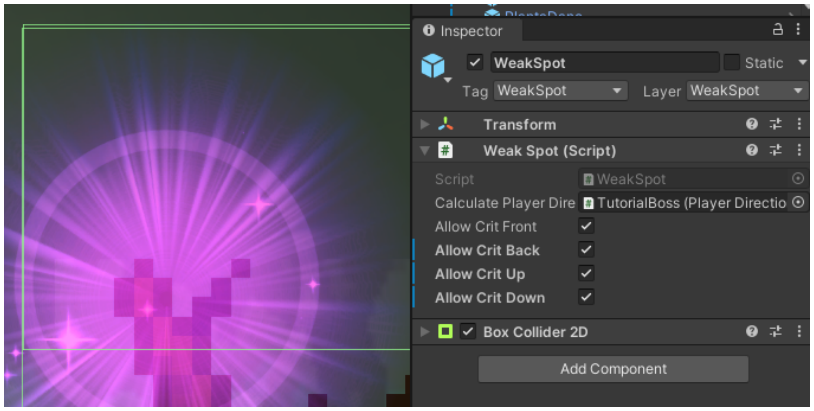
**Figura 3.2:** *Inimigo voador implementado. Imagem elaborada pelos autores*

- Piranha Plant: Inimigo Complexo, verifica a distância na qual o jogador se encontra, caso esteja perto o suficiente, dá uma mordida na direção do alvo. De forma simplificada esse inimigo possui 2 comportamentos: um quando o inimigo está longe, onde fica calculando a distância do jogador; e o de ataque que quando o alvo está próximo realiza o ataque repetidamente. Foi implementado utilizando uma máquina de estados, que pode ser vista com mais detalhes na Subseção 3.2.1.
- Corrupted Figtree: Chefe do tutorial, possuindo três ataques: raízes que prendem o jogador no chão, galho emergindo do chão que dá dano e *knockback*, e por fim, um ataque corpo a corpo. O chefe foi criado utilizando uma combinação de BT, com cada um dos ataques escolhido aleatoriamente a cada ciclo da árvore, e uma FSM, dividida em duas partes: Vivo e morto. Quando o chefe está no estado 'Vivo', a árvore é executada, quando está no estado 'Morto', uma rotina de morte do chefe é executada. No estado morto a BT com as ações inimigas também para de ser executadas. Mais detalhes sobre esse inimigo chefe podem ser vistos na Subseção 3.2.2.

### Ponto Fraco

O ponto fraco é uma área especial de uma entidade que quando atingida de uma determinada direção, se aplica um dano multiplicado ao inimigo, a área do ponto fraco é delimitada por um colisor. É possível observar a implementação de ponto fraco para o inimigo chefe abaixo:

**(a)** *Ponto fraco do inimigo chefe*




**(b)** *Colisor e Script do ponto fraco*

**Figura 3.3:** *Ponto fraco no inimigo chefe* Cursed Tree. *Imagem elaborada pelos autores.*

O sistema de ponto fraco serve como uma forma de aprimorar a experiência de combate, fazendo com o que o jogador possa escolher se quer um combate mais tradicional ou se prefere procurar janelas de oportunidade e precisão para derrotar os inimigos com apenas um ataque.

Este sistema é implementado da seguinte maneira: quando o jogador ataca um inimigo, o script de ataque verifica qual foi o colisor atingido, caso o colisor esteja com uma tag "WeakSpot", o script WeakSpot do inimigo é chamado e verifica se o ataque veio de uma direção válida, pois a ideia é que o ataque tenha que vir de uma direção específica, caso o ataque seja válido, ele devolve um booleano *true* para a rotina de ataque do jogador, que então chama o script HittableBehaviour do inimigo com o modificador booleano, que dará um dano multiplicado especificado no próprio script. Como é possível observar nos trechos de código a seguir:

**Programa 3.1** Função de detecção de pontos fracos e aplicação de dano.

```
1       GameObject[] CheckForWeakSpots(Collider2D[] col) {
2           GameObject[] weakSpotsReached = new GameObject[col.Length];
3
4           int i = 0;
5           foreach(Collider2D c in col) {
6               WeakSpot w = c.GetComponent<WeakSpot>();
7               if (w != null) {
8                   if (w.IsHittable() && w.IsPlayerCorrectPos(transform))
9                       weakSpotsReached[i] =
10                       w.gameObject.transform.parent.parent.gameObject;
11              }
12              i++;
13          }
14
15          return weakSpotsReached;
16      }
17
18      void ApplyEffect(HittableBehaviour hittableBehaviour,
19      bool hitWeakSpot) {
20          if (hittableBehaviour.IsBounceable() &&
21           playerController.ShouldBounce()) {
22              playerController.Bounce();
23          }
24
25          hittableBehaviour.TakeDamage(damage, hitWeakSpot);
26      }
27
28
```

**Programa 3.2** Função dentro de WeakSpot confirma se a direção do ataque é correta.

```
1   public bool IsPlayerCorrectPos(Transform player) {
2       if (calculatePlayerDirection.IsPlayerFront(player) && allowCritFront) {
3           return true;
4       }
5       if (calculatePlayerDirection.IsPlayerBack(player) && allowCritBack) {
6           return true;
7       }
8       if (calculatePlayerDirection.IsPlayerUp(player) && allowCritUp) {
9           return true;
10      }
11      if (calculatePlayerDirection.IsPlayerDown(player) && allowCritDown) {
12          return true;
13      }
14
15      return false;
16  }
```

### Colisão de dano

O script HurtBox tem como função aplicar dano constante enquanto em contato com um colisor. Para ser implementado, o objeto no qual o script está atrelado precisa ter um componente Collider atribuído. Ele funciona da seguinte maneira: com o colisor interno (do objeto do qual o script é componente), e um colisor de um objeto externo, caso o externo entre em contato com o interno, o script comunica a função TakeDamage do colisor externo para aplicar o dano. Como é possível observar no trecho de código a baixo:

**Programa 3.3** Script HurtBox.

```
1  public class HurtBox : MonoBehaviour
2  {
3      private HittableBehaviour collidingEntity;
4      [SerializeField] float damage = 10.0f;
5
6      void OnTriggerStay2D(Collider2D col) {
7          collidingEntity = col.gameObject.GetComponent<HittableBehaviour>();
8          if (col.CompareTag("Player") && collidingEntity != null)
9              collidingEntity.TakeDamage(damage);
10     }
11 }
```

Esse script é usando em diversas áreas do jogo aonde se precisa de uma fonte constante de dano, como: colisão dos inimigos, de modo que eles não possam ser atravessados diretamente e ataques que dão dano ao aplicar contato com o jogador, sejam eles constantes ou apenas uma vez (como os ataques de espinhos do primeiro chefe do jogo ou a mordida da Piranha Plant).

## 3.1.4 Classes auxiliares

As classes auxiliares utilizadas, visa integrar o funcionamento das classes principais e/ou realizar funcionalidades secundárias. Como, por exemplo, a detecção de inimigos para o ataque do Player, ou o cálculo de direção do Knockback das entidades.

### Combate

1. PowerUp

   O Power Up é um item que concede uma habilidade nova para o jogador ao ser coletado. Para realizar essa funcionalidade, os scripts foram separados entre o objeto coletável, objeto do jogador e uma classe abstrata utilizada como base para os diferentes tipos de Power Up.

   A classe abstrata apenas define o método virtual Use() vazio, e o custo de mana para utilizar a habilidade. Como pode ser observado no trecho de código abaixo:

**Programa 3.4** Script AbstractPowerUp.

```
1  public class AbstractPowerUp : MonoBehaviour {
2      public int manaCost;
3      public virtual void Use() { }
4  }
```

Além da classe abstrata, a funcionalidade dos *PowerUps* necessita de duas partes para funcionar, já que esses objetos são separados do jogador, para depois serem coletados:

- Objeto Coletável: Possui dois scripts, um que detecta quando o jogador colide com o PowerUp, o script se chama *CollectablePowerUp* e apenas implementa um detector de colisão com o jogador, como pode ser visto no código a baixo:

**Programa 3.5** Script CollectablePowerUp.

```
1  public class CollectablePowerUp : MonoBehaviour {
2      [SerializeField]
3      public AbstractPowerUp powerUp;
4
5      void OnTriggerEnter2D(Collider2D otherCollider) {
6          if (otherCollider.CompareTag("Player")) {
7              otherCollider.GetComponent<PowerUpManager>().equipPowerUp(powerUp);
8              gameObject.SetActive(false);
9          }
10     }
11 }
```

O outro script que implementa a classe abstrata AbstractPowerUp com a funcionalidade específica desejada. O objeto coletável implementado pode ser visto na figura abaixo, o qual é um *PowerUp* que cura o jogador ao ser utilizado:


**Figura 3.4:** *Exemplo Objeto Coletável Implementado. Imagem elaborada pelos autores.*

Para a implementação é feito o *override* do método *Use()* vazio definido em *AbstractPowerUp*, como é possível observar no trecho de código abaixo:

**Programa 3.6** Override do método Use() vazio.

```
1       public override void Use()
2       {
3           playerHealthManager.Heal(healthRegen);
4       }
```

- Objeto Jogador: O jogador possui o script PowerUpManager que utiliza delegação para controlar a mana do jogador (pontos de mana e interface da mana) e para ativar o efeito equipado. O *Script* verifica todo *frame* se o botão de utilizar o *PowerUp* foi apertado, e se o jogador possui *mana* o suficiente, se sim o efeito do coletável é ativado. Como é possível observar no código a baixo (Pode-se ver o programa inteiro no Programa A.21):

**Programa 3.7** Detecção de input do jogador do Script PowerUpManager

```
1  public class PowerUpManager : MonoBehaviour
2  {
3      // Update is called once per frame
4      void Update() {
5          FrameInput = _input.FrameInput;
6          if (FrameInput.PowerUp) {
7              if (powerUp != null && manabar.HasEnoughMana(powerUp.manaCost)) {
8                  powerUp.Use();
9                  manabar.LoseMana(powerUp.manaCost);
10             }
11         }
12     }
13 }
```

2. Direção do Jogador

   Para calcular em qual direção a força de Knockback deverá ser aplicada, é utilizado uma classe auxiliar que possui um conjunto de funções que calculam a direção do jogador em relação aos pontos de referência da entidade em questão. O cálculo é feito por meio de pontos de referência, que indicam qual é a frente (*front*), as costas (*back*), o topo do inimigo (*topOfHead*) e os pés do inimigo (*bottomOfFeet*). Um exemplo de função em que é calculado se o jogador está acima do inimigo pode ser visto no trecho de código abaixo (O programa inteiro pode ser visto no Programa A.22 e no Programa A.23):

**Programa 3.8** Cálculo para detectar se o jogador está acima do inimigo

```
1      public bool IsPlayerUp(Transform player) {
2          Vector3 result = player.position - topOfHead.position;
3
4          if (result.y > 0)
5              return true;
6
7          return false;
8      }
```

A classe PlayerDirection também possui uma função que verifica a posição do jogador e retorna $-1$, 0 ou $+1$ para multiplicar o valor da força para a aplicação do Knockback. Como pode ser visto no trecho de código abaixo:

**Programa 3.9** Função que retorna o modificador para multiplicação da força de Knockback

```
1       public int WhichPlayerDirection(Transform player)
2       {
3           if (IsPlayerBack(player)) return 1;
4           else if(IsPlayerFront(player)) return -1;
5           else return 0;
6       }
7
```

3. Detector de colisão Para a lógica de aplicação de dano e detecção de pontos fracos alcançados de inimigos, é necessária uma lista com todas as entidades com a tag "Enemy"que foram atingidas. Para isso foi criado o script "PlayerWeaponColliderDetector", que é componente da arma do jogador. A arma, juntamente com o script, possui um colisor para cada ataque que a arma consegue realizar, assim ao ativar e desativar esses colisores, o script da arma mantém uma lista de todos os inimigos no alcance da arma na duração do ataque, como pode ser visto no trecho de código abaixo:

**Programa 3.10** Detector de colisão de inimigos na arma do jogador

```
1       public List<Collider2D> enemiesInWeaponRange;
2       [SerializeField] string[] hittableTags;
3
4       void Awake() {
5           enemiesInWeaponRange = new List<Collider2D>();
6       }
7
8       void OnTriggerEnter2D(Collider2D col) {
9           foreach(string t in hittableTags) {
10              if (col.gameObject.CompareTag(t)) {
11                  enemiesInWeaponRange.Add(col);
12              }
13          }
14      }
```

### Mapeamento de Controles

O gerenciamento eficiente de entrada é fundamental para proporcionar uma experiência de jogo envolvente, considerando tanto os dispositivos de teclado quanto os controles de joystick. Por isso, foi implementado utilizando o pacote *New Input System* nativo da Unity.

Para o mapeamento e definição dos controles padrões, foi utilizado um *Input Action Asset*, sendo um *asset* que contém as ações de input e suas teclas associadas e esquemas de controle (teclado ou controle), funcionando como um mapa de ações-tecla. Como é possível observar na figura abaixo:

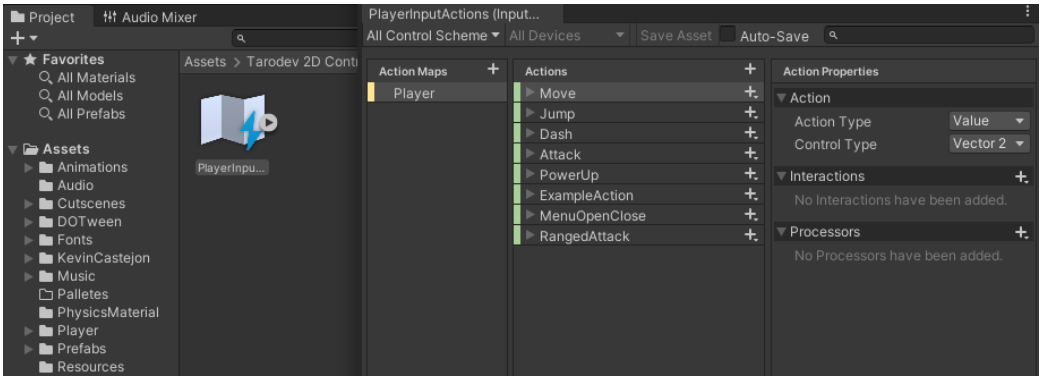



**Figura 3.5:** *Arquivo PlayerInputAction do jogador. Imagem elaborada pelos autores.*

O script "Input Manager", adaptado para o pacote Tarodev, é responsável por detectar quando a tecla de abrir o menu for pressionada, para então abrir a interface visual do menu. A interface visual utiliza o Canvas com botões nativos da Unity, que permite a navegação intuitiva do menu pelo jogador, já que é implementado pelo motor de jogos a navegação pelo mouse ou por joystick, ou por teclado.

