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

Jogo digital 2D focado na expressão do jogador por meio da seleção de mecânicas de jogo

Eduardo Yukio Rodrigues

Monografia Final

mac 499— Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Ricardo Nakamura

São Paulo
10 de Março de 2021

Resumo

Eduardo Yukio Rodrigues. **Jogo digital 2D focado na expressão do jogador por meio da seleção de mecânicas de jogo**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

O objetivo deste trabalho foi desenvolver um jogo digital 2D, do gênero de plataforma, que desse ao jogador a possibilidade de se expressar por meio da escolha de mecânicas de jogo. Para cumprir este objetivo, inicialmente foi feito um processo de catalogação de mecânicas, no qual analisou-se 50 jogos relevantes para a indústria de videogames e elaborou-se uma lista de mecânicas utilizadas por eles. Em seguida, foi feito o *game design*, em que definiu-se todos os elementos fundamentais do jogo e quais mecânicas da lista seriam implementadas. Então, o processo avançou para a etapa de desenvolvimento, em que ocorreu a programação do jogo, com vasta utilização de máquinas de estado. Por último, o jogo foi lançado na plataforma *Itch.io* e obteve uma grande quantidade de *feedback* positivo, tanto por meio de um formulário ativamente compartilhado quanto por engajamento orgânico, proveniente da própria plataforma.

Palavras-chave: Jogo digital. Desenvolvimento de jogos. Mecânica de jogo. Game Design. Expressão do jogador. Máquina de estados.

Abstract

Eduardo Yukio Rodrigues. **2D digital game focused on the player's expression through the selection of game mechanics**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

The objective of this work was to develop a 2D digital game, of the platform genre, that would give the player the possibility of expressing himself through the selection of game mechanics. To fulfill this objective, a cataloging process was initially carried out, in which 50 games relevant to the video game industry were analyzed and a list of mechanics used by them was elaborated. Then, the game design was made, in which all the game's fundamental elements were defined and the mechanics that would be implemented from the list were decided. Then, the process advanced to the development stage, in which the game was programmed with wide use of state machines. Finally, the game was launched on the Itch.io platform and it obtained a large amount of positive feedback, both through an actively shared form and through organic engagement, coming from the platform itself.

Keywords: Digital game. Game development. Game mechanic. Game design. Player's expression. State Machine.

Sumário

Capítulo 1

Introdução

1.1 Motivação

Jogos são obras de entretenimento que se diferenciam de demais obras, como filmes e livros, principalmente por serem interativos [Cra03]. Em um jogo digital, o usuário (ou, jogador), deve interagir com a obra por meio de uma interface, como um controle ou um teclado, enviando comandos de modo a alterar o estado do jogo. Esta interação demanda que o jogador faça escolhas, tanto relacionadas à narrativa e ao desenvolvimento dos personagens, quanto àquelas referentes a quais comandos ele irá executar, em qual ordem, com qual velocidade, etc.

Existem diversas maneiras de se explorar o fato do jogador ter de fazer escolhas, sendo uma delas a alocação de pontos em atributos de jogos de *RPG* (*Role-Playing Game*), nos quais o jogador controla um ou mais personagens que possuem qualidades como força, agilidade e carisma.

Em muitos jogos deste gênero, é permitido que o jogador escolha como vai distribuir seus pontos entre os atributos disponíveis, impactando profundamente sua experiência de jogo. Por exemplo, se o jogador escolher priorizar agilidade, seu personagem poderá se esquivar melhor dos inimigos e atacá-los com maior frequência; ao passo que se escolher priorizar carisma, poderá superar seus inimigos por meio do diálogo. A característica de permitir que as escolhas do jogador tenham impacto e, de fato, alterem profundamente o decorrer da experiência é um dos grandes diferenciais desta mídia. O mesmo jogo pode gerar experiências muito diferentes, dependendo das escolhas dos jogadores. Um exemplo deste sistema pode ser observado na Figura 1.1.

Figura 1.1: *Interface de escolha de atributos do jogo Fallout 3.*

Na indústria dos videogames, essa abordagem é observada predominantemente em jogos do gênero de *RPG* [Pau17], sendo executada, em geral, de maneira conservadora: a escolha do jogador se resume a definir o quão proficiente seu personagem será em cada uma das suas ações básicas (dialogar, atacar, se esquivar), porém, as ações em si não podem ser alteradas.

1.2 Objetivos

Este trabalho tem como objetivo expandir a ideia de que o jogador possa se expressar por meio de suas escolhas. Para isso, será desenvolvido um jogo digital 2D, do gênero de plataforma, no qual a abordagem em questão é pouco explorada pela indústria. Além disso, pretende-se ir além do convencional, permitindo que o jogador escolha e altere as ações básicas do seu personagem.

Também é importante ressaltar que jogos digitais são *softwares* compostos por diversos componentes, cada um responsável por algo bastante específico: receber *inputs* do usuário, simular um mundo físico, calcular a lógica do jogo, reproduzir gráficos e sons, servir uma interface ao jogador, etc. Implementar esse sistema de maneira que todos os componentes funcionem harmoniosamente, e com alta performance, representa um grande desafio computacional.

Em conjunto com os objetivos de design citados nos parágrafos anteriores, também

há o objetivo de implementar o sistema aplicando os conhecimentos adquiridos no curso de Ciência da Computação do IME-USP, especialmente nas disciplinas de Algoritmos e Estruturas de Dados, Álgebra Linear, Design e Programação de Games, Interação Humano-Computador, Engenharia de Software e Laboratório de Métodos Ágeis. Também serão aplicados os conhecimentos adquiridos durante 3 anos de participação nas atividades extracurriculares do grupo de extensão USPGameDev.

1.3 Metodologia

Primeiramente, foi feito um trabalho de pesquisa e catalogação de jogos digitais 2D do gênero de plataforma.

Em seguida, foi elaborado o *game design*, no qual a estrutura do jogo e as principais diretrizes de desenvolvimento foram definidas.

Posteriormente, foi executado o processo de desenvolvimento, em que foram implementados os personagens e suas ações; os objetos e obstáculos dos níveis; os menus e as interfaces com o jogador; a reprodução de música e dos efeitos sonoros e, por fim, alguns efeitos gráficos de embelezamento.

Por último, o projeto foi validado por meio de testes com jogadores reais, os quais forneceram seu *feedback* através de um formulário.

Todas as etapas foram acompanhadas e validadas pelo orientador Ricardo Nakamura.

Capítulo 2

Conceitos

Neste capítulo serão apresentados os principais conceitos ligados ao desenvolvimento de jogos digitais, que são importantes para a discussão realizada no restante do texto.

2.1 Jogo Digital

Segundo Jesse Schell, escritor de *The Art of Game Design*, um jogo é uma atividade de resolução de problemas, abordada com uma atitude lúdica. O autor expande essa definição citando que jogos são interativos, possuem objetivos, conflitos, regras, desafios e deixam o usuário engajado [Sch08]. Um jogo digital encapsula essa atividade na forma de um *software* que permite interação com o usuário por meio de um dispositivo de entrada como um controle, um mouse, um teclado ou um aparelho de detecção de movimento. As entradas do usuário são processadas pelo sistema e podem resultar em mudanças no estado do jogo, as quais são informadas ao jogador por meio de *feedbacks* visuais, auditivos e táteis.