O menu tem as opções de remapear os controles tanto do teclado quanto do *joystick*. Isto é feito a partir de um dos exemplos importados do pacote *New Input System*, que contém um script "RebindingActionUI"[1], que associa a um botão do menu, uma ação do *Input Action Asset* e substitui a tecla associada por uma nova escolhida.

Como o remapeamento utilizado é um exemplo de uso do pacote da Unity, ele acaba por faltar funções importantes esperadas que estão presentes em jogos modernos. Por conta disso, novas funcionalidades como sair da tela de remapeamento, mudar o título do botão segundo o nome da ação, etc., foram adicionadas. Porém, as duas mais importantes dentre elas são: verificar se a mesma tecla não está sendo usada em duas ações e persistência das teclas. Para solucionar este problema, o script foi modificado para adicionar as seguintes funções: CheckDuplicateBindings (ver programa abaixo), que varre toda a lista de teclas para verificar que nenhuma outra ação contenha a tecla passada, e para os dados persistentes foi criado um script "RebindSaveLoad"(ver Programa A.27), que salva os dados modificados quando o menu de remapeamento é fechado e os carrega quando o jogo inicia.

[1] O script completo pode ser encontrado em https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/api/UnityEngine.InputSystem.Samples.RebindUI.RebindActionUI.html

**Programa 3.11** Função Check Duplicate Bindings

```
1  private bool CheckDuplicatedBindings(InputAction action,
2  int bindingIndex, bool allCompositeParts = false) {
3      InputBinding newBinding = action.bindings[bindingIndex];
4      foreach (InputBinding binding in action.actionMap.bindings)
5      {
6          if (binding.action == newBinding.action) {
7              continue;
8          }
9          if (binding.effectivePath == newBinding.effectivePath) {
10             return true;
11         }
12     }
13
14     if (allCompositeParts) {
15         for(int i = 1; i < bindingIndex; i++) {
16             if (action.bindings[i].effectivePath == newBinding.overridePath) {
17                 return true;
18             }
19         }
20     }
21
22     return false;
23 }
```

### Controle de níveis

Para o controle dos níveis foram criados três sistemas, um permite selecionar o nível que se quer jogar, feito por um *hub* central, o outro controla o progresso do jogador no nível selecionado, o controle é feito via checkpoint, e o último é um sistema auxiliar ao *hub* para salvar o progresso do jogador no *hub* e carregar as cenas dos níveis. Esses sistemas são explicados em mais detalhes a seguir:

1. Hub: Uma característica comum em diversos jogos de plataforma é a presença de um sistema de níveis, frequentemente organizados mediante um *hub* central, que serve como um ponto de partida para diferentes fases ou áreas do jogo. Este é um espaço central que os jogadores exploram, e possui acessos a cada nível. O *hub* define aos jogadores a ordem em que devem enfrentar os níveis.

   Para implementar tal sistema, criamos uma cena com um *hub* com três níveis de teste. A cena possui um objeto "jogador", que serve apenas para indicar qual nível será selecionado. A cena de teste pode ser vista na figura a baixo:

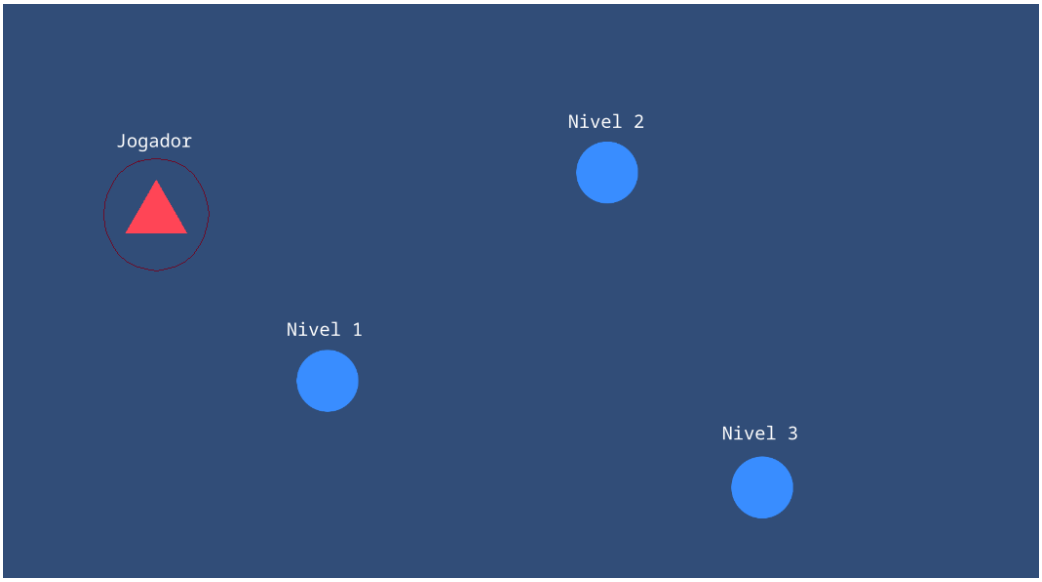



**Figura 3.6:** *Cena de teste para o sistema de Hub. Imagem elaborada pelos autores.*

O jogador só poderá se mover para níveis que já tiver acesso. O script "HUB_Player"é responsável por verificar quais níveis o jogador tem acesso, fazendo isso por meio da classe nativa da Unity "PlayerPrefs", que guarda dados permanentes, os dados são salvos como um ID, o qual é um inteiro. Com esse ID, o script verifica se o nível seguinte ao que o player está é menor que o número de níveis desbloqueados, se for, se move em direção aquele nível, senão, fica parado. Além disso, quando o nível é selecionado, muda a cena para a cena do nível selecionado (ver Programa A.24).

Além disso, o script "HUB_Position", colocado em cada objeto nível, guarda o nível anterior, o nível posterior, o nome da cena que deverá ir e o seu ID. Este script fornece as informações que o "HUB_Player"precisa para decidir se poderá se mover para um próximo nível, e qual cena deverá ser carregada. O script *HUB_Position* guarda as seguintes informações:

**Programa 3.12** Classe de dados HUB_Position.

```
1     public class HUBPosition : MonoBehaviour
2     {
3         public HUBPosition prev;
4         public HUBPosition next;
5         public string sceneName;
6         public int level;
7
8         // Start is called before the first frame update
9         void Start() { }
10
11        // Update is called once per frame
12        void Update() { }
13    }
```

2. Checkpoint

O sistema de *checkpoint* foi implementado no script "Checkpoint", que verifica se o

jogador colidiu com o objeto *checkpoint*, se colidiu, seu *animator* tocar a animação do checkpoint e comunica ao GameManager para mudar a posição do checkpoint para a sua posição, de modo que caso o jogador morra, ele volte para a posição do *checkpoint*. Como é possível observar no trecho de código a seguir:

**Programa 3.13** Classe de dados HUB_Position.

```
1       void OnTriggerEnter2D(Collider2D col) {
2                   if (col.CompareTag("Player") && !activated) {
3                           activated = true;
4                           PlayActivateAnimation();
5                           SetNewCheckpoint();
6                   }
7           }
```

3. SceneLoader e LevelEnder

   As classes *SceneLoader* e os scripts *LevelEnder* trabalham em conjunto para salvar os níveis desbloqueados utilizando *PlayerPrefs*. A classe *SceneLoader* define as funções para salvar os níveis desbloqueados e para carregar a próxima cena, como pode ser visto no código a seguir:

**Programa 3.14** Classe SceneLoader.

```
1  public class SceneLoader : MonoBehaviour {
2      public void LoadScene(string sceneName) {
3          SceneManager.LoadScene(sceneName);
4      }
5
6      public void LoadCurrentScene() {
7          LoadScene(SceneManager.GetActiveScene().name);
8      }
9
10     public void SaveLevel(int level) {
11         int l = PlayerPrefs.GetInt("LevelsUnlocked", 1);
12         if(l > level){
13             return;
14         }
15         else {
16             PlayerPrefs.SetInt("LevelsUnlocked", level + 1);
17         }
18     }
19 }
```

Os scripts que servem para terminar o nível utilizam a classe *SceneLoader* para chamar a função de salvar o nível e depois carregando a cena necessária, como pode ser visto no trecho de código a seguir:

**Programa 3.15** Script LevelEnder utilizando a classe SceneLoader.

```
1      public void FinishLevel() {
2                  sceneLoader.SaveLevel(levelNumber);
3                  sceneLoader.LoadScene(sceneToGo);
4      }
```

### Controle de som

O volume é controlado por uma estrutura da Unity chamada *Global Audio Mixer*, que foi dividido em três canais de som, o canal de efeitos sonoros (*SFX*), de música (*Music*) e canal principal (*Master*).

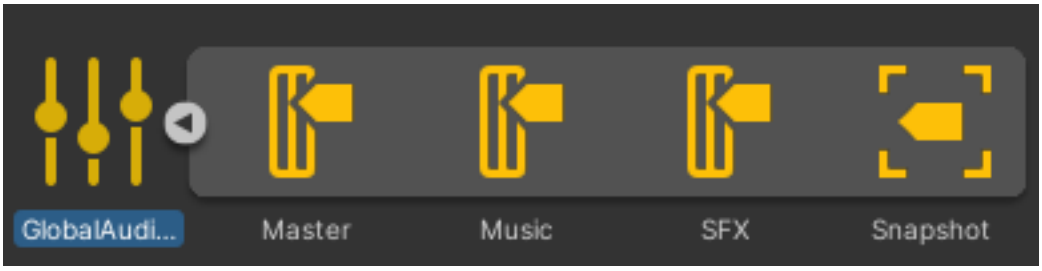



**Figura 3.7:** *Global Audio Mixer. Imagem elaborada pelos autores.*

O canal de efeitos sonoros é usado para tocar sons como o jogador pular, cair, som de morte de inimigos, ambientação, etc. O canal de música é reservado para música do nível, e o canal *Master* controla o som de efeitos sonoros e música ao mesmo tempo.

Para controlar cada canal individualmente, basta alterar a propriedade volume de cada canal.

### Câmera

A dinâmica da câmera é essencial para garantir que os jogadores tenham controle e compreensão adequados do ambiente. A adaptação da câmera às ações do jogador, como movimento, direção e mudança de ambiente, é crucial para manter a clareza e a visibilidade, evitando ângulos desfavoráveis ou que prejudiquem a jogabilidade.

Para a criação do sistema de câmeras, foi utilizado o pacote Cinemachine, que permite a criação e transição de diversos tipos de câmera. Cada câmera criada é filha de um objeto Collider, que controla até onde a câmera pode exibir a tela do jogo e ativa ou desativa cada câmera. Isso permite que cada região tenha uma câmera própria, que trata individualmente sobre como deve exibir o conteúdo na tela do jogador.

Cada região *Bound*, possui um script "BoundCamera", que referencia a câmera filha, e é responsável por ativá-la ou desativá-la caso o jogador entre em contato com a região determinado pelo colisor.

Dois tipos de câmera foram utilizados:

- Câmera fixa: exibe determinada região e não altera seu conteúdo de exibição conforme a movimentação do jogador.
- Câmera móvel: acompanha a movimentação do jogador, variando o conteúdo exibido consoante a posição do jogador. Possui o script "CameraFollowPlayer", que referenciando o controle do jogador ao ser iniciado, ajusta a posição da câmera com base na posição e na velocidade do jogador. Ele modifica a posição horizontal (X) da câmera conforme a direção do movimento do jogador. Se o jogador estiver caindo (velocidade Y negativa), a câmera ajusta sua altura (Y) para um valor menor, e se a velocidade vertical for menor que $-20$, ajusta o *damping* (amortecimento vertical) da câmera para dar a sensação de queda mais rápida. Essa atualização é feita utilizando o *LateUpdate()*. Além disso, a câmera é posicionada a frente do jogador para que ele possa visualizar o cenário a frente.

### Menu de configurações de som

No menu de configurações de som é possível alterar o volume do jogo. As configurações são persistentes, ou seja, se o jogador modificar uma configuração e reiniciar o jogo, as configurações permanecem.

Para salvar dados de som, é utilizado uma classe de dados chamada *SettingsData*, que guarda três valores de zero a um, onde cada valor se refere ao volume dos canais *Master*, *SFX* e *Music*.

Ao usuário modificar os valores dos *sliders* no menu, é chamado a função ChangeVolume que atualiza o valor guardado na classe de dados. Ao usuário sair do menu de configuração, os dados da classe SettingsData são salvos em um arquivo binário de caminho persistente. Ao abrir o jogo, o script GameManager aplica os valores guardados nesse arquivo. O caminho persistente é providenciado pela Unity através da variável *Application.persistentDataPath* sendo garantido sempre ser o mesmo em toda execução do jogo.


**Figura 3.8:** *Tela de configuração de som. Imagem elaborada pelos autores.*

### Menu de configurações de teclas de atalho

No menu de configurações é possível alterar as teclas de atalho para controle do jogador. As alterações são persistentes, ou seja, se o jogador modificar uma configuração e fechar e abrir o jogo, as configurações permanecem.

Para o controle das teclas de atalho e da permanência de dados é utilizado um pacote chamado *Input System* que é um pacote nativo na Unity, que guarda todas as configurações em uma estrutura chamada *InputActionAsset* e salva essa estrutura como um arquivo Json. A estrutura é salva quando a tela de menu é fechada, sendo carregada do arquivo quando a tela de menu é iniciada.

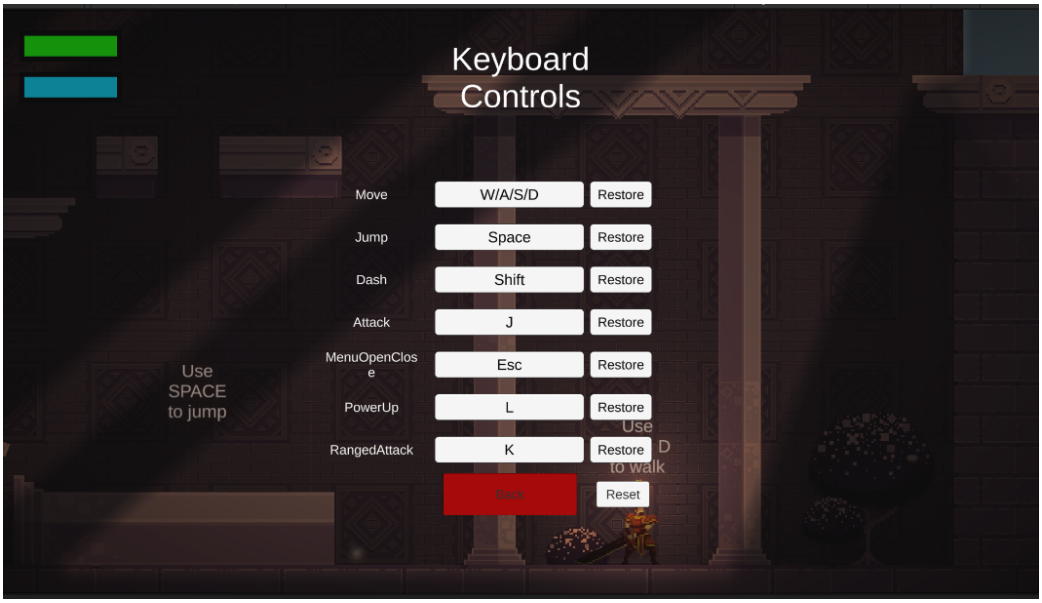



**Figura 3.9:** *Tela de configuração de controle. Imagem elaborada pelos autores.*

### 3.1.5 Diagrama da arquitetura de classes

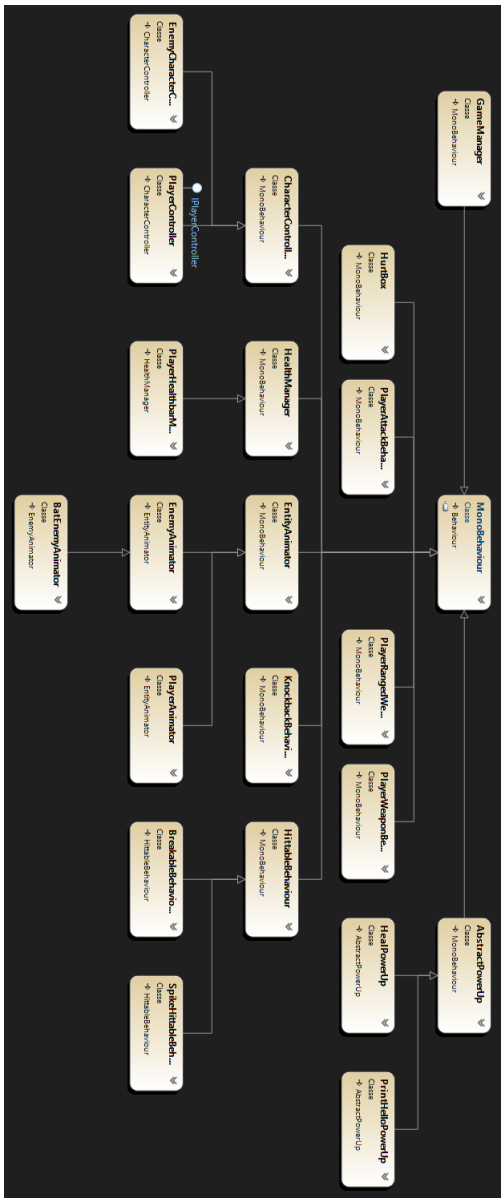



**Figura 3.10:** *Diagrama de Classes do jogo. Imagem elaborada pelos autores.*

## 3.2 Implementação de Behavior Trees (BT) e Finite State Machines (FSM)

Para a implementação de comportamentos complexos de entidades do jogo, foi utilizado um conjunto de duas estruturas de dados, *BehaviorTrees* (BT) e *Finite State Machines* (FSM). Isso permite flexibilidade para implementação de comportamentos variados, por serem combinadas as vantagens e desvantagens de FSM e BT conforme a necessidade. Para inimigos complexos (ver Seção 1.2), foi utilizado inteiramente FSMs e para a implementação de chefes de nível (ver Seção 1.2) foi utilizado uma combinação de FSMs e BTs.

### 3.2.1 Implementação inimigos complexos

Cada inimigo complexo possui uma FSM que determina seu comportamento. Na implementação, cada estado representa uma ação a ser executada e cada transição representa as condições para executar tal ação. Dessa forma é possível determinar o comportamento inteiro do inimigo em apenas um script, definindo todas as funções no mesmo, e depois chamando elas nas transições e estados. Cada estado possui três funções que podem ser chamadas, *onEnter* que é chamado quando o estado é ativado, *onExit* que é chamado ao se sair do estado, e *onLogic* chamado em *loop* enquanto o estado está ativo.

#### Inimigo Teste

Primeiramente foi criado um inimigo de teste, para determinar a viabilidade do método utilizado. O inimigo tem como funcionalidade básica seguir o jogador, para fazer isso ele espera o jogador entrar em seu campo de visão. Quando o jogador entra em seu campo de visão, ele o segue. Se o jogador sair do campo de visão, o inimigo continua até chegar na última posição do jogador registrada, para então virar para o outro lado.

Foi definido uma FSM com três estados, *Idle*, *Active* e *Flip* e três transições. O diagrama do estado pode ser visto a seguir:

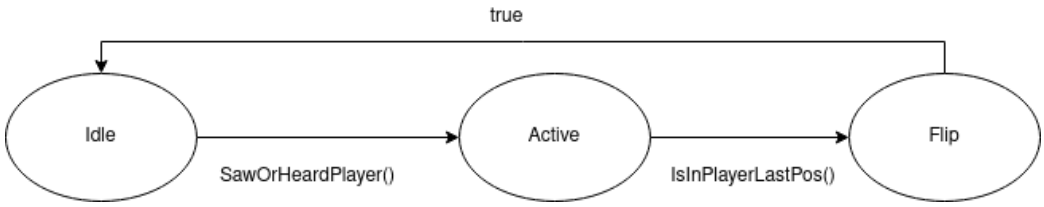



**Figura 3.11:** *Diagrama inimigo teste. Imagem elaborada pelos autores.*

O estado *Idle* não faz nenhuma ação, apenas espera o inimigo detectar o jogador. Ao jogador ser detectado é feita a transição do estado *Idle* para o estado *Active*, em que o inimigo segue o jogador em linha reta. Ao inimigo alcançar a última posição vista do jogador é realizado a transição para o estado *Flip*, que inverte o sentido inimigo e logo em seguida é feita a transição para o estado *Idle* novamente.

A definição da máquina de estados pode ser vista no código a seguir:

**Programa 3.16** Definição FSM do inimigo teste.

```
1  void Start()
2  {
3      fsm = new StateMachine(this);
4
5      fsm.AddState("Idle");
6      fsm.AddState("Active",
7                   onEnter: (state) => animator.unityAnimator.
8                       SetBool("Following", true),
9                   onExit: (state) => {
10                      thisEnemyCharacterController.SetSpeed(Vector2.zero);
11                      animator.unityAnimator.SetBool("Following", false);
12                  },
13                  onLogic: (state) => FollowPlayer());
14     fsm.AddState("Flip", onEnter: (state) => thisEnemy.Flip());
15
16     fsm.AddTransition("Idle",
17                       "Active",
18                       t => SawOrHeardPlayer());
19
20     fsm.AddTransition("Active", "Flip", t => IsInPlayerLastPos());
21     fsm.AddTransition("Flip", "Idle", t => true);
22
23     fsm.SetStartState("Idle");
24     fsm.Init();
25 }
26
27 bool IsInPlayerLastPos() { ... }
28 bool SawOrHeardPlayer() { ... }
29 void FollowPlayer() { ... }
30 float Step() { ... }
31
```

### Inimigo *Piranha Plant*

O inimigo propriamente implementado, é o *PiranhaPlant* (ver Figura 3.13), que possui o comportamento básico de atacar o jogador quando ele está próximo o suficiente. A declaração da máquina de estados pode ser vista no código a seguir:

**Programa 3.17** Definição FSM do inimigo *Piranha Plant*.

```
1  void Start()
2  {
3      playerTransform = GameObject.FindGameObjectWithTag("Player").transform;
4
5      fsm = new StateMachine(this);
6
7      fsm.AddState("Idle");
8      fsm.AddState("Active");
9      fsm.AddState("Attacking",
10                 onEnter: (state) => {
11                 animator.unityAnimator.SetTrigger("Attack");
12                 } );
13     fsm.AddState("Flip",
14                 onEnter: (state) => thisEnemy.Flip());
15
16     fsm.AddTransition("Idle", "Active", t => SawOrHeardPlayer());
17     fsm.AddTransition("Active", "Attacking", t => IsPlayerClose());
18     fsm.AddTransition("Active", "Flip", t => IsPlayerBehind());
19     fsm.AddTransition("Attacking", "Active", t => IsAttackingFinished());
20     fsm.AddTransition("Flip", "Active", t => true);
21
22     fsm.Init();
23 }
24
25 bool SawOrHeardPlayer() { ... }
26
27 bool IsPlayerClose() { ... }
28
29 bool IsPlayerBehind() { ... }
30
31 bool IsAttackingFinished() { ... }
32
33 public void EndOfAttackAnimation() { ... }
```

Como é possível observar, são definidos quatro estados, com cinco transições. O estado *Idle* não faz nenhuma ação, e o inimigo sai desse estado quando ele detecta o jogador. Após detectar o jogador, o inimigo vai do estado *Idle* para o estado *Active*, e desse estado o inimigo muda para os estados de ação.

As ações desse inimigo são simples, sendo representadas por dois estados, *Attacking* e *Flip*, como o nome sugere, as duas ações que esse inimigo realiza é virar para a direção do jogador, e atacar quando ele está perto o suficiente.

O diagrama dessa máquina de estados se encontra a seguir:

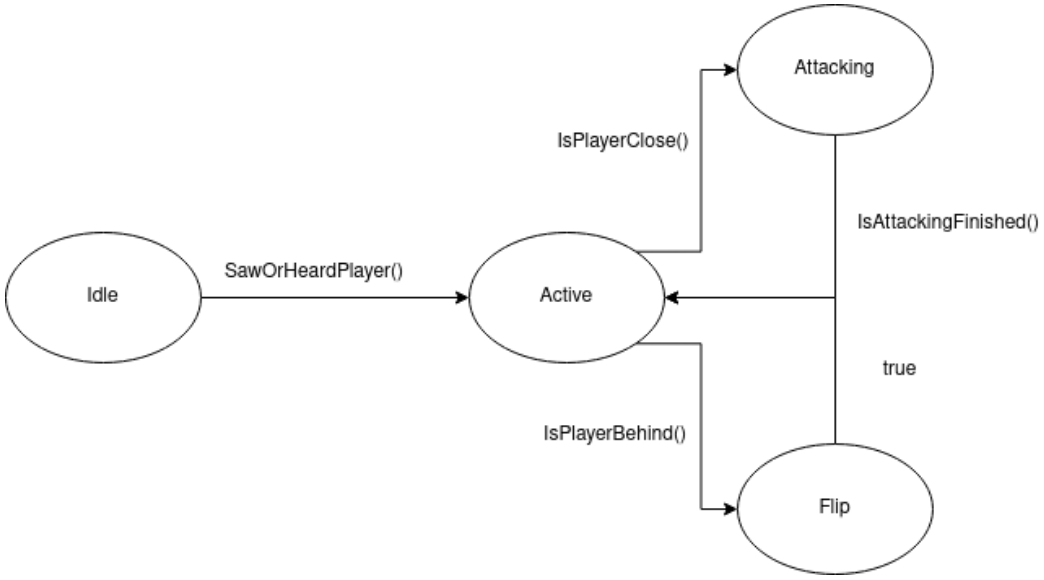



**Figura 3.12:** *Diagrama FSM Piranha Plant. Imagem elaborada pelos autores.*

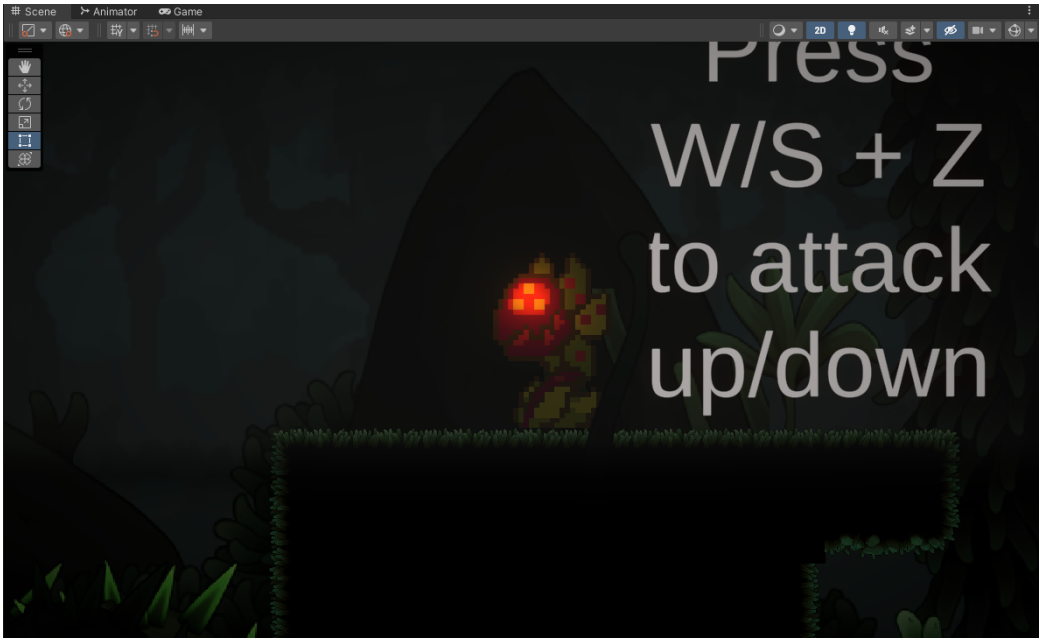



**Figura 3.13:** *Inimigo Piranha Plant. Imagem elaborada pelos autores.*

### 3.2.2 Implementação de inimigos chefes

Cada inimigo chefe possui uma FSM e uma BT para definir seu comportamento, isso foi feito para utilizar as vantagens de FSMs, as quais são a resposta rápida para mudanças de variáveis. Devido ao comportamento de BTs, não há resposta instantânea para a mudança de alguma variável, já que as condições verificadas são relacionadas somente com o nó que está sendo executado no momento, enquanto em uma FSM é possível adicionar uma transição de qualquer estado para qualquer outro estado.

Devido à biblioteca utilizada para implementar as BTs, é necessário declarar a árvore em forma de texto, onde é possível declarar nós nativos da biblioteca, ou definir novos nós de controle de fluxo (condições) e novos nós de ação. Como é possível observar no trecho de código a seguir:

**Programa 3.18** Exemplo declaração BT por código.

```
1  battleBehaviourTree = new BehaviorTreeBuilder(gameObject)
2      .Sequence()
3          .RepeatUntilSuccess()
4              .Selector()
5                  .Sequence()
6                      .Condition(() => {return SawOrHeardPlayer();})
7                      .Do(() => { return TaskStatus.Success; })
8                  .End()
9                  .Sequence()
10                     .Condition(() => {return !SawOrHeardPlayer();})
11                     .Flip()
12                 .End()
13             .End()
14         .End()
15     .End().Build();
```

Como é possível observar, é utilizado uma classe chamada *BehaviorTreeBuilder* para declarar os nós da árvore. Na biblioteca é possível definir ações e condições no decorrer da declaração da árvore, sem a necessidade de funções intermediárias, como os nós *Condition()* e *Do()* que aceitam a condição e o código a ser executado como argumento da declaração do nó.