Esse processo é implementado por meio de um padrão de projeto chamado *game loop* [Nys14], apresentado na Listagem 2.1.

```
1  while(true) {
2      processInput();
3      update();
4      render();
5  }
```

Listagem 2.1: *Código resumido de um game loop.*

Para que um jogo seja experienciado de maneira fluida, este *loop* deve ocorrer 60 vezes por segundo, ou seja, uma vez a cada 16 milissegundos [Nys14]. Este fato, em conjunto com os altos níveis de complexidade que podem ser atingidos na etapa de renderização (síntese de imagens), fazem com que jogos digitais sejam dependentes de um alto desempenho computacional. Por esta razão, jogos grandes geralmente são feitos utilizando a linguagem C++, reconhecida por ser muito eficiente [Str20]. Na etapa de renderização, ocorre a

síntese de uma imagem a partir de modelos que podem ser 2D ou 3D [Wik21f]. A escolha dos modelos utilizados é tão impactante que serve como parâmetro para classificação e diferenciação entre os jogos. Uma comparação entre esses modelos pode ser vista na Figura 2.1.

Figura 2.1: *Comparação entre um jogo 2D (Super Mario World, à esquerda) e um jogo 3D (Super Mario 64, à direita).*

Em menor escala, também existem jogos que são classificados como um intermediário entre 2D e 3D: são os jogos *2.5D* [Wik21a]. Neles, são utilizados modelos 3D para a renderização do ambiente e dos objetos, porém, o jogador só tem liberdade para controlar seu personagem em um plano 2D, como pode ser visto na Figura 2.2.

Figura 2.2: *Exemplo de jogo 2.5D (Kirby 64), em que há utilização de modelos 3D e restrição de movimento em duas dimensões: na direção do caminho destacado e na direção do pulo.*

2.2 Mecânica de jogo

Mecânicas de jogo são as regras, ações, comportamentos e mecanismos de controle disponíveis para o jogador. Elas definem como o jogador pode interagir com o jogo, como a progressão ocorre, o que acontece, quando acontece e quais condições determinam a vitória ou a derrota [HLZ04] [AD12].

Em um jogo de cartas, por exemplo, existem mecânicas como embaralhar o monte, colocar uma carta da mão em jogo e apostar. Já em um jogo digital do gênero de plataforma, como *Super Mario Bros.*, encontram-se: mecânicas relacionadas às ações do jogador como andar, correr e pular; mecânicas de progressão, como os *checkpoints*, que salvam o progresso feito na fase; mecânicas de interação com o cenário como o pulo mais alto ao encostar em trampolins e a derrota ao encostar em inimigos.

As mecânicas mais influentes, que afetam diversos aspectos do jogo, são chamadas de *core mechanics* (mecânicas essenciais), as quais são utilizadas como base para a classificação dos gêneros de jogos. Em jogos de plataforma, as mecânicas de andar e pular constituem sua essência [Wik21e]. Já os jogos do gênero de luta têm como mecânicas principais as ações relacionadas ao combate um contra um de curto alcance, como socos e chutes [Wik21b]. Esses dois exemplos estão ilustrados na Figura 2.3 e na Figura 2.4.

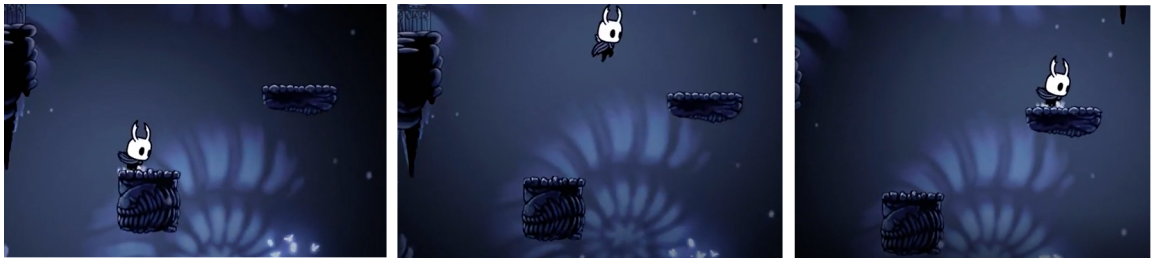

Figura 2.3: *Exemplo de jogo do gênero plataforma (Hollow Knight) e sua mecânica principal: o pulo entre plataformas.*

Figura 2.4: *Exemplo de jogo de luta (Street Fighter) e uma de suas mecânicas principais: o soco.*

2.3 Game Design

Game design é o ato de decidir o que um jogo deve ser. Nesse processo, são feitas decisões que envolvem regras, desafios, aparências, sensações, ritmos, recompensas, punições e basicamente tudo o que é experienciado pelo jogador [Sch08].

No jogo *Pac-Man*, por exemplo, as decisões de *game design* abrangem a movimentação do personagem, que pode ser feita apenas em 4 direções (cima, baixo, esquerda, direita); a existência de 4 inimigos, cada um se movimentando de uma maneira própria e a existência das pílulas de comida comuns e da super-pílula, que faz os inimigos ficarem temporariamente vulneráveis ao contato com o jogador. A tela principal do jogo pode ser vista na Figura 2.5.

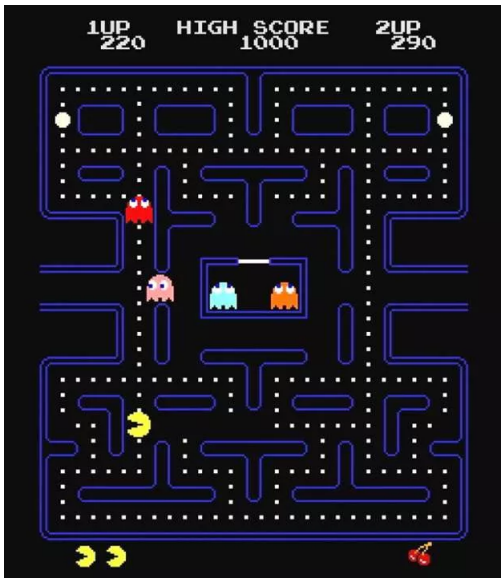

Figura 2.5: *Jogo Pac-Man.*

O *game designer* estipula também a condição de vitória (jogador comer todas as pílulas) e a condição de derrota (encostar em um inimigo, sem o efeito da super-pílula). Até mesmo o som é afetado pelo *game design*: o fato da música mudar e aumentar o seu ritmo quando o personagem adquire a super-pílula também é uma decisão de *design*. No entanto, quem produz de fato a música e os efeitos sonoros é o *sound designer*. Quem implementa as decisões de *design* em um jogo digital, de maneira concreta, é o programador. Para efeitos de clareza, será dado um exemplo um pouco mais extenso sobre a parte de programação, para que não hajam dúvidas em relação à diferença entre *design* e implementação.