Também é possível definir nós e condições customizados, assim como o nó *Flip()*, que inverte o sentido dos inimigos. A declaração de nós customizados precisa ser feita em no mínimo duas classes separadas, uma para determinar como o nó deve ser chamado, e a outra classe deve determinar a funcionalidade propriamente dita do novo nó. Um exemplo de um novo nó pode ser visto no código a seguir:

**Programa 3.19** Exemplo adição de nós customizados em uma BT.

```
1  public static class BTExtension {
2      public static BehaviorTreeBuilder Flip
3        (this BehaviorTreeBuilder builder, string name = "Flip") {
4          return builder.AddNode(new Flip() {
5              Name = name,
6          });
7      }
8  }
9
10 public class Flip : ActionBase {
11     Enemy thisEnemy;
12
13     protected override void OnStart()
14     {
15         thisEnemy = Owner.GetComponent<Enemy>();
16         thisEnemy.Flip();
17     }
18
19     protected override TaskStatus OnUpdate ()
20     {
21         return TaskStatus.Success;
22     }
23
24     protected override void OnExit()
25     {
26     }
27 }
```

### Inimigo chefe teste

Para o teste dessas funcionalidades, foi criado um inimigo chefe de teste, em que a FSM definida apenas troca, de forma instantânea, de um estado *Idle* do inimigo para um estado ativo. O inimigo pode ser visto na figura a seguir:

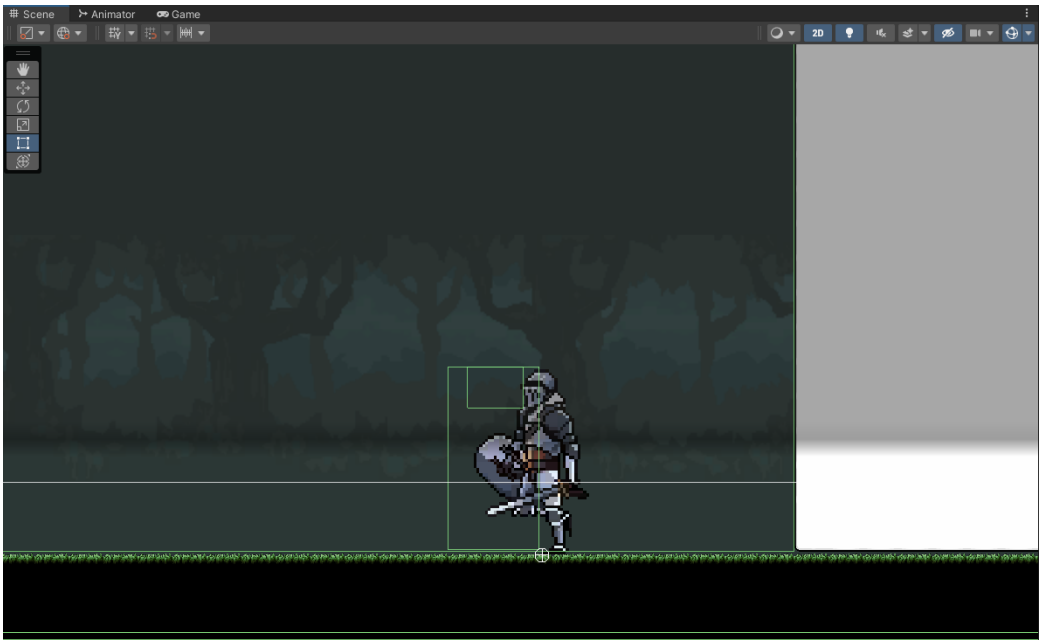



**Figura 3.14:** *Inimigo chefe teste. Imagem elaborada pelos autores.*

O estado *Idle* da FSM é definido pela execução de uma BT vazia, enquanto que o estado ativo determina as ações do inimigo. As ações definidas pelo modo ativo do inimigo chefe de teste são as seguintes: andar para frente, andar para trás, ataque parado e ataque correndo. Também é definido a ação de inverter o sentido (esquerda ou direita) do inimigo chefe.

A biblioteca utilizada, *FluidBT*, possui um visualizador das árvores criadas, o comportamento descrito acima pode ser visto no diagrama a seguir:

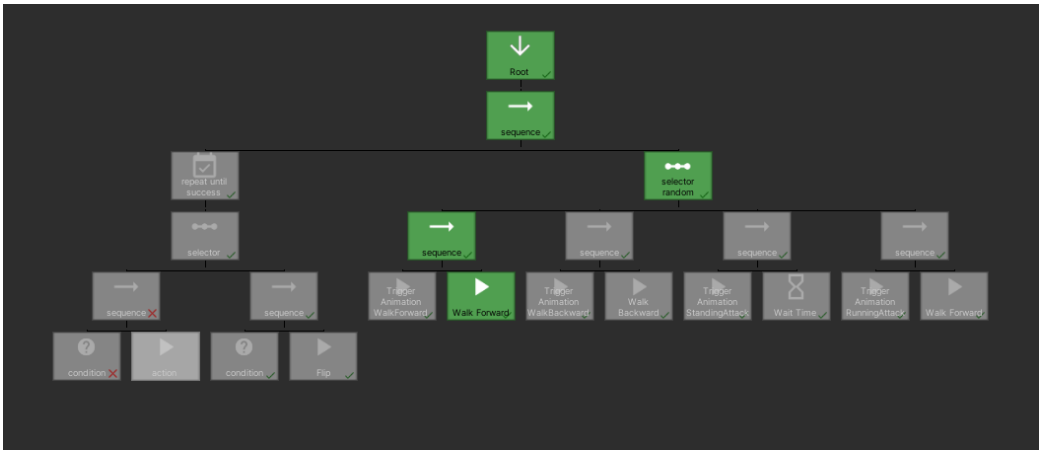



**Figura 3.15:** *Diagrama BT do inimigo chefe teste. Imagem elaborada pelos autores.*

O diagrama anterior é equivalente à declaração por código a seguir:

**Programa 3.20** Declaração BT inimigo teste.

```
1  void DefineBattleBT()
2  {
3      battleBehaviourTree = new BehaviorTreeBuilder(gameObject)
4          .Sequence()
5              .RepeatUntilSuccess()
6                  .Selector()
7                      .Sequence()
8                          .Condition(() => {return SawOrHeardPlayer();})
9                          .Do(() => { return TaskStatus.Success; })
10                     .End()
11                     .Sequence()
12                         .Condition(() => {return !SawOrHeardPlayer();})
13                         .Flip()
14                     .End()
15                 .End()
16             .End()
17
18             .SelectorRandom()
19                 .Sequence()
20                     .TriggerAnimation("WalkForward")
21                     .WalkForward(3.5f, 5)
22                 .End()
23                 .Sequence()
24                     .TriggerAnimation("WalkBackward")
25                     .WalkBackward(3.5f, 5)
26                 .End()
27                 .Sequence()
28                     .TriggerAnimation("StandingAttack")
29                     .WaitTime(1.0f)
30                 .End()
31                 .Sequence()
32                     .TriggerAnimation("RunningAttack")
33                     .WalkForward(7, 7)
34                 .End()
35             .End()
36         .End()
37         .Build();
38 }
39
```

### Inimigo chefe *Cursed Tree*

O inimigo chefe propriamente implementado é chamado de *Cursed Tree* (Figura 3.16), que foi utilizado no fim da fase tutorial do jogo. O inimigo também usa a mecânica de ponto fraco, em que se o jogador conseguir acertar o topo da cabeça do inimigo ele será derrotado instantaneamente.

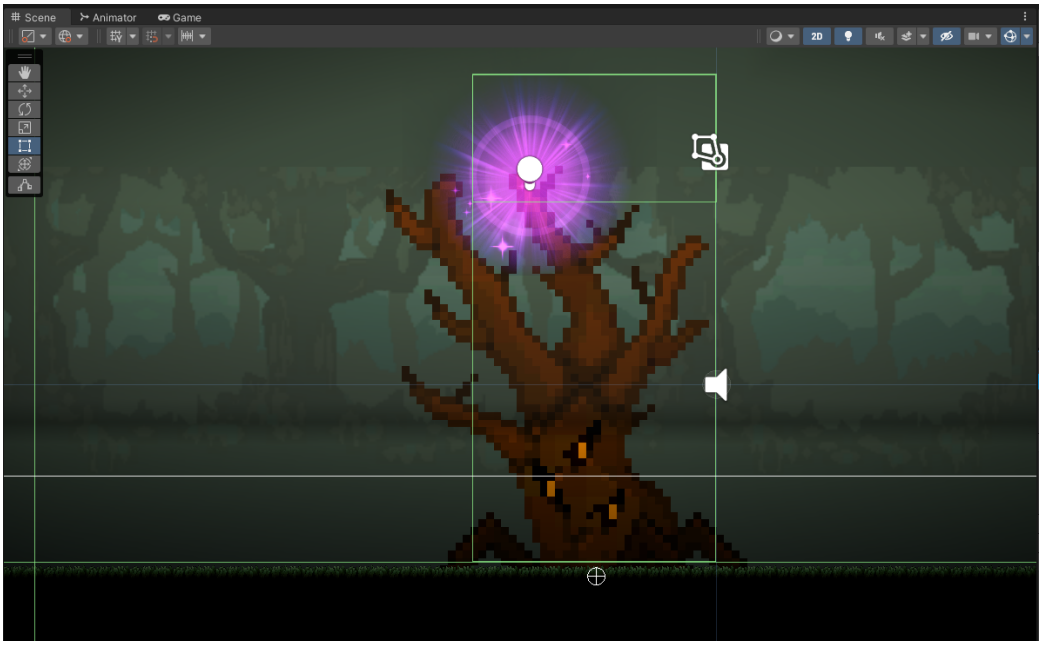



**Figura 3.16:** *Inimigo chefe do tutorial. Imagem elaborada pelos autores.*

O inimigo possui o comportamento básico de escolher de forma aleatória entre três ataques. Os ataques são *Stomp*, *Constrict* e *Sweep*. Para a implementação desses ataques, foram criados nós customizados para tais. Como pode ser visto no diagrama a seguir:


**Figura 3.17:** *Diagrama inimigo chefe tutorial. Imagem elaborada pelos autores.*

O diagrama anterior é equivalente à declaração por código a seguir:

**Programa 3.21** Declaração BT inimigo teste.

```
1  mainBehaviour = new BehaviorTreeBuilder(gameObject)
2  .Sequence()
3    .WaitTime(1.1f)
4    .SelectorRandom()
5      .Sequence()
6        .Condition(() => {return IsPlayerInCorrectDistance();})
7        .TriggerAnimation("Sweep")
8      .End()
9      .Sequence()
10       .TriggerAnimation("Stomp")
11       .WaitTime(0.35f)
12       .Stomp(spikePrefab, spikeUndergroundReferece)
13     .End()
14     .Sequence()
15       .TriggerAnimation("Constrict")
16       .WaitTime(0.7f)
17       .Constrict(constrictPrefab, groundLevelReference.transform)
18     .End()
19   .End()
20 .End().Build();
21
```

## 3.3 Outros Códigos

Até aqui foram apresentados os principais códigos, e seus principais auxiliares. Porém alguns outros códigos, com impacto menor no projeto podem ser encontrados em https://github.com/JohnnyLima67/MAC0499.

# Capítulo 4

# Resultados

O desenvolvimento do jogo Sculpted Fate atingiu seus objetivos centrais, proporcionando uma experiência de jogo completa e funcional. A integração bem-sucedida dos sistemas, incluindo a programação de todos os sistemas iniciais requeridos, design, arte e trilha sonora, resultou em um produto coeso que reflete parcialmente a visão inicial do projeto. As mecânicas de jogo foram implementadas com sucesso, atendendo as principais demandas propostas.

As ideias implementadas, que mantiveram-se as mesmas desde o escopo inicial, foram:

- Sistema de inimigos, com os três tipos de inimigos propostos (simples, complexo e chefe). Com três inimigos e um chefe. (ver Subseção 3.1.3)
- Sistema de câmera, com a possibilidade de ter diversas câmeras com diversas funcionalidades diferentes. (ver Subsubseção 3.1.4)
- Obstáculos de plataforma, com plataformas fixas, semi-sólidas e móveis, além de buracos e buracos com espinhos. (ver Subseção 1.2.1)
- *Level design* com nível que se estende horizontalmente, com a ambientação que contribui para a história. (ver Seção 1.2)
- Sistema de combate, com ataque corpo a corpo, em quatro direções, e ataque a distância. (ver Subsubseção 3.1.2, Subsubseção 3.1.2)
- Sistema de vida, com cada inimigo e o jogador possuindo certa quantidade de vida. (ver Programa A.1)
- Sistema de *Knockback* (ver Programa A.5)
- Sistema de movimentação, uma movimentação bem controlada sendo possível controlar aspectos como altura do pulo, gravidade, inércia, etc. (ver Programa A.4)
- Sistema de *Power Up*, com um powerUp implementado. (ver Subseção 3.1.4)
- Sistema de *Checkpoint* utilizado no tutorial. (ver Subsubseção 3.1.4)

- Sistema de salvar dados de configuração e mapeamento de teclas. (ver Subsubseção 3.1.4
- Sistema de animações (ver Subseção 3.1.1)

Também foram implementados sistemas que acabaram não sendo utilizados no jogo final, dentre eles estão:

- Mapa do jogo: um mapa no estilo de *Super Mario World*
- Sistema de *cutscenes*
- Sistema de troca de armas
- Sistema de paredes quebráveis
- Sistema de plataformas móveis
- Sistema de salvar progresso de níveis

Apesar disso, houve objetivos abandonados ou não atingidos. Com relação à ideia inicial do projeto, o jogo obtido foi menor em conteúdo. Os níveis, que inicialmente seriam cinco, foram reduzidos ao nível tutorial. Em relação às mecânicas, a principal mecânica de ataques com precisão foi descartada, por conta de nenhuma arte atender os requisitos necessários, além da ideia dos ataques recuperarem uma certa quantidade de mana ser descartada.

Outro ponto majoritário que mudou foi o sistema de vida. Inicialmente pensado para ser por quantidade de *hits* tomados, mas acabou se transformando em uma barra de vida, de modo que o jogador tem uma quantidade definida de vida.

### 4.0.1 Feedbacks

Durante o processo de desenvolvimento, um ponto-chave foi coletar feedbacks da comunidade de jogadores. Essas avaliações forneceram sugestões importantes sobre a jogabilidade, a experiência do usuário e possíveis melhorias. As respostas positivas destacaram os pontos fortes do jogo, enquanto as sugestões construtivas orientaram ajustes finais para otimizar a qualidade geral. O formulário com os feedbacks podem ser encontrados no Apêndice B.

Os principais pontos positivos recebidos pela avaliação dos usuários foram:

- O aspecto artístico do jogo, como arte dos personagens, cenários, animações, efeitos sonoros e visuais, e iluminação.
- Existência de mecânicas como *coyote jump* e a diversidade de movimentação.

Já as principais críticas foram:

- O pulo foi considerado muito baixo, dificultando certas passagens de plataforma.
- Câmera muito próxima do jogador.
- Ataque à distância não dá a sensação de impacto e é desbalanceado.

- A mecânica de *Bounce* nos inimigos se mostrou difícil de realizar para a maioria dos jogadores
- Sistema de checkpoint pareceu injusto, uma que quando o jogador cai em um buraco, ele morre e recomeça a fase, tornando morrer muito punitivo, uma vez que os pulos são difíceis de realizar. Uma sugestão dada foi o jogador voltar antes de realizar o pulo.
- Falta de itens no menu, como opção de sair do jogo.
- Falta de informação visual para os ataques do inimigo chefe, dificultando prever aonde eles surgirão.
- Arte inconsistente.

# Capítulo 5

# Discussão

Apesar de apresentar uma menor quantidade de conteúdo em relação ao jogo inicial, os objetivos centrais foram considerados atingidos pelos desenvolvedores, uma vez que o foco era a implementação e interconexão de todos os sistemas inicialmente propostos. A versão final se diferencia principalmente por apresentar um escopo reduzido em relação ao original, uma vez que houve diversos contratempos e dificuldades no processo de desenvolvimento, dentre eles podem ser citados:

- A falta de artistas no projeto resultou na maioria das artes serem extraídas da internet, por esse motivo, muitas das mecânicas inicialmente propostas tiveram de ser modificadas;
- A elaboração de sistemas facilmente expansíveis e que pudessem se comunicar entre si provou-se mais complexa do que inicialmente pensado, por conta disso, não foi possível implementar novos níveis, inimigos e *power-ups*, uma vez que o foco foi dedicado a fazer o sistema base funcionar corretamente.
- Algumas mecânicas, como a personalização de armas, não chegaram a entrar no corte final do jogo, seja por não funcionarem tão bem com o conjunto do jogo completo, seja por conta do cronograma apertado. Isso acabou prejudicando o projeto, uma vez que muitos destes sistemas foram implementados, mas não acoplados no jogo.

A funcionalidade dos sistemas significa que possíveis novas implementações e expansões poderão ser feitas no futuro, de modo que seja necessária pouca ou nenhuma reestruturação da arquitetura do projeto.

Os *feedbacks* dados pela comunidade também serviram para fazer ajustes no jogo, tanto na correção de erros na *gameplay* quanto na experiência do jogador em geral. Uma das principais e fundamentais mudanças que ocorreram foi na movimentação do jogador, uma vez que a precisão dos controles em um jogo de plataforma impactam a experiência do jogador. Vale ressaltar que essas mudanças foram feitas alterando valores de variáveis, não sendo necessário modificar o código. Assim como a movimentação do jogador, muitos dos pontos abordados pelos jogadores podem ser resolvidos sem mudanças na estruturação do código, mas apenas alterando valores de variáveis.

# Capítulo 6

# Conclusão

O desenvolvimento do jogo "Sculpted Fate" proporcionou uma visão abrangente do processo de criação de jogos *indie* centrado na programação e integração de sistemas. Os resultados obtidos, mesmo diante de contratempos, representam uma jornada significativa e uma base sólida para futuras iterações.

Ao atingir os objetivos centrais, o jogo demonstrou uma integração bem-sucedida dos sistemas essenciais, englobando programação, design, arte e trilha sonora. As mecânicas de jogo refletiram as principais demandas da ideia original, oferecendo uma experiência completa e funcional aos jogadores.

A coleta de *feedbacks* da comunidade desempenha um papel crucial no aprimoramento de qualquer jogo, evidenciando a importância da participação da comunidade no refinamento do produto final.

Na discussão dos desafios enfrentados, é evidente que questões como a falta de artistas no projeto e o cronograma apertado, impactaram o escopo original. A falta de artistas, no entanto, se revela um problema maior, uma vez que se já há uma dificuldade de atender as expectativas que o jogo tem para um escopo relativamente pequeno, para um jogo maior, com mais fases, se mostrará algo inviável, uma vez que os elementos visuais terão mais dificuldade de se manterem coesos durante a totalidade do jogo, fazendo com que o jogo não tenha identidade.

No entanto, a funcionalidade dos sistemas estabelecidos oferece uma base robusta para futuras expansões, exigindo pouca ou nenhuma reestruturação. Isso mostra que é importante reconhecer inicialmente as limitações temporais e de recursos, sendo fundamentais para planejar um projeto de maneira realista.

Em última análise, o projeto "Sculpted Fate"não apenas entregou um produto funcional, mas também proporcionou aprendizados valiosos sobre a complexidade do desenvolvimento de jogos *indie* centrados em sistemas, e expôs os principais desafios que um desenvolvedor *indie* pode enfrentar durante o desenvolvimento de seu projeto.

# Capítulo 7

# Bibliografia recomendada

Neste capítulo são apresentadas algumas fontes de estudo para além do que foi utilizado na construção do jogo até o momento da publicação deste projeto. Trata-se de uma lista de artigos e tutoriais que podem ser úteis para qualquer pessoa interessada em continuar desenvolvendo este projeto.

- **Mecânicas de Movimento 2D:** como mencionado anteriormente, foi utilizado um pacote de movimento 2D com diversas mecânicas já implementadas. No entanto, futuros desenvolvedores podem estar interessados em expandir esse conjunto e implementar novas mecânicas e, consequentemente, diversificar ainda mais o *level-design*. Este é um artigo que apresenta múltiplas mecânicas 2D com exemplos de jogos clássicos que as implementam e pode despertar o interesse de futuros desenvolvedores: https://elliotcox.com/2d-movement-mechanics.
- **Agentes complexos:** apesar do inimigo chefe no tutorial do projeto ser bem simples, o código que contém sua lógica pode ser difícil de assimilar à primeira vista. Com a pretensão de criar chefes mais complexos, algumas ferramentas mais profissionais podem ser interessantes para desenvolvedores futuros. Recomenda-se estudar a biblioteca Behavior Designer, que possui uma API mais robusta para árvores de comportamento em Unity, além de uma interface gráfica que facilita bastante a construção de árvores maiores para agentes mais complexos. Um excelente tutorial pode ser acompanhado por este vídeo no YouTube: https://www.youtube.com/watch?v=X7VwAGvAOIw.
- **Sistemas de progressão e recompensa:** algo que precisaria ser implementado no jogo para deixá-lo, de fato, completo seria implementar mais sistemas que fazem os jogares sentirem que estão progredindo e os recompensam ao longo do tempo. Nesse caso, é necessário que desenvolvedores futuros expandam o escopo inicial deste projeto. Algumas bases de *game-design* clássicas e populares podem ser encontradas a seguir:
  - RPGs (Role Playing Games)
  - Jogos Roguelike
  - Jogos sem fim, por exemplo: Minecraft, Terraria, Tetris e Stardew Valley

# Apêndice A

# Programas

**Programa A.1** Classe HealthManager.

```
1       [SerializeField] protected float maxHealth = 200f;
2       [SerializeField] protected float health = 200f;
3       [SerializeField] protected float damageInterval = 0.5f;
4       protected float passedTime = -1.0f;
5
6       public virtual void Start() {
7           health = Mathf.Clamp(health, 0, maxHealth);
8       }
9
10      public virtual void FixedUpdate() {
11          if (Mathf.Sign(passedTime) == 1) {
12              passedTime += Time.deltaTime;
13
14              if (passedTime >= damageInterval)
15                  passedTime = -1.0f;
16          }
17      }
18
19      public bool CanTakeDamage() { return (Mathf.Sign(passedTime) == -1); }
20
21      public virtual void TakeDamage(float damage) {
22          if (Mathf.Sign(passedTime) == -1) {
23              passedTime = +0.0f;
24              if (damage > 0)
25                  health = Mathf.Max(health - damage, 0);
26          }
27      }
28
29      public virtual float CurrentHealth(){ ... }
30      public virtual void Heal(float healing) { ... }
31      public virtual bool isDead() { ... }
32      public virtual float GetDamageInterval() { ... }
```

**Programa A.2** Classe EntityAnimator.

```
1  public class EntityAnimator : MonoBehaviour
2  {
3      public Animator unityAnimator;
4
5      public virtual IEnumerator PlayTakeDamage() { yield break; }
6
7      public virtual IEnumerator PlayTakeCriticalDamage() { yield break; }
8
9      public virtual IEnumerator PlayDie() { yield break; }
10 }
```

**Programa A.3** Classe EnemyAnimator.

```
1  public class EnemyAnimator : EntityAnimator
2  {
3      [SerializeField] SpriteRenderer playerSprite;
4      [SerializeField] HealthManager healthbarManager;
5      private bool isInCriticalAnimation = false;
6
7      public override IEnumerator PlayTakeDamage() {
8          isInCriticalAnimation = true;
9          float interval = healthbarManager.GetDamageInterval();
10
11         playerSprite.DOColor(Color.red, interval / 2);
12         yield return new WaitForSeconds(interval / 2);
13         playerSprite.DOColor(Color.white, interval / 2);
14
15         isInCriticalAnimation = false;
16         yield break;
17     }
18
19     public override IEnumerator PlayTakeCriticalDamage() {
20         isInCriticalAnimation = true;
21         float interval = healthbarManager.GetDamageInterval();
22
23         playerSprite.DOColor(Color.magenta, interval / 2);
24         gameObject.transform.DOShakePosition(0.8f, 2.0f, 15, 90.0f, false, true);
25         yield return new WaitForSeconds(interval / 2);
26         playerSprite.DOColor(Color.white, interval / 2);
27
28         isInCriticalAnimation = false;
29         yield break;
30     }
31
32     public override IEnumerator PlayDie() {
33         unityAnimator.SetTrigger("Dead");
34
35         float alphaVal = playerSprite.color.a;
36
37         while (alphaVal > 0) {
38             alphaVal -= 0.01f * Time.deltaTime * 15;
39             alphaVal = Mathf.Clamp01(alphaVal);
40             Color tmp = playerSprite.color;
41             tmp.a = alphaVal;
42             playerSprite.color = tmp;
43             yield return null; // Wait for the next frame
44         }
45
46         Destroy(gameObject);
47         yield break;
48     }
49
50     public bool IsInCriticalAnimation(){...}
51 }
```

**Programa A.4** Classe CharacterController.

```
1     void Awake() {
2         currentFallingSpeed = Vector2.zero;
3     }
4
5     void Update() {
6         Collider2D[] col = Physics2D.OverlapCircleAll(bottomOfFeet.position,
7                                                       groundRadius,
8                                                       groundLayer);
9
10        if (col.Length > 0 && !isGrounded)
11            isGrounded = true;
12        else if (isGrounded) isGrounded = false;
13    }
14
15    public virtual void FixedUpdate() {
16        if (!isGrounded) {
17            currentFallingSpeed = Vector2.MoveTowards(currentFallingSpeed,
18                                        new Vector2(0, -maxFallingSpeed),
19                                        gravity * Time.fixedDeltaTime);
20
21            _rb.velocity = thisSpeed + thisExternalSpeed + currentFallingSpeed;
22        }
23        else {
24            _rb.velocity = thisSpeed + thisExternalSpeed;
25            currentFallingSpeed = Vector2.zero;
26        }
27
28        thisExternalSpeed = Vector2.MoveTowards(thisExternalSpeed,
29                                                Vector2.zero,
30                                                thisDrag * Time.fixedDeltaTime);
31    }
32
33    public virtual void BeforeStartKnockback(){ ... }
34    public virtual void AfterEndKnockback(){ ... }
35    public virtual void ApplyForce(Vector2 force){ ... }
36    public virtual void ApplySpeed(Vector2 speed){ ... }
37    public virtual void EndExternalSpeed(){ ... }
38    public virtual void SetSpeed(Vector2 speed){ ... }
39    public virtual void SetHorizontalSpeed(Vector2 speed){ ... }
40    public virtual void SetVerticalSpeed(Vector2 speed){ ... }
41
```

**Programa A.5** Classe KnockBack Behavior.

```
1  public class KnockbackBehaviour : MonoBehaviour {
2      [SerializeField] private CharacterController characterController;
3
4      private float knockBackDuration = 1.0f;
5      private float currentTime = 0.0f;
6      private bool isInKnockback = false;
7
8      public virtual void ApplyKnockback(Vector2 force, float duration) {
9          characterController.BeforeStartKnockback();
10         characterController.ApplyForce(force);
11         knockBackDuration = duration;
12         isInKnockback = true;
13     }
14
15     public virtual void EndKnockback() {
16         characterController.AfterEndKnockback();
17         isInKnockback = false;
18     }
19
20     void Update() {
21         if (!isInKnockback) return;
22
23         currentTime += Time.deltaTime;
24
25         if (currentTime >= knockBackDuration) {
26             EndKnockback();
27             currentTime = 0.0f;
28         }
29     }
30 }
```

**Programa A.6** Classe HittableBehavior.

```
1
2  public class HittableBehaviour : MonoBehaviour
3  {
4      Variáveis:
5      [...]
6
7      void Awake() {
8          cameraEffects = GameObject.FindGameObjectWithTag("CameraEffects")
9          .GetComponent<CameraEffects>();
10     }
11
12     public virtual void TakeDamage(float damage) {
13         if (died || !healthbarManager.CanTakeDamage() || iFrame)
14             return;
15
16         healthbarManager.TakeDamage(damage);
17
18         if (healthbarManager.isDead())
19             Die();
20         else {
21             StartCoroutine(animator.PlayTakeDamage());
22             StartKnockback();
23         }
24     }
25
26     public virtual void TakeDamage(float damage, bool hitWeakSpot) {
27         if (died || !healthbarManager.CanTakeDamage() || iFrame)
28             return;
29
30         if (hitWeakSpot) {
31             healthbarManager.TakeDamage(damage * weakSpotDmgMul);
32
33             if (healthbarManager.isDead())
34                 Die();
35             else {
36                 StartCoroutine(animator.PlayTakeCriticalDamage());
37                 StartKnockback();
38             }
39
40             cameraEffects.ShakeCamera();
41         }
42         else {
43             healthbarManager.TakeDamage(damage);
44
45             if (healthbarManager.isDead())
46                 Die();
47             else {
48                 StartCoroutine(animator.PlayTakeDamage());
49                 StartKnockback();
50             }
51         }
52     }
```