Para implementar a movimentação do personagem, o programador terá que fazer o software checar, constantemente, se há alguma entrada do usuário; se houver a detecção de que a seta para cima do teclado foi acionada, por exemplo, deve ser feita uma outra checagem para decidir se há um caminho livre para o personagem se deslocar nessa direção; se a resposta for positiva, é preciso alterar a direção de movimento do personagem e gerar a imagem do *Pac-Man* com a sua boca voltada para cima. Este processo isolado pode ser visto, de maneira resumida, no pseudocódigo da Listagem 2.2.

```
while(true) {
    if(upInput) { // processamento de entradas
        if(canGoUp) {
            set_player_direction_to_up(); // atualização da lógica
            render_player_up_image(); // síntese de imagens
        }
    }
}
```

Listagem 2.2: *Exemplo ilustrativo da implementação de uma rotina do Pac-Man*

Um dos ramos do *game design* é o *level design*, que consiste no processo de construção da experiência que será oferecida diretamente ao jogador, usando os componentes fornecidos pelo *game designer* [Ada10]. Por exemplo, o *game designer* pode definir que o jogo terá espinhos e serras elétricas como obstáculos ao jogador. O *level designer* definirá, de maneira concreta, a quantidade, o posicionamento e a frequência com que esses obstáculos aparecerão nos níveis (níveis são ambientes nos quais o jogador interage com o universo do jogo, como estágios, mapas ou missões).

Se fornecidas de maneira isolada, as mecânicas de jogo servem apenas como um brinquedo para o usuário, que só poderá interagir com elas sem um objetivo definido. Para extrair o potencial das mecânicas de maneira mais completa, é necessário integrá-las ao *level design*, utilizando-as para definir desafios e objetivos, estabelecendo-se assim, a estrutura de um jogo [AD12].

Um exemplo da separação entre essas duas disciplinas de *design* pode ser observado no jogo de plataforma *Super Meat Boy*. Neste jogo, os níveis possuem um modo de dificuldade comum e um modo mais difícil. As mecânicas e os elementos de *game design* são os mesmos: o jogador pode andar, correr, deslizar nas paredes, pular do chão e pular da parede; se ele encostar nas serras, ele perde; se ele encostar no personagem rosa (que está no canto superior direito da tela), ele ganha. O que diferencia os modos é exclusivamente o *level design*, ou seja, a quantidade e o posicionamento das serras no nível. A comparação entre os modos pode ser observada na Figura 2.6.

Figura 2.6: *Comparação entre modo comum (à esquerda) e modo difícil (à direita) de um dos níveis do jogo Super Meat Boy. A diferença de dificuldade é definida pelo level design.*

Capítulo 3

Catalogação

Para embasar a escolha das mecânicas a serem desenvolvidas, foi feito um trabalho de catalogação, no qual diversos jogos relevantes para a indústria de videogames foram analisados. Neste capítulo, serão apresentadas a metodologia, os critérios de escolha e uma análise do catálogo.

3.1 Processo de filtragem

Os jogos a serem analisados foram escolhidos por meio de um processo extenso de filtragem, o qual tinha o objetivo de obter uma lista sucinta de jogos de plataforma 2D.

Previamente a este trabalho, uma lista bastante ampla de jogos relevantes para a indústria já estava sendo compilada, de maneira informal, no site *HowLongToBeat*[1]. Para que a lista pudesse ser bem utilizada neste projeto, foi feita uma pesquisa mais formal e completa, de maneira a adicionar à ela jogos importantes de plataforma.

Para filtrá-la, optou-se por uma estratégia *top-down*, na qual partiu-se da lista extensa, que continha mais de 4000 jogos dos mais variados tipos e contextos, até obter-se uma lista final contendo apenas 50 jogos de plataforma.

No site que hospeda a lista, cada jogo possui uma página própria contendo diversas informações, incluindo o seu gênero. Este fato foi utilizado para automatizar o processo de filtragem, no qual foi desenvolvido um *script* de *Web Scraping* em *Python* [Apêndice A] que analisou a lista inteira e extraiu dela apenas os jogos de plataforma.

Posteriormente, verificou-se que alguns jogos importantes não estavam presentes na lista. Foi identificado que, no site, alguns jogos estavam registrados como *Sem gênero*, ou estavam na categoria mais genérica *Action*. Portanto, o *script* foi ajustado para também considerar os jogos dessas duas categorias.

Ao final da extração dos dados, obteve-se uma lista com cerca de 600 jogos. A partir dela, foi feita uma análise manual, para manter apenas os jogos de plataforma 2D, e descartar os jogos 3D. Neste ponto, identificou-se também a existência dos jogos 2.5D, os quais foram

[1] https://howlongtobeat.com/user.php?n=Sorcker&s=games&custom=1

considerados válidos para entrarem na lista final, pois, em termos de mecânicas de jogo, eles se comportam de maneira similar aos jogos 2D. Feita esta separação, a lista diminuiu para cerca de 500 jogos.

Novamente foi feita uma seleção manual, na qual os critérios de escolha foram: relevância, inovação, abrangência e evolução. Estes critérios serão explicados mais profundamente na próxima seção. Esta foi a última etapa de filtragem, a qual gerou a lista da Figura 3.1, contendo os 50 jogos a serem analisados no trabalho:

20XX	Little Samson
Alex Kidd in Miracle World	Mark of the Ninja
Alex Kidd in Shinobi World	Mega Man
Bionic Commando (1987)	Mega Man 2
Bionic Commando (1988)	Mega Man X
Bonk's Adventure	Mega Turrican
Braid	Metal Slug
Bubble Bobble	Metroid
Castlevania	New Super Mario Bros.
Castlevania: Symphony of the Night	Ninja Gaiden (1988)
Celeste	Rogue Legacy
Cocoron	Shovel Knight
Contra	Sonic the Hedgehog (1991)
Contra III: The Alien Wars	Sonic the Hedgehog 2 (16-bit)
Cuphead	Sonic the Hedgehog 3
Donkey Kong (1981)	Strider (NES)
Donkey Kong Country	Super Mario Bros.
DuckTales	Super Mario Bros. 2
Earthworm Jim	Super Mario Bros. 3
Fez	Super Mario World
Ghosts 'n Goblins	Super Meat Boy
Hollow Knight	Super Metroid
I Wanna Be the Guy	VVVVVV
Kirby's Dream Land	Wario Land: Super Mario Land 3
Limbo	Zelda II: The Adventure of Link

Figura 3.1: *Lista de jogos de plataforma que foram analisados para a catalogação de mecânicas.*

3.2 Critérios de escolha

Os critérios que definiram a escolha dos 50 jogos estão apresentados a seguir:

- **Relevância**: jogos que apresentaram um nível elevado de destaque na indústria.
- **Inovação**: jogos que trouxeram algo novo e criativo para o cenário de jogos digitais.
- **Evolução**: jogos de diferentes épocas que se encaixam no mesmo estilo, para que fosse possível analisar o desenvolvimento das mecânicas com o passar do tempo.
- **Abrangência**: jogos de diversos subgêneros de plataforma, incluídos com o intuito de englobar mecânicas mais diversas.

O principal critério foi o da **relevância**; na lista, pode-se encontrar muitos jogos essenciais para a história dos videogames, que ajudaram a definir o gênero de plataforma, como *Donkey Kong* (1981) e *Super Mario Bros.* (1985) [EST08], que podem ser vistos na Figura 3.2.