**Programa A.7** Classe HittableBehavior.

```
1       public virtual void Die() {
2           if (died)
3               return;
4
5           died = true;
6                   audioSource.Stop();
7                   audioSource.clip = deathSound;
8                   audioSource.PlayDelayed(deathSoundDelay);
9           StartCoroutine(animator.PlayDie());
10          foreach(Collider2D col in entityColliders) {
11              col.enabled = false;
12          }
13      }
14
15          public virtual void AfterDie() {
16                  gameObject.SetActive(false);
17          }
18
19      public virtual bool IsBounceable() {
20          if(healthbarManager.isDead()) return false;
21          return isBounceable;
22      }
23
24      private void StartKnockback() {
25          Transform player = GameObject.FindGameObjectWithTag("Player").transform;
26
27          Vector2 force = Vector2.zero;
28          if (player.gameObject == this.gameObject)
29              force = new Vector2(-knockbackForceX *
30              Mathf.Sign(thisFlippable.transform.rotation.y),
31              knockbackForceY);
32          else {
33              float xForceValue = -knockbackForceX *
34                  calcPlayerDir.WhichPlayerDirection(player) *
35                  Mathf.Sign(thisFlippable.transform.rotation.y);
36
37              force = new Vector2(xForceValue, knockbackForceY);
38          }
39
40          knockbackBehaviour.ApplyKnockback(force, knockbackDuration);
41      }
42
43      public void SetIFrame(bool status) {
44          iFrame = status;
45      }
46  }
```

**Programa A.8** Classe PlayerHealthbarManager.

```
1  public class PlayerHealthbarManager : HealthManager
2  {
3      [SerializeField] EntityAnimator animator;
4      [SerializeField] GameObject _deathMenu;
5      public Image healthBar;
6
7      // Start is called before the first frame update
8      public override void Start()
9      {
10         health = Mathf.Clamp(health, 0, maxHealth);
11         healthBar.fillAmount = health / maxHealth;
12     }
13
14     public override void TakeDamage(float damage)
15     {
16
17         if (health <= damage){
18             _deathMenu.SetActive(true);
19         }
20         if (Mathf.Sign(passedTime) == -1)
21         {
22             passedTime = +0.0f;
23             if (damage > 0)
24             {
25                 health = Mathf.Max(health - damage, 0);
26                 healthBar.fillAmount = health / maxHealth;
27                 StartCoroutine(animator.PlayTakeDamage());
28             }
29         }
30     }
31
32     public override void Heal(float healing)
33     {
34         if (healing > 0)
35         {
36             health = Mathf.Clamp(health + healing, 0, maxHealth);
37             healthBar.fillAmount = health / maxHealth;
38         }
39     }
40 }
```

**Programa A.9** Classe PlayerAnimator.

```
1  public class PlayerAnimator : EntityAnimator {
2      [SerializeField] SpriteRenderer playerSprite;
3      [SerializeField] HealthManager healthbarManager;
4      private bool isInCriticalAnimation = false;
5
6      public override IEnumerator PlayTakeDamage() {
7          isInCriticalAnimation = true;
8          float interval = healthbarManager.GetDamageInterval();
9
10         playerSprite.DOColor(Color.red, interval / 2);
11         yield return new WaitForSeconds(interval / 2);
12         playerSprite.DOColor(Color.white, interval / 2);
13
14         isInCriticalAnimation = false;
15         yield break;
16     }
17
18     public override IEnumerator PlayDie() {
19         isInCriticalAnimation = true;
20         gameObject.transform.DOShakePosition(1.0f, 0.5f, 10, 90.0f, false, true);
21
22         yield return new WaitForSeconds(1.0f);
23         Vector3 newEndPosition = new Vector3(gameObject.transform.position.x, -1000.0f, 0.0f);
24         gameObject.transform.DOMove(newEndPosition, 100.0f);
25
26         yield return new WaitForSeconds(4.0f);
27         isInCriticalAnimation = false;
28         yield break;
29     }
30
31     public IEnumerator PlayPlayerHorizontalAttackAnimation(PlayerWeaponBehaviour playerWeapon) {
32         unityAnimator.SetTrigger("AttackHorizontal");
33         playerWeapon.TriggerHorizontalAttackAnimation();
34         yield break;
35     }
36
37     public IEnumerator PlayPlayerDownAttackAnimation(PlayerWeaponBehaviour playerWeapon) {
38         unityAnimator.SetTrigger("AttackDown");
39         playerWeapon.TriggerDownAttackAnimation();
40         yield break;
41     }
42
43     public IEnumerator PlayPlayerUpAttackAnimation(PlayerWeaponBehaviour playerWeapon) {
44         unityAnimator.SetTrigger("AttackUp");
45         playerWeapon.TriggerUpAttackAnimation();
46         yield break;
47     }
```

**Programa A.10** Classe PlayerAnimator.

```
1       public IEnumerator PlayPlayerProjectileAnimation(PlayerRangedWeaponBehaviour
2       playerRangedWeapon) {
3           unityAnimator.SetTrigger("Fire");
4           playerRangedWeapon.TriggerFireAnimation();
5
6           yield break;
7       }
8
9       public bool IsInCriticalAnimation() {
10          return isInCriticalAnimation;
11      }
12  }
```

**Programa A.11** Classe PlayerAnimator.

```
1  public enum Direction {HORIZONTAL, UP, DOWN}
2
3  public class PlayerAttackBehaviour : MonoBehaviour
4  {
5      [SerializeField] TarodevController.PlayerController playerCharacterController;
6      [SerializeField] PlayerWeaponBehaviour playerWeapon;
7      [SerializeField] PlayerRangedWeaponBehaviour playerRangedWeapon;
8      [SerializeField] PlayerAnimator playerAnimator;
9      [SerializeField] LayerMask enemyLayer;
10
11     TarodevController.PlayerController playerController;
12
13     void Awake()
14     {
15         playerController = GameObject.FindGameObjectWithTag("Player")
16         .GetComponent<TarodevController.PlayerController>();
17         playerController.Attacked += InitAttack;
18         playerController.RangedAttacked += InitProjectile;
19     }
20
21     public void InitAttack()
22     {
23         Direction dir = playerController.GetDirection();
24         if (dir == Direction.HORIZONTAL || dir == Direction.HORIZONTAL)
25             StartCoroutine(playerAnimator.PlayPlayerHorizontalAttackAnimation(playerWeapon));
26         else if (dir == Direction.DOWN)
27             StartCoroutine(playerAnimator.PlayPlayerDownAttackAnimation(playerWeapon));
28         else if(dir == Direction.UP)
29             StartCoroutine(playerAnimator.PlayPlayerUpAttackAnimation(playerWeapon));
30         else
31         {
32             Debug.LogError("Direction not recognized for InitAttack");
33             return;
34         }
35     }
36
37     public void InitProjectile()
38     {
39         StartCoroutine(playerAnimator.PlayPlayerProjectileAnimation(playerRangedWeapon));
40     }
41
42     public void LaunchProjectile()
43     {
44         playerRangedWeapon.Fire();
45     }
46
47     public void StopPlayerMovement()
48     {
49         playerCharacterController.TakeAwayInputControl();
50     }
51
52     public void ReturnPlayerMovement()
53     {
54         playerCharacterController.ReturnInputControl();
55     }
56
57     public void ResetPlayerSpeed()
58     {
59         playerCharacterController.ResetPlayerSpeed();
60     }
61 }
```

**Programa A.12** Classe PlayerWeaponBehaviour.

```
1  public class PlayerWeaponBehaviour : MonoBehaviour
2  {
3      Variaveis:
4      [...]
5
6      void Awake()
7      {
8          playerController = GameObject.FindWithTag("Player").
9          GetComponent<TarodevController.PlayerController>();
10     }
11
12     public void TriggerHorizontalAttackAnimation()
13     {
14         audioSource.PlayOneShot(attackSwooshSound, 0.40f);
15         animator.SetTrigger("AttackHorizontal");
16     }
17
18     public void TriggerDownAttackAnimation()
19     {
20         audioSource.PlayOneShot(attackSwooshSound, 0.40f);
21         animator.SetTrigger("AttackDown");
22     }
23
24     public void TriggerUpAttackAnimation()
25     {
26         audioSource.PlayOneShot(attackSwooshSound, 0.40f);
27         animator.SetTrigger("AttackUp");
28     }
29
30     GameObject[] CheckForWeakSpots(Collider2D[] col)
31     {
32         GameObject[] weakSpotsReached = new GameObject[col.Length];
33
34         int i = 0;
35         foreach(Collider2D c in col)
36         {
37             WeakSpot w = c.GetComponent<WeakSpot>();
38             if (w != null)
39             {
40                 if (w.IsHittable() && w.IsPlayerCorrectPos(transform))
41                     weakSpotsReached[i] = w.gameObject.transform.parent.parent.gameObject;
42             }
43             i++;
44         }
45
46         return weakSpotsReached;
47     }
```

**Programa A.13** Classe PlayerWeaponBehaviour.

```
    void FinalizeAttack()
    {
        GameObject[] attackedWeakSpots = CheckForWeakSpots(colliderDetector
        .enemiesInWeaponRange.ToArray());

        foreach(Collider2D c in colliderDetector.enemiesInWeaponRange)
        {
            HittableBehaviour hittableBehaviour = c.GetComponent<HittableBehaviour>();
            if (hittableBehaviour != null)
            {
                bool reachedWeakSpot = false;

                foreach (GameObject o in attackedWeakSpots)
                {
                    if (o == hittableBehaviour.gameObject)
                    {
                        reachedWeakSpot = true;
                        break;
                    }
                }

                ApplyEffect(hittableBehaviour, reachedWeakSpot);
            }
        }

        colliderDetector.enemiesInWeaponRange = new List<Collider2D>();
    }

    void ApplyEffect(HittableBehaviour hittableBehaviour, bool hitWeakSpot)
    {
        if (hittableBehaviour.IsBounceable() && playerController.ShouldBounce())
        {
            playerController.Bounce();
        }

        hittableBehaviour.TakeDamage(damage, hitWeakSpot);
    }

    public void DisableAllAttackColliders()
    {
        attackUpCollider.enabled = false;
        airAttackHorizontalCollider.enabled = false;
        attackHorizontalCollider.enabled = false;
        attackDownCollider.enabled = false;
        colliderDetector.enemiesInWeaponRange = new List<Collider2D>();
    }

    public void StartAttackUp() { attackUpCollider.enabled = true; }
    public void EndAttackUp() { attackUpCollider.enabled = false; }

    public void StartAirAttackHorizontal() { airAttackHorizontalCollider.enabled = true; }
    public void EndAirAttackHorizontal() { airAttackHorizontalCollider.enabled = false; }

    public void StartAttackHorizontal() { attackHorizontalCollider.enabled = true; }
    public void EndAttackHorizontal() { attackHorizontalCollider.enabled = false; }

    public void StartAttackDown() { attackDownCollider.enabled = true; }
    public void EndAttackDown() { attackDownCollider.enabled = false; }
}
```

**Programa A.14** Classe PlayerRangedWeaponBehaviour.

```
1  public class PlayerRangedWeaponBehaviour : MonoBehaviour
2  {
3      public GameObject directionSetter;
4
5      [SerializeField] GameObject projectile;
6
7      public void Fire()
8      {
9          GameObject instance = Instantiate(projectile,
10                                           directionSetter.transform.position,
11                                           directionSetter.transform.rotation);
12         instance.GetComponent<Projectile>().FireProjectile(directionSetter.transform.right);
13     }
14
15     public void TriggerFireAnimation()
16     {
17
18     }
19 }
```

**Programa A.15** Classe Projectile.

```
1  public class Projectile : MonoBehaviour
2  {
3      ...
4
5      public void FireProjectile(Vector3 direction) {
6          this.direction = direction;
7          lastPosition = GetCurrentPosition();
8      }
9
10     public void FixedUpdate() {
11         time += Time.deltaTime;
12         if (time >= projectileDuration) ExpireSequence();
13
14         // get difference between last position and next position
15         Vector3 displacement = velocity * Time.fixedDeltaTime * Mathf.Sign(direction.x);
16
17         transform.position += displacement;
18         RaycastHit2D hit = Physics2D.Raycast(lastPosition,
19                                              transform.position - lastPosition,
20                                              displacement.magnitude,
21                                              groundLayer);
22
23         if (hit.collider != null)
24             HitNonHittableSequence();
25
26         lastPosition = GetCurrentPosition();
27
28         Collider2D[] col = Physics2D.OverlapCircleAll(transform.position,
29                                                       hitRadius,
30                                                       hitMask);
31
32         if (col.Length > 0)
33             FinalizeAttack(col);
34     }
35
36     void FinalizeAttack(Collider2D[] col) {
37         HittableBehaviour hittableBehaviour = col[0].GetComponent<HittableBehaviour>();
38         HealthManager healthManager = col[0].GetComponent<HealthManager>();
39
40         if (healthManager != null && healthManager.isDead()) return;
41         if (hittableBehaviour == null || healthManager == null) {
42             HitNonHittableSequence();
43             return;
44         }
45
46         hittableBehaviour.TakeDamage(damage);
47         HitHittableSequence();
48     }
49
50     void HitHittableSequence() { Destroy(gameObject); }
51     void HitNonHittableSequence() { Destroy(gameObject); }
52     void ExpireSequence() { Destroy(gameObject); }
53 }
```

**Programa A.16** Classe SlimeMovement.

```
1  public class SlimeMovement : AbstractMovement
2  {
3      [SerializeField] EnemyCharacterController thisEnemyCharacterController;
4      [SerializeField] Enemy thisEnemy;
5      [SerializeField] Transform visionObject;
6      [SerializeField] Transform bottomObject;
7
8      [SerializeField] LayerMask groundMask;
9      //[SerializeField] HealthManager healthManager;
10
11     [SerializeField] private float speed = 10.0f;
12     [SerializeField] private float visionDistance = 10.0f;
13
14     protected override void Move()
15     {
16
17         RaycastHit2D hit = Physics2D.Raycast(transform.position,
18         Vector3.Normalize(visionObject.transform.position - bottomObject.position),
19         visionDistance, groundMask);
20
21         if (hit.collider != null)
22         {
23             thisEnemy.Flip();
24         }
25
26         Vector3 s = Vector3.Normalize(visionObject.transform.position -
27         bottomObject.position) * speed;
28         thisEnemyCharacterController.SetHorizontalSpeed(s);
29     }
30
31     protected override void NotMove()
32     {
33         thisEnemyCharacterController.SetSpeed(Vector2.zero);
34         thisEnemyCharacterController.EndExternalSpeed();
35         return;
36     }
37 }
```