Figura 3.2: *Jogos definidores do gênero de plataforma (Donkey Kong, à esquerda, e Super Mario Bros., à direita).*

Unindo relevância com **inovação**, notam-se os jogos das séries *Metroid* e *Castlevania*, os quais fizeram algo tão diferente que um subgênero foi criado e nomeado a partir de seus títulos: o gênero *Metroidvania*. Alguns jogos entraram na lista especificamente pelo fator **inovação**: *Cocoron* inaugurou, em 1991, a opção do jogador montar o seu personagem escolhendo suas mecâncias preferidas [1UP06], que é exatamente o tema deste projeto; *Bionic Commando* foi um dos primeiros jogos a apresentarem como mecânica principal de movimentação um gancho, ao invés do pulo; *I Wanna be the Guy* foi pioneiro no cenário de jogos de plataforma focados em dificuldade. Dois desses jogos estão apresentados na Figura 3.3.

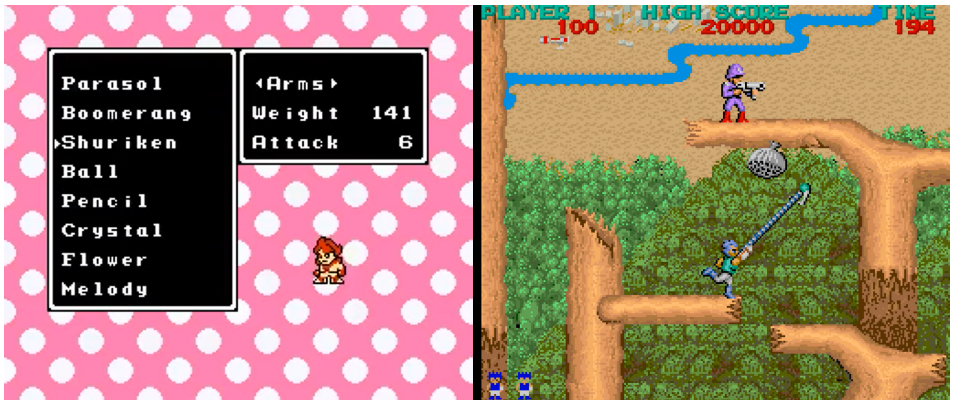

Figura 3.3: *Tela de seleção de armas do jogo Cocoron, à esquerda; mecânica de gancho do jogo Bionic Commando, à direita.*

Para analisar a **evolução** das mecânicas ao longo do tempo, contrastou-se jogos do mesmo estilo que foram lançados em épocas bem diferentes, como

- *Super Mario Bros.* (1985) vs *New Super Mario Bros.* (2006).
- *Mega Man* (1987) vs *Mega Man X* (1993) vs *20XX* (2014).
- *Contra* (1987) vs *Cuphead* (2017).
- *Metroid* (1986) vs *Super Metroid* (1994).

Na Figura 3.4, compara-se jogos com 8 anos de diferença em seus lançamentos.

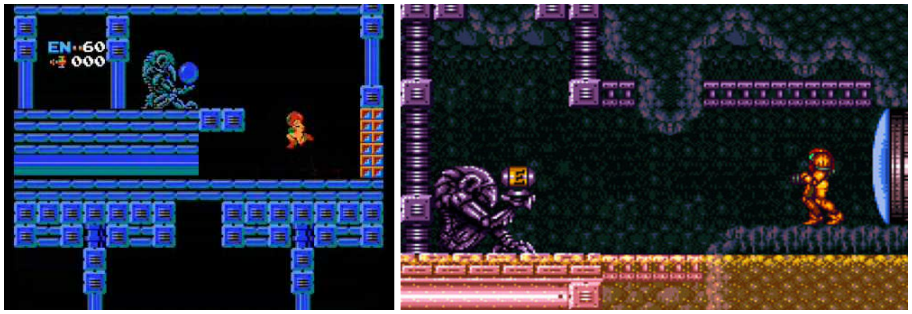

Figura 3.4: *Comparação entre os jogos Metroid e Super Metroid.*

A **abrangência** foi importante para que fosse possível extrair uma grande variedade de mecânicas. Na lista, podemos identificar os seguintes subgêneros de plataforma:

- *Puzzle-platformer*[2] (*Braid*, *Fez*, *Limbo*).
  São jogos que mantêm a estrutura de um *platformer*, porém, os desafios aparecem primariamente na forma de quebra-cabeças. Muitos jogos deste gênero apresentam uma mecânica criativa que pode ser usada de diversas maneiras diferentes, exigindo inteligência e criatividade na resolução dos *puzzles*. Em *Braid*, a mecânica especial é voltar no tempo; em *Fez*, é mudar a perspectiva. A Figura 3.5 exibe esses dois jogos.

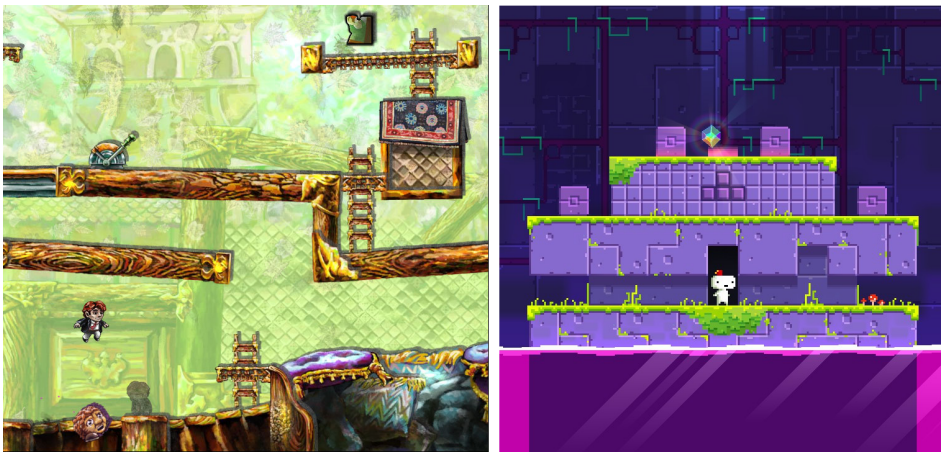

Figura 3.5: *Exemplos de jogos de plataforma que focam em resolução de puzzles (Braid, à esquerda e Fez, à direita).*

[2] https://en.wikipedia.org/wiki/Platform_game#Puzzle-platform_game

- *Run-and-gun*[3] (*Contra*, *Metal Slug*, *Cuphead*).
  Nestes jogos, apesar de ainda haver desafios relacionados a pulos precisos, é dada uma ênfase maior em combater inimigos atirando projéteis em múltiplas direções. Também é comum encontrarmos cenários chamados de *bullet hells*, nos quais o jogador é desafiado a desviar de uma grande quantidade de projéteis vindos em sua direção. Esses dois elementos estão ilustrados na Figura 3.6.

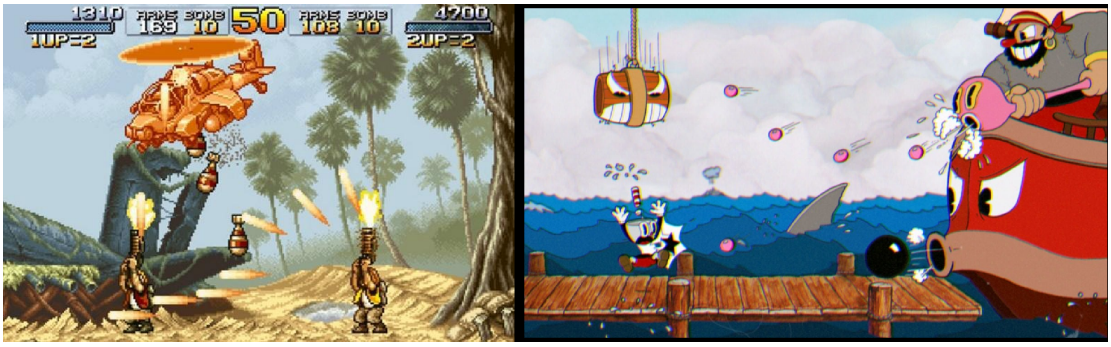

Figura 3.6: *Exemplos de tiros em múltiplas direções (Metal Slug, à esquerda) e um tipo de bullet hell (Cuphead, à direita).*

- *Metroidvania*[4] (*Super Metroid*, *Castlevania: Symphony of the Night*, *Hollow Knight*).
  Este gênero inovador veio na contramão de diversos conceitos que já estavam solidificados na indústria. A estrutura de níveis fechados e independentes é abandonada, dando lugar a um mapa gigantesco, interconectado, no qual o jogador vai e volta pelos caminhos de maneira não-linear. Geralmente, algumas áreas deste mapa estão inacessíveis para o jogador no começo do jogo e vão sendo liberadas assim que novas habilidades vão sendo adquiridas, como pode-se observar na Figura 3.7.

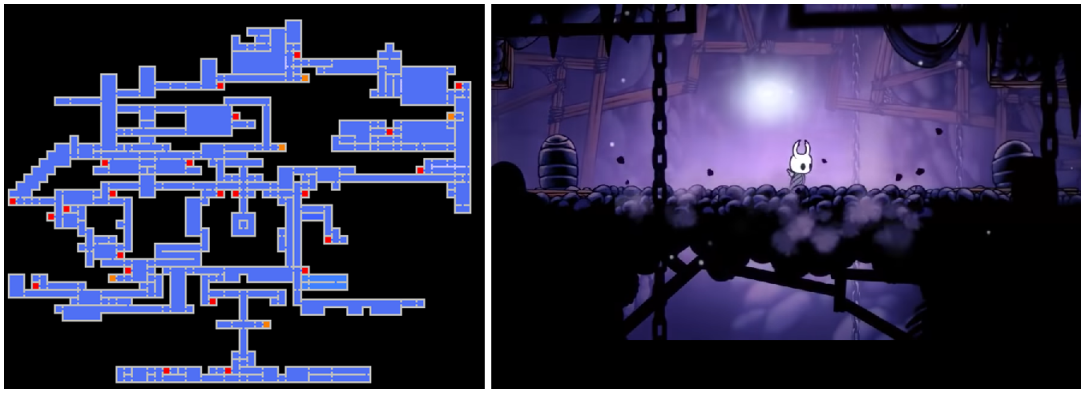

Figura 3.7: *Exemplo de um mapa grande interconectado (Castlevania: Symphony of The Night, à esquerda) e de uma área que só pode ser acessada depois do jogador obter o poder de se lançar para baixo (Hollow Knight, à direita).*

[3] https://en.wikipedia.org/wiki/Platform_game#Run-and-gun_platform_game

[4] https://en.wikipedia.org/wiki/Metroidvania

- *Roguelite platformers*[5] (*Rogue Legacy*, *20XX*).
  São jogos de plataforma que contêm alguns elementos inspirados em *Rogue*[6], um jogo muito influente de 1980. Suas principais características são: geração procedural do mapa e morte permanente do personagem[7]. Alguns exemplares deste gênero estão apresentados na Figura 3.8.

Figura 3.8: *Exemplo da continuidade do uso de mecânicas de plataforma (20XX, à esquerda) e de um mapa gerado proceduralmente (Rogue Legacy, à direita).*

- *Masocore*[8] (*I Wanna be the Guy*, *Super Meat Boy*, *Celeste*).
  Este subgênero justifica o seu nome (junção das palavras *masochism* e *hardcore*) dando um destaque especial à dificuldade do jogo, exigindo do jogador paciência, habilidade e aprendizagem por tentativa e erro. As principais características deste gênero são: vidas infinitas e a abundância de obstáculos; a última pode ser observada na Figura 3.9.

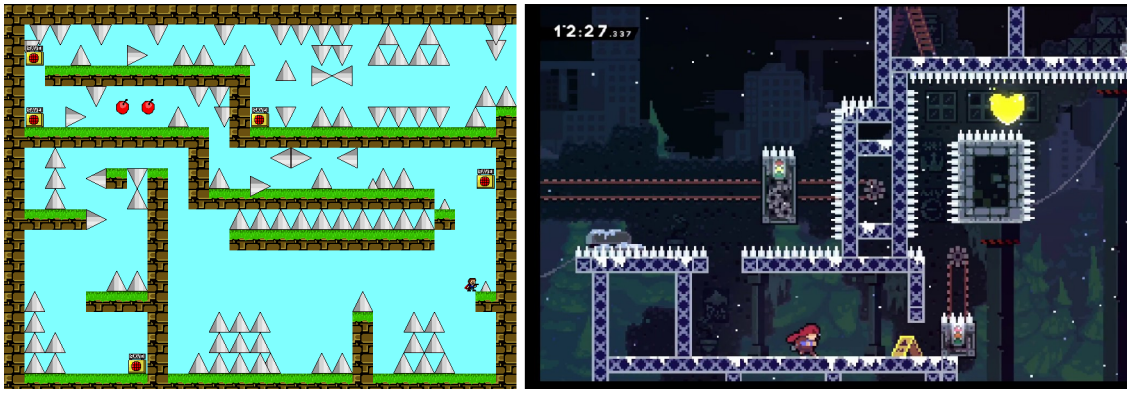

Figura 3.9: *Exemplo da abundância de obstáculos (I Wanna be the Guy, à esquerda e Celeste, à direita).*

[5] https://en.wikipedia.org/wiki/Roguelike#Rogue-lites_and_procedural_death_labyrinths

[6] https://en.wikipedia.org/wiki/Rogue_(video_game)

[7] https://en.wikipedia.org/wiki/Permadeath

[8] https://www.giantbomb.com/masocore/3015-1165/

3.3 Análise dos jogos

A maioria dos jogos da lista necessita de um tempo relativamente curto para serem concluídos: menos de duas horas. Portanto, nestes casos, analisou-se vídeos no *Youtube* chamados de *Longplays*[9], nos quais o jogo é apresentado sem cortes, do começo ao fim. No caso de jogos com duração muito longa, foi escolhido o método de analisar suas mecânicas por meio de sites que documentam o conteúdo destes jogos, como os seguintes: Hollow Knight Wiki e 20XX Wiki.

Sempre que uma mecânica nova era identificada, ela era adicionada ao catálogo dentro de sua categoria específica, juntamente com a informação de qual jogo possuía aquela mecânica e, opcionalmente, exemplos de como ela foi implementada:

```
-Categoria:
    Mecânica [Jogo 1 (exemplo), Jogo 2 (exemplo)].
```

Mais de 200 mecânicas foram catalogadas e podem ser acessadas no seguinte *link*: Catálogo de Mecânicas.