**Programa A.17** Classe Predaplant.

```
public class Predaplant : MonoBehaviour
{
    private StateMachine fsm;

    private Transform playerTransform;

    public bool finishedAttackAnimation = false;

    [SerializeField] bool drawVisionGizmos = false;
    [SerializeField] bool drawAttackRadius = false;

    [SerializeField] Transform visionOrigin;
    [SerializeField] Transform attackOrigin;

    [SerializeField] EnemyAnimator animator;
    [SerializeField] Enemy thisEnemy;

    [SerializeField] LayerMask playerLayer;

    [SerializeField] float visionRadius;
    [SerializeField] float attackRadius;

    [SerializeField] PlayerDirection playerDir;

    void Start()
    {
        playerTransform = GameObject.FindGameObjectWithTag("Player").transform;

        fsm = new StateMachine(this);

        fsm.AddState("Idle");
        fsm.AddState("Active");
        fsm.AddState("Attacking",
            onEnter: (state) => { animator.unityAnimator.SetTrigger("Attack"); } );
        fsm.AddState("Flip",
            onEnter: (state) => thisEnemy.Flip());

        fsm.AddTransition("Idle", "Active", t => SawOrHeardPlayer());
        fsm.AddTransition("Active", "Attacking", t => IsPlayerClose());
        fsm.AddTransition("Active", "Flip", t => IsPlayerBehind());
        fsm.AddTransition("Attacking", "Active", t => IsAttackingFinished());
        fsm.AddTransition("Flip", "Active", t => true);

        fsm.Init();
    }

    void Update()
    {
        fsm.OnLogic();
    }

    bool SawOrHeardPlayer()
    {
        Collider2D col = Physics2D.OverlapCircle(visionOrigin.position,
        visionRadius, playerLayer);

        return col != null;
    }
```

**Programa A.18** Classe Predaplant.

```
1     bool IsPlayerClose()
2     {
3         if (playerTransform != null)
4         {
5             return (playerTransform.position - attackOrigin.position).magnitude < attackRadius;
6         }
7         return false;
8     }
9
10    bool IsPlayerBehind()
11    {
12        return playerDir.IsPlayerBack(playerTransform);
13    }
14
15    bool IsAttackingFinished()
16    {
17        if (finishedAttackAnimation)
18        {
19            finishedAttackAnimation = false;
20            return true;
21        }
22        return false;
23    }
24
25    public void EndOfAttackAnimation()
26    {
27        finishedAttackAnimation = true;
28    }
```

**Programa A.19** Classe TwoPointMove.

```
1  public class TwoPointMove : AbstractMovement
2  {
3      [SerializeField] Vector3[] points;
4      [SerializeField] Enemy thisEnemy;
5
6      int target = 1;
7      int speed = 5;
8
9      Vector3 startPosition;
10     bool startPositionSet = false;
11
12     protected override void Move()
13     {
14         if (points.Length < 2)
15             return;
16
17         if (!startPositionSet)
18         {
19             startPosition = transform.position;
20             startPositionSet = true;
21         }
22
23
24         Vector3 globalTarget = startPosition + points[target];
25
26         Vector3 movement = globalTarget - transform.position;
27
28         float frameOffset = speed * Time.deltaTime;
29         float distToTarget = movement.sqrMagnitude;
30
31         if (distToTarget > frameOffset)
32         {
33
34             transform.position += movement.normalized * speed * Time.deltaTime;
35         }
36         else
37         {
38             transform.position = globalTarget;
39             target = 1 - target;
40             thisEnemy.Flip();
41         }
42     }
43 }
```

**Programa A.20** Classe TutorialBoss.

```
1  public class TutorialBoss : MonoBehaviour
2  {
3          [SerializeField] Level_teste_1_Manager lvlChanger;
4          [SerializeField] HealthManager health;
5          [SerializeField] BehaviorTree mainBehaviour;
6          [SerializeField] GameObject spikePrefab;
7          [SerializeField] GameObject spikeUndergroundReferece;
8
9          [SerializeField] GameObject constrictPrefab;
10         [SerializeField] GameObject groundLevelReference;
11
12     // Start is called before the first frame update
13     void Start()
14     {
15                 mainBehaviour = new BehaviorTreeBuilder(gameObject)
16                         .Sequence()
17                           .WaitTime(1.1f)
18                           .SelectorRandom()
19                             .Sequence()
20                               .Condition(() => {return IsPlayerInCorrectDistance();})
21                               .TriggerAnimation("Sweep")
22                             .End()
23                             .Sequence()
24                               .TriggerAnimation("Stomp")
25                               .WaitTime(0.35f)
26                               .Stomp(spikePrefab, spikeUndergroundReferece)
27                             .End()
28                             .Sequence()
29                               .TriggerAnimation("Constrict")
30                               .WaitTime(0.7f)
31                               .Constrict(constrictPrefab, groundLevelReference.transform)
32                             .End()
33                           .End()
34                         .End().Build();
35     }
36
37     void Update()
38     {
39                 if (!health.isDead())
40                 mainBehaviour.Tick();
41                 else {
42                         Finish();
43                 }
44     }
45
46         private IEnumerator Finish(){
47                 yield return new WaitForSeconds(10);
48                 lvlChanger.FinishLevel();
49                 yield break;
50
51         }
52         bool IsPlayerInCorrectDistance()
53         {
54                 return true;
55         }
56 }
```

**Programa A.21** Classe PowerUpManager.

```
1  public class PowerUpManager : MonoBehaviour
2  {
3      private AbstractPowerUp powerUp;
4      private InputManager _input;
5      private FrameInput FrameInput;
6
7      [SerializeField]
8      private PlayerManabarManager manabar;
9
10     protected virtual void Awake() {
11         _input = GetComponent<InputManager>();
12
13     }
14
15     // Update is called once per frame
16     void Update()
17     {
18         FrameInput = _input.FrameInput;
19         if (FrameInput.PowerUp)
20         {
21             if (powerUp != null && manabar.HasEnoughMana(powerUp.manaCost))
22             {
23                 powerUp.Use();
24                 manabar.LoseMana(powerUp.manaCost);
25             }
26             else
27             {
28             }
29         }
30     }
31
32     public void equipPowerUp(AbstractPowerUp aPowerUp)
33     {
34         powerUp = aPowerUp;
35         Debug.Log("Power Up equipado!");
36     }
37 }
38
```

**Programa A.22** Classe PlayerDirection 1.

```
1  public class PlayerDirection : MonoBehaviour
2  {
3      [SerializeField] Enemy thisEnemy;
4      [SerializeField] Transform topOfHead;
5      [SerializeField] Transform bottomOfFeet;
6      [SerializeField] Transform front;
7      [SerializeField] Transform back;
8      [SerializeField] Transform flippable;
9
10     public bool IsPlayerFront(Transform player)
11     {
12         Vector3 result = new Vector3(0, 0, 0);
13
14         if (!thisEnemy.IsFlipped())
15             result = player.position - front.position;
16         else
17             result = player.position - back.position;
18
19         if ((result.x < 0 && !thisEnemy.IsFlipped()) ||
20         (result.x > 0 && thisEnemy.IsFlipped()))
21         {
22             return true;
23         }
24
25         return false;
26     }
27
28     public bool IsPlayerBack(Transform player)
29     {
30         Vector3 result = new Vector3(0, 0, 0);
31
32         if (!thisEnemy.IsFlipped())
33             result = player.position - back.position;
34         else
35             result = player.position - front.position;
36
37         if ((result.x > 0 && !thisEnemy.IsFlipped()) ||
38         (result.x < 0 && thisEnemy.IsFlipped()))
39             return true;
40
41         return false;
42     }
43
44     public bool IsPlayerUp(Transform player)
45     {
46         Vector3 result = player.position - topOfHead.position;
47
48         if (result.y > 0)
49             return true;
50
51         return false;
52     }
```

**Programa A.23** Classe PlayerDirection 2.

```
    public bool IsPlayerDown(Transform player)
    {
        Vector3 result = player.position - bottomOfFeet.position;

        if (result.y < 0)
            return true;

        return false;
    }

    public int WhichPlayerDirection(Transform player)
    {
        if (IsPlayerBack(player)) return 1;
        else if(IsPlayerFront(player)) return -1;
        else return 0;
    }
}
```

**Programa A.24** Classe HUB_Player.

```
public class HUB_Player : MonoBehaviour
{
    private float velocity = 15.0f;
    private bool input_locked = true;
    public int Levels;
    [SerializeField]
    private HUBPosition currentHubPosition;

    // Start is called before the first frame update
    void Start()
    {
        int nLevels = PlayerPrefs.GetInt("LevelsUnlocked", 1);
        Levels = nLevels;
        Debug.Log(nLevels);
    }

    // Update is called once per frame
    void Update()
    {
        Debug.Log(input_locked);
        if (!input_locked)
        {
            if (Input.GetKeyDown(KeyCode.RightArrow) &&
            currentHubPosition.next != null &&
            Levels > currentHubPosition.level)
            {
                input_locked = true;
                currentHubPosition = currentHubPosition.next;
                Debug.Log("Entrei");
            }
            if (Input.GetKeyDown(KeyCode.LeftArrow) && currentHubPosition.prev != null)
            {
                input_locked = true;
                currentHubPosition = currentHubPosition.prev;
            }
        }
        else
        {
            WalkToPosition();
        }

        if (!input_locked && Input.GetKeyDown(KeyCode.Return))
        {
            SceneManager.LoadScene(currentHubPosition.sceneName);
        }
    }

    void WalkToPosition()
    {
        Vector3 from = transform.position;
        Vector3 to = currentHubPosition.transform.position;
        Vector3 walkDirection = (to - from).normalized;

        if (velocity * Time.deltaTime < (to - from).magnitude)
            transform.Translate(walkDirection * velocity * Time.deltaTime);
        else
        {
            transform.Translate(to - from);
            input_locked = false;
        }
    }
}
```

**Programa A.25** Classe Input Manager (1).

```
#if ENABLE_INPUT_SYSTEM
using UnityEngine.InputSystem;
#endif
namespace TarodevController {
    public class InputManager : MonoBehaviour {
        public static InputManager instance;
        public FrameInput FrameInput { get; private set; }


        private PlayerInput _playerInput;
        private InputAction _move, _jump, _dash,
        _attack, _rangedAttack, _powerUp,
        _exampleAction, _escape;

        private void Awake() {

            if (instance == null){
                instance = this;
            }

            _playerInput = GetComponent<PlayerInput>();
            SetupActions();
        }

        private void SetupActions() {
            _move = _playerInput.actions["Move"];
            _jump = _playerInput.actions["Jump"];
            _dash = _playerInput.actions["Dash"];
            _attack = _playerInput.actions["Attack"];
            _powerUp = _playerInput.actions["PowerUp"];
            _rangedAttack = _playerInput.actions["RangedAttack"];
            _escape = _playerInput.actions["MenuOpenClose"];
        }

        private void Update() {
            FrameInput = UpdateInputs();
        }
```

**Programa A.26** Classe Input Manager (2).

```
        private FrameInput UpdateInputs() {
            return new FrameInput {
                Move = _move.ReadValue<Vector2>(),
                JumpDown = _jump.WasPressedThisFrame(),
                JumpHeld = _jump.IsPressed(),
                DashDown = _dash.WasPressedThisFrame(),
                AttackDown = _attack.WasPressedThisFrame(),
                PowerUp = _powerUp.WasPressedThisFrame(),
                RangedAttackDown = _rangedAttack.WasPressedThisFrame(),
                EscapeDown = _escape.WasPressedThisFrame(),
            };
        }
    }

    public struct FrameInput {
        public Vector2 Move;
        public bool JumpDown;
        public bool JumpHeld;
        public bool DashDown;
        public bool AttackDown;
        public bool PowerUp;
        public bool RangedAttackDown;
        public bool EscapeDown;
        public bool ExampleActionHeld;
    }

}
```

**Programa A.27** Classe RebindSaveLoad

```
public class RebindSaveLoad : MonoBehaviour
{
    public InputActionAsset actions;

    public void OnEnable()
    {
        var rebinds = PlayerPrefs.GetString("rebinds");
        if (!string.IsNullOrEmpty(rebinds))
            actions.LoadBindingOverridesFromJson(rebinds);
    }

    public void OnDisable()
    {
        var rebinds = actions.SaveBindingOverridesAsJson();
        PlayerPrefs.SetString("rebinds", rebinds);
    }
}
```

# Apêndice B

# Formulário de feedback dos jogadores

## B.1 O que você gostou no jogo?

- O jogo está muito bem polido: Com animações incriveis, efeitos sonoros absurdos, iluminação e arte ótimas. Ótimo menu e opening do jogo.
- Movimentação boa, com elementos que dão muito feedbacks ao jogador.
- A forma que foi desenvolvido, pode facilitar a vida de pessoas que não tem muita acessibilidade
- "1) A pixelart é muito bonita, a música e efeitos sonoros são muito bons

  2) Gostei da vibe Castlevania

  3) Os ataques são bem legais, dá pra sentir o peso da arma no tempo que demora entre apertar a tecla e o ataque acabar. Achei muito legal o ataque pra cima"
- Arte, trilha sonora e ambientação.
- A arte está muito bonita, tanto no estilo quanto execução.
- acho que foi feito com carinho e cuidado, vi que implementaram coyote jump, as animacoes pareceram bem complexas e o personagem tem um moveset bastante extenso

## B.2 O que podemos melhorar?

- Acho que a minha única crítica são os controles: Usar wasd e barra de espaço é perfeito, mas J e K não
- Colocar informações em libras
- "1) O personagem parece muito "pesado", o pulo é muito curto e ele cai muito rápido. Na primeira parte que precisa passar pelas colunas, os pulos precisam ser

milimetricamente calculados, você precisa ir até o último pixel da borda, e isso é muito chato. Em particular, o pulo logo após matar a primeira planta (texto do ranged attack) é muito injusto, porque você precisa começar a cair pra pular e conseguir atravessar. O que mais senti foi que o pulo precisa ser um pouquinho mais alto e o personagem precisa cair um pouco menos devagar.