As mecânicas relacionadas às ações do jogador foram divididas nas seguintes categorias: movimentação, ataque, defesa, utilidade e *puzzle*. Já as demais mecânicas catalogadas se referem às plataformas, aos obstáculos e demais elementos que compõem o cenário.

Dentre as mecânicas de ação do jogador, destaca-se a predominância das de movimentação e ataque em detrimento das demais. Esse resultado pode ser visto no gráfico da Figura 3.10.

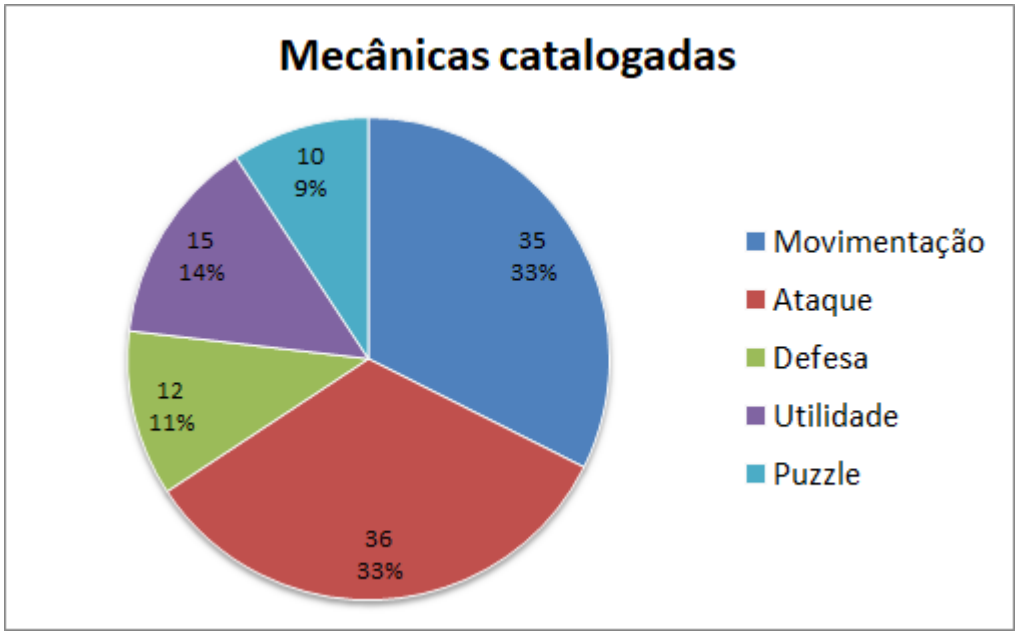

Figura 3.10: *Gráfico contendo informações sobre as mecânicas catalogadas referentes às ações do personagem, indicando as quantidades e suas devidas proporções.*

[9] https://en.wikipedia.org/wiki/Longplay_(video_games)

Verificou-se, também, que há uma grande repetição de mecânicas entre os jogos. A frequência com que novas mecânicas eram adicionadas ao catálogo foi diminuindo conforme os jogos eram analisados, até chegar ao ponto em que alguns não adicionaram nenhuma mecânica nova ao catálogo.

A evolução das mecânicas em jogos do mesmo estilo se mostrou conservadora. Apesar de já ser esperado que jogos da mesma série mativessem seus elementos principais, poucas mecânicas foram adicionadas. Ao compararmos, por exemplo, jogos da série *Super Mario Bros* com mais de 20 anos de diferença, houve apenas a adição do *wall jump*, mecânica bastante presente nos jogos modernos, e uma leve alteração no mergulho em direção ao chão, que já existia no *Super Mario World*. Nota-se que os esforços se concentram mais na parte artística: os gráficos evoluem drasticamente ao longo dos anos, ao contrário das mecânicas.

Concluiu-se que, em geral, a variação entre os jogos não se dá pela diversidade de mecânicas, e sim pela parte gráfica e pela criação de novos ambientes, contextos, narrativas e design de níveis.

Capítulo 4

Design do Jogo

Neste capítulo, serão apresentadas as decisões e os elementos de *design* que definiram o jogo feito como projeto deste trabalho de conclusão de curso. O projeto consiste de um jogo digital 2D chamado *Mechanical Playground*, do gênero de plataforma. Seguindo as características definidoras desse gênero, o jogador inicia o jogo podendo utilizar duas mecânicas fundamentais, que não podem ser desativadas: andar e pular.

Ao longo da experiência, o usuário será gradualmente introduzido a outras 18 mecânicas, organizadas nas categorias de movimentação, ataque e utilidade. Depois de passar por níveis de aprendizado, o jogador deverá escolher um conjunto de 6 mecânicas e ficará livre para mudá-lo a qualquer momento. Foram desenvolvidos 15 níveis, contendo diversos obstáculos como espinhos, serras elétricas e 4 tipos de inimigos. Ao encostar em um desses obstáculos, o jogador é derrotado e o nível se reinicia. Para vencer, ele deve utilizar de maneira hábil as mecânicas escolhidas, superar os obstáculos e alcançar o troféu no final do nível. Na Figura 4.1 há o exemplo de um nível do jogo.

Figura 4.1: *Exemplo de nível do Mechanical Playground.*

4.1 Personagem principal

O jogador interage com o jogo controlando o personagem principal, que está ilustrado na Figura 4.2.

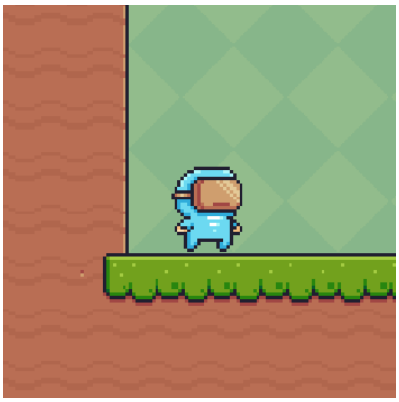

Figura 4.2: *Personagem principal.*

4.2 Obstáculos

Para compor os desafios apresentados ao jogador, foram desenvolvidos diversos obstáculos comuns ao gênero de plataforma. Alguns são não-letais, ou seja, apenas se apresentam como um bloqueio ao progresso do jogador, enquanto outros são letais, os quais causam a derrota do jogador se houver contato.

4.2.1 Obstáculos não-letais

- Terreno

  Em muitos locais do jogo, o terreno se apresenta como um obstáculo ao progresso do jogador. Elevações baixas, como as da Figura 4.3, podem ser superadas facilmente com um simples pulo, porém, elevações mais altas exigem a utilização de outras mecânicas.

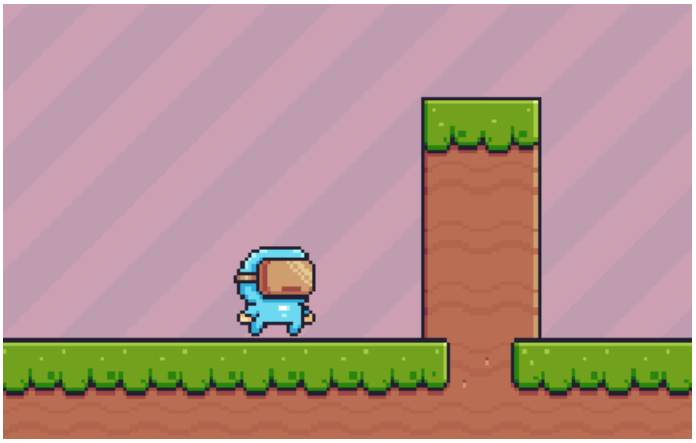

Figura 4.3: *Terreno funcionando como um obstáculo não-letal.*

- Portão

  O portão é um obstáculo não-letal que, em conjunto com o terreno, bloqueia totalmente o caminho do jogador. Ele pode ser superado de algumas maneiras diferentes, sendo a mais direta a utilização de uma chave que o jogador obtém nos arredores do portão, o que está ilustrado na Figura 4.4. Também há a possibilidade de superá-lo utilizando outras mecânicas como o *Blink* (Teleporte) ou a Explosão.

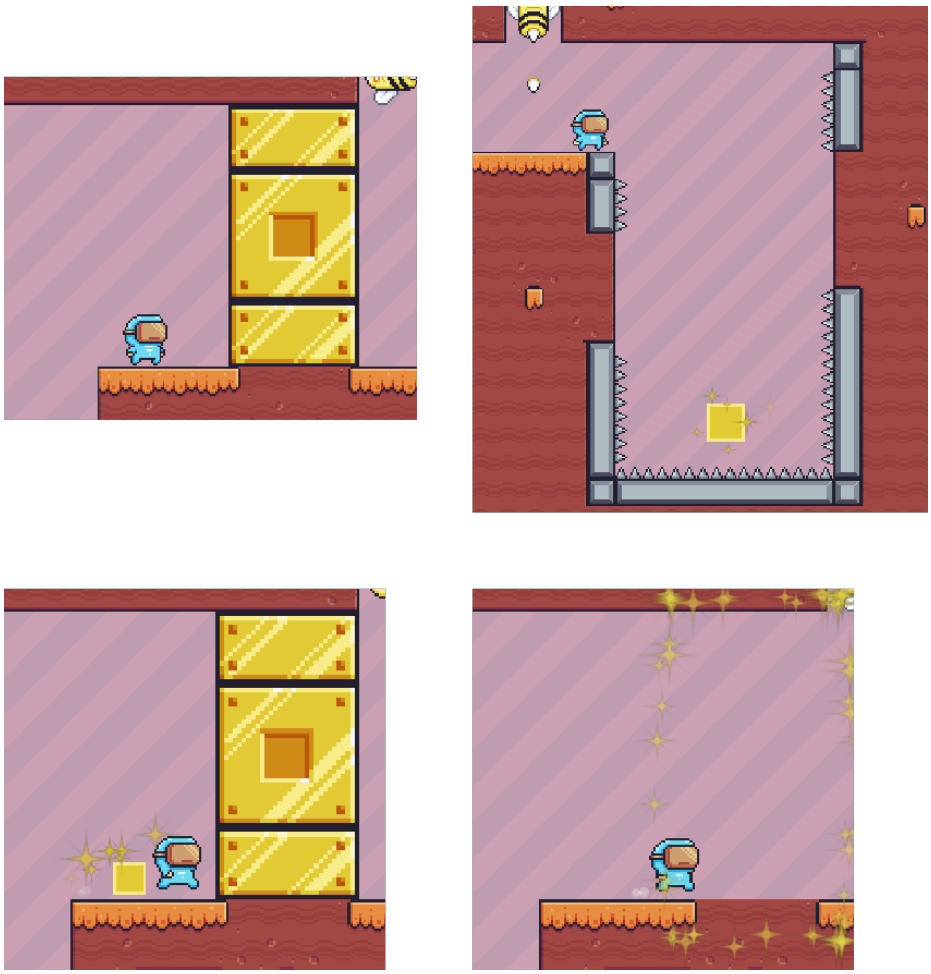

Figura 4.4: *Portão obstruindo o caminho do jogador; chave (quadrado amarelo) em um local desafiador; e portão sendo aberto com a utilização da chave.*

4.2.2 Obstáculos letais

- Espinhos

  São obstáculos letais estáticos com área de colisão retângular e relativamente pequena. Uma área do jogo em que eles são os principais obstáculos está apresentado na Figura 4.5.

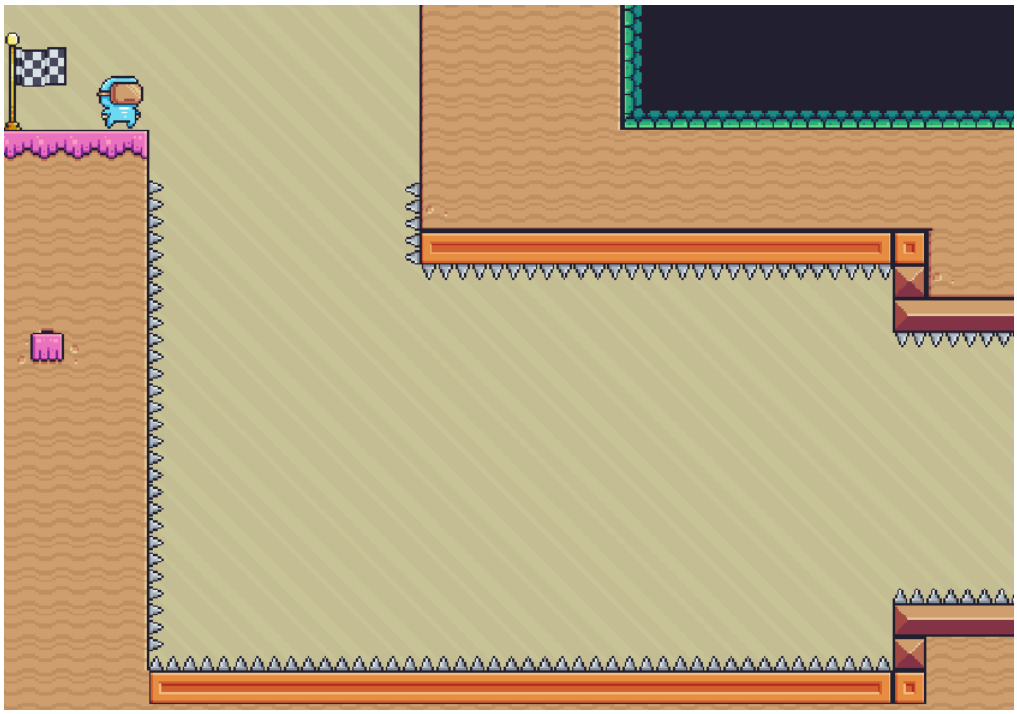

Figura 4.5: *Área com muitos espinhos.*

- Serra Elétrica

  É um obstáculo letal estático com área de colisão circular e relativamente grande. Uma área com serras pode ser visualizada na Figura 4.6.