2) O coyote time é necessário pra passar do pulo da primeira planta (ou pelo menos eu só consegui assim), o que é meio injusto, pois não houve uma seção que ensina isso antes desse pulo. Consequentemente, o jogador muito provavelmente vai morrer nas primeiras vezes que chegar nesse pulo, o que é chato

3) 90% das vezes o ataque aéreo pra baixo não acerta o inimigo antes do personagem tomar dano e ser jogado pra trás. Não sei se estou errando o timing, mas parece que o HitBox do morceguinho é muito grande, porque parece que o personagem leva dano antes de encostar nesse inimigo. Talvez o HitBox do ataque aéreo pra baixo seja muito pequeno, talvez o HitBox do morcego seja muito grande, talvez só esteja errando o timing

4) Demorei séculos pra passar do pulo antes da árvore, por causa de uma combinação dos itens (1) e (3). Esse pulo parece muito injusto, é muito difícil acertar o ataque aéreo pra baixo sem levar dano, e o problema é que se você leva dano o personagem não vai pra frente, só pra cima, então não dá pra passar

3) O ataque de longo alcance precisa de um cooldown, porque agora dá pra spammar ele e matar o boss final em muito pouco tempo. Também está precisando de um sprite, em vez de uma linha branca de raycast

4) Não tem como sair do jogo sem morrer, precisa ter uma opção de voltar ao menu principal no menu de pausa

5) É difícil prever onde a árvore vai atacar com as raízes subindo, precisava ter uma indicação mais clara. Além disso, os galhos que agarram o personagem são legais, mas não deu pra entender como escapar deles, se precisa de um ataque específico ou só tentar pular

6) Depois da árvore não tem nada indicando que a alpha acabou, o que deixa o jogador confuso"

- Sistema de checkpoint e ser um pouco mais fácil de pular no inimigo para atravesar; o salto do personagem poderia ser um pouco mais pra frente ao invés de pra cima e a câmera poderia ser um pouco mais afastada para poder ver o cenário à frente.
- Primeiramente o ataque a distância é bem esquisito, ele não tem "peso"nenhum, só lança uma linha para frente. Também achei a mecânica de pular em cima dos inimigos no ar muito difícil de realizar, além de geralmente me lançar apenas para cima e não para frente (não consegui fazer o pulo longo com o morcego no meio). Senti a falta de checkpoints para cada parte do tutorial (sim eu morri bastante no morcego). Parece que tem um pouco de input lag nos movimentos, eles parecem demorar muito para começar, talvez seja impressão ou coisa de ter jogado no browser.
- "em termos de level design: - o primeiro consumivel(no tutorial) não pode ser alcan-

cado caso o jogador caia na plataforma de baixo. talvez ou botar o consumivel no nivel do chão normal ou pensar em alguma forma de deixar ele pegar.

- Algumas vezes me confundi com o inimigo de planta, meu cerebro interpretou como sendo coisa do background e tomei dano sem entender direito o que aconteceu.

- Qdo cai em buracos vc morre e volta do começo... pode ser intencional a dificuldade do jogo, uma vez que os ataques são pesados e lentos, talvez a intenção seja ir pra uma pegada mais dark souls. mas achei bastante punitivo cair e voltar tudo. talvez eu faria o player perder um pouco de vida ao cair de buracos, mas spawnaria ele logo antes do pulo para que não morra instantaneamente hehehe mas pode ser só porque eu sou ruim.

Em termos de arte: - achei a paleta de cores, o estilo dos personagens e a resolução dos sprites um pouco inconsistentes. talvez seja legal pensar em uma resolução para trabalhar (por exemplo, usar sprites de tamanho 16x16 como base e sempre respeitar esse tamanho. não dar scale em objetos feitos em pixel art tb é uma boa para manter consistencia). Usar pixel art tb tem a vantagem de ser facil de trabalhar. não precisa ser mto bom de desenho pra fazer uns pixel arts bem bonitos e umas animações bacanas. Para a paleta de cores, eu trabalharia com uma paleta bem limitada... tipo com 16 cores no maximo e trabalhar em cima dela... vcs podem ir nesse site aqui: https://lospec.com/palette-list e escolher uma paleta e só usar cores desta paleta no jogo inteiro. isso dá uma consistencia visual bem bacana. Voces podem reparar que, ao se limitar a uma unica paleta de cor, os sprites individualmente vão parecer mais feios e menos detalhados. voces vão ter que comprometer detalhes pra respeitar a paleta de cores e as vezes isso é bem chato, mas apesar dos sprites individuais ficarem mais feios, o conjunto todo, do jogo, de todos os sprites seguindo um unico estilo visual, vai fazer o jogo ficar bem bem bem bonitão hehehe

Em termos de jogabilidade: Eu particularmente gosto de jogos com poucos botões x) tem muitos comandos e alguns são faceis de esquecer. fico pensando se o botão dedicado para dash não poderia ser substituido por um duplo clique das setinhas. ou se o ataque a distancia não poderia ser uma combinação, como por exemplo abaixar+atacar (já que o ataque a distancia sai de +- a altura do joelho do personagem). (: "

## B.3 Conte-nos como foi sua experiência

- A minha experiência foi boa, joguei um pouco
- Achei interessante
- Demorou um bom tempo pra me acostumar com os pulos, o que é meio frustrante. O personagem é pesado demais e você precisa acertar as bordas do pulo de um jeito pixel perfect. Morri muitas vezes no pulo logo antes da árvore, então foi muito chato ficar voltando do começo pra tentar de novo o mesmo pulo e errar o timing do ataque aéreo pra baixo. Os pulos são a parte mais frustrante do jogo. Fiquei tentando entender o timing do ataque aéreo pra baixo, mas não consegui pegar o feeling dele.

- Divertido, agradável e gostoso de jogar
- Eu sinto que o jogo tem bastante potencial, sua atmosfera é bonita e interessante, mas a gameplay está um pouco fraca ainda, parece não muito natural ainda, mas tenho certeza de que irá melhorar com o tempo.
- foi bacana! eu gostei do estilo do jogo, gostei que pareceu um jogo mais dark e com uma jogabilidade mais pesada... mas acho que dá pra dar umas polidas em alguns lugares (:

## B.4 Alguma outra observação?

- "1) Demorei pra entender que a tela inicial é um sprite do herói petrificado. Como não sabia que ele tinha uma espada gigante, fiquei achando que era duas pessoas ou algo assim

  2) O morcego logo antes do powerup está na mesma linha de tiro da primeira planta, então é muito fácil matar ele antes de chegar na tela que daria pra ver. Assim, na maioria das vezes eu não consegui pegar o powerup, porque cheguei perto dele e o morcego não estava lá

  3) A resolução no itch está cortando metade do ícone de tela cheia (canto inferior direito)

  4) Menu Keyboard Controls: opção MenuOpenClose dá wrap no texto e a letra "e"fica embaixo

  5) Remapeamento de tecla: muito difícil de ler o texto branco no fundo cinza claro. Recomendo escurecer o background

  6) Por que o menu de settings in-game tem mais opções que o menu principal? Pra mim faz sentido ter todas as opções já no menu inicial, como o áudio e todos os mapeamentos de todas as teclas

  7) Settings: Sugiro que a barra de volume seja ajustada para que ela não comece no máximo, é melhor o default ter um espaço pra aumentar o volume se necessário

  8) No texto "Press W/S + J to attack up/down"não ficou claro se deveria existir um ataque com a combinação Baixo + Ataque com o personagem no chão. Acho que a intenção é que existe o Cima + Ataque e o Baixo + Ataque exige que o personagem esteja no ar, mas isso não é muito claro"

# Apêndice C

# Game Design Document (GDD)

O documento de design do jogo pode ser encontrado no *link* a seguir: https://www.linux.ime.usp.br/~andregnl/MAC0499/extra/GDD%20-%20TCC.pdf

# Referências


[Alkakrab 2022a] Alkakrab. *Fantasy RPG Music*. 2022. URL: https://alkakrab.itch.io/free-25-fantasy-rpg-game-tracks-no-copyright-vol-2 (acesso em 23/10/2023).

[Alkakrab 2022b] Alkakrab. *Pixel RPG Music*. 2022. URL: https://alkakrab.itch.io/free-12-tracks-pixel-rpg-game-music-pack (acesso em 23/10/2023).

[Ash 2017] Ash Blue. *Fluid Behavior Tree*. 2017. URL: https://github.com/ashblue/fluid-behavior-tree (acesso em 22/10/2023).

[ASKS 2019] Askiisoft. *Katana Zero*. 2019. URL: https://www.katanazero.com (acesso em 25/11/2023).

[Fejér 2023] Attila Fejér. *What Does It Mean to Program to Interfaces?* 2023. URL: https://www.baeldung.com/cs/program-to-interface (acesso em 24/11/2023) (citado na pg. 39).

[Doran 2023] Doranarasi. *JRPG Battle BGM*. 2023. URL: https://doranarasi.itch.io/jrpg-battle-bgm-pack-1 (acesso em 24/10/2023).

[Tween 2020] DOTween. *A fast, efficient, fully type-safe object-oriented animation engine fo Unity*. 2020. URL: https://dotween.demigiant.com/index.php (acesso em 16/06/2023).

[Inspiaa 2022] Inspiaa. *Unity Hierarquical Finite State Machine*. 2022. URL: https://github.com/Inspiaaa/UnityHFSM (acesso em 03/07/2023).

[Jeremy 2017] Jeremy Parish. *How Can I Play It? Rondo of Blood and Symphony of the Night*. 2017. URL: https://retronauts.com/article/608/how-can-i-play-it-rondo-of-blood-and-symphony-of-the-night (acesso em 01/12/2023) (citado na pg. 21).

[Juego 2023] Juego Studios. *Game Design Principles*. 2023. URL: https://www.juegostudio.com/blog/game-design-principles-every-game-designer-should-know (acesso em 22/09/2023) (citado na pg. 30).

[Karleigh 2022] Karleigh Moore. *Finite State Machines*. 2022. URL: https://brilliant.org/wiki/finite-state-machines/ (acesso em 12/05/2023).

[Karlote 2020] Karlote. *Cycle Lighting Star Effect*. 2020. URL: https://karlote.itch.io/cycle-lighting-with-star-effect (acesso em 23/10/2023).

[Maaot 2021] Maaot. *Mossy Cavern Assets*. 2021. URL: https://maaot.itch.io/mossy-cavern (acesso em 25/11/2023).

[Tarodev 2022] Matthew J. Spencer. *Ultimate 2D Controller (Free Version)*. 2022. URL: https://github.com/Matthew-J-Spencer/Ultimate-2D-Controller (acesso em 14/06/2023).

[Miggoh 2014] Miggoh. *Branching Paths (Spoiler free)*. 2014. URL: https://steamcommunity.com/sharedfiles/filedetails/?l=brazilian%5C&id=224590222 (acesso em 01/12/2023) (citado na pg. 32).

[NINTENDO 2019] NINTENDO. *New Super Mario Bros.™ U Deluxe*. 2019. URL: https://www.nintendo.com/pt-br/store/products/new-super-mario-bros-u-deluxe-switch/ (acesso em 01/12/2023) (citado na pg. 24).

[Nintendo 1983] Nintendo. *Nintendo: Super Mario World*. 1983. URL: https://www.nintendo.pt/Jogos/Super-Nintendo/Super-Mario-World-752133.html (acesso em 03/11/2023).

[Cragg 2016] Oliver Cragg. *Dark Souls 3 video guide: How to beat first boss Iudex Gundyr*. 2016. URL: https://www.ibtimes.co.uk/dark-souls-3-video-guide-how-beat-first-boss-iudex-gundyr-1554218 (acesso em 01/12/2023) (citado na pg. 33).

[outerStudios 2020] OuterStudios. *How to Create a Beautiful 2d Level in Unity*. 2020. URL: https://www.youtube.com/watch?v=CMlX0LfhJYk (acesso em 27/11/2023).

[Raphael 2017] Raphael Nascimento. *Super Mario: os 12 melhores jogos do famoso encanador!* 2017. URL: https://www.ligadosgames.com/melhores-jogos-do-mario/ (acesso em 01/12/2023).

[samyam 2020] samyam. *How to Blend/Switch Between Cinemachine Cameras*. 2020. URL: https://www.youtube.com/watch?v=Ri8PEbD4w8A (acesso em 27/11/2023).

[SBS 2023] Sasquatch B Studios. *How to Rebind Your Controls in Unity*. 2023. URL: https://www.youtube.com/watch?v=qXbjyzBlduY (acesso em 27/11/2023).

[Castro 2021] Sebastian Castro. *Introduction to Behavior Trees*. 2021. URL: https://robohub.org/introduction-to-behavior-trees/ (acesso em 28/11/2023) (citado nas pgs. 41–44).

[shaped 2022] Shaped by Rain Studios. *How to make a Save Load System in Unity*. 2022. URL: https://www.youtube.com/watch?v=aUi9aijvpgs (acesso em 27/11/2023).

[SlimCK 2019] SlimCK. *8-Bit Small Mario Crouching*. 2019. URL: https://www.reddit.com/r/MarioMaker2/comments/c7aiuw/8bit_small_mario_crouching (acesso em 01/12/2023) (citado na pg. 28).

[TeamCherry 2017] Team Cherry. *Hollow Knight*. 2017. URL: https://hollowknight.com (acesso em 20/10/2023).

[Unity 2016] Unity. *Cinemachine*. 2016. URL: https://unity.com/unity/features/editor/art-and-design/cinemachine (acesso em 02/05/2023).

[Unity 2022] Unity. *Unity User Manual 2022.3 (LTS)*. 2022. URL: https://docs.unity3d.com/Manual/index.html (acesso em 03/07/2023) (citado na pg. 10).

[VVA 2023] Vidya Vrat Agarwal. *A complete guide to Object Oriented Programming in C#*. 2023. URL: https://www.c-sharpcorner.com/UploadFile/84c85b/object-oriented-programming-using-C-Sharp-net/ (acesso em 13/02/2023) (citado na pg. 10).

[War 2022] War. *High Fantasy*. 2022. URL: https://warsvault.itch.io/high-fantasy-slime-enemy (acesso em 04/08/2023).

[WIKI 1981] Wikipedia. *Platformer*. 1981. URL: https://en.wikipedia.org/wiki/Platformer (acesso em 25/11/2023).

[Konami 1986] Wikipedia. *Konami: Castlevania*. 1986. URL: https://pt.wikipedia.org/wiki/Castlevania (acesso em 03/11/2023).

[WIKI 2023a] Wikipedia. *Delegation (object-oriented programming)*. 2023. URL: https://en.wikipedia.org/wiki/Delegation_(object-oriented_programming) (acesso em 28/11/2023).

[WIKI 2023b] Wikipedia. *Finite-state machine*. 2023. URL: https://en.wikipedia.org/wiki/Finite-state_machine (acesso em 28/11/2023) (citado na pg. 45).

[WIKI 2023c] Wikipedia. *Super Mario World*. 2023. URL: https://en.wikipedia.org/wiki/Super_Mario_World (acesso em 01/12/2023) (citado na pg. 4).

[WIKI 2023d] Wikipedia. *Tetris*. 2023. URL: https://pt.wikipedia.org/wiki/Tetris (acesso em 28/11/2023) (citado na pg. 31).

[XYEzawr 2019] XYEzawr. *Pixel Flame effects*. 2019. URL: https://xyezawr.itch.io/free-pixel-effects-pack-3?download (acesso em 24/11/2023).