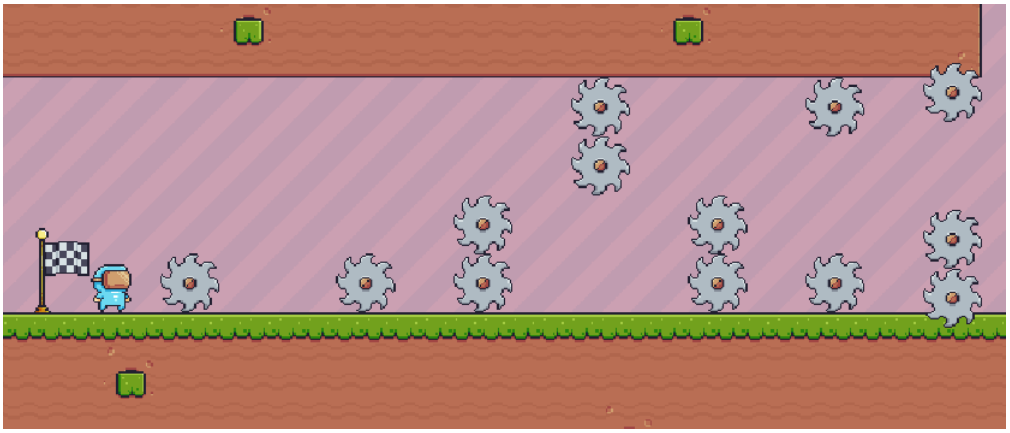

Figura 4.6: *Área na qual os principais obstáculos são as serras elétricas.*

- *Bee*

  Bee é uma abelha que pode se manter estática ou se movimentar, tanto de maneira horizontal, quanto vertical. Periodicamente, ela atira um projétil, que pode variar tanto em velocidade quanto em frequência de lançamento. Um exemplo dela em ação está apresentado na Figura 4.7.

Figura 4.7: *Abelha lançando um projétil para baixo e se movimentando para a direita.*

- *Red Bee*

  Red Bee é uma abelha vermelha, com comportamento muito parecido com o da abelha comum. Seu diferencial é a capacidade de lançar projéteis etéreos, que ultrapassam o terreno. Esta propriedade pode ser utilizada para exigir que o jogador lide com os projéteis, quando as abelhas estão posicionadas atrás de paredes compridas, tornando-as inalcançáveis. Este cenário está ilustrado na Figura 4.8.

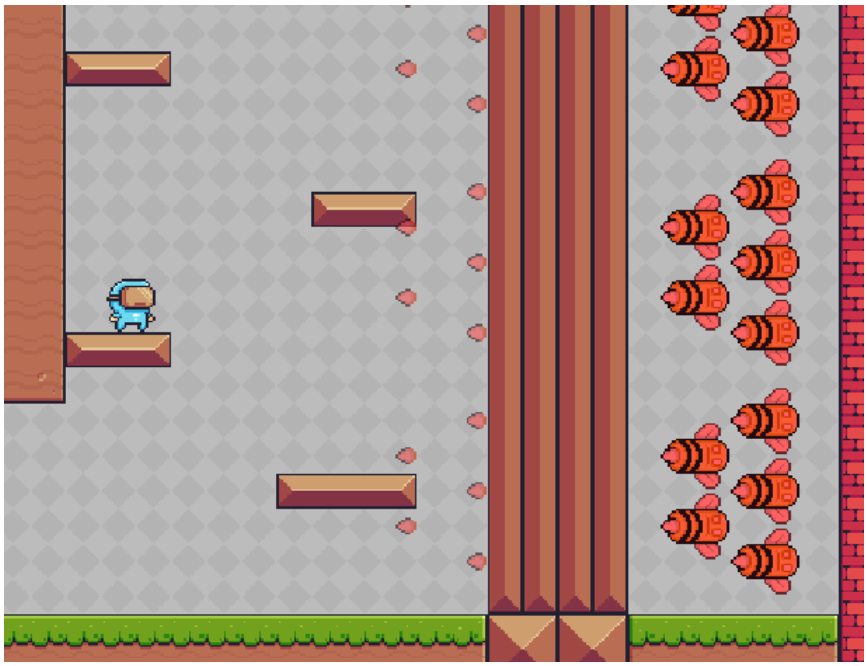

Figura 4.8: *Abelhas vermelhas lançando projéteis etéreos.*

- *Spiky*

  Spiky é um inimigo estático que, periodicamente, lança espinhos em 5 direções, conforme ilustrado na sequência de quadros da Figura 4.9.

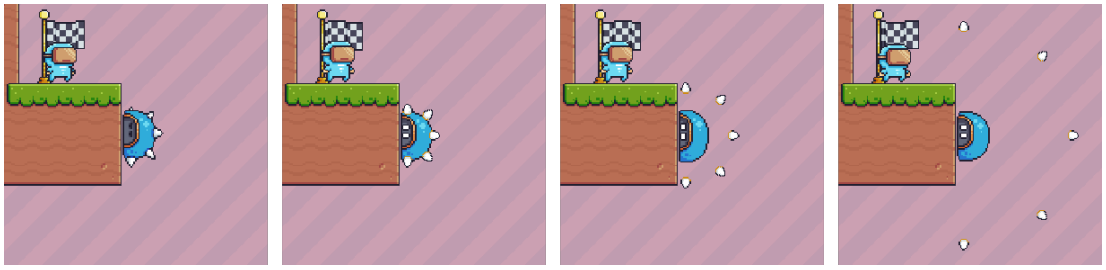

Figura 4.9: *Spiky lançando seus espinhos.*

- *Trunk*

  Trunk é um inimigo que se movimenta horizontalmente e, quando encontra um obstáculo ou o fim do terreno, inverte o sentido do movimento. Quando detecta o jogador em sua frente, ele lança projéteis periodicamente em direção ao jogador, como apresentado na Figura 4.10.

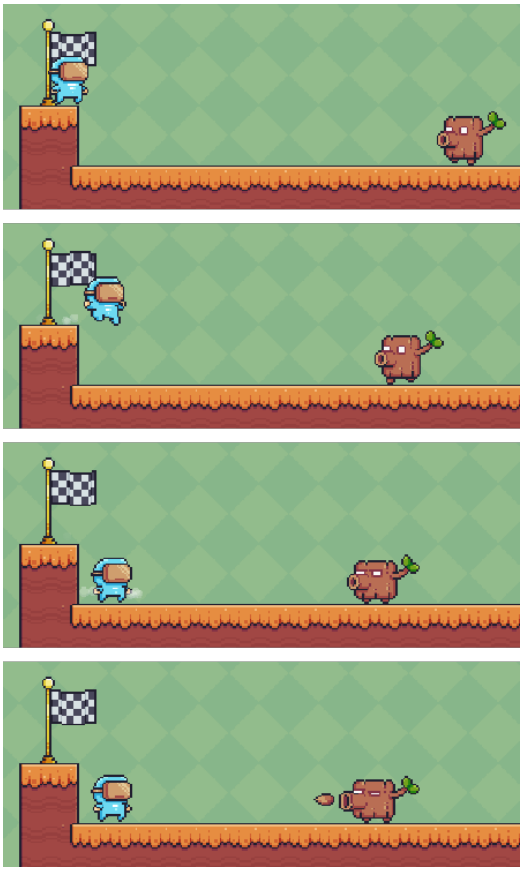

Figura 4.10: *Trunk andando tranquilamente até ver o personagem na sua frente.*

4.3 Mecânicas do Personagem

Todas as mecânicas implementadas no jogo tiveram como inspiração as que foram catalogadas no processo descrito no capítulo anterior. As mecânicas básicas são inerentes ao personagem e não podem ser desativadas. As demais poderão ser ativadas ou desativadas, de acordo com a escolha do jogador.

4.3.1 Mecânicas Básicas

- *Walk* (Andar)

  Permite o deslocamento horizontal do personagem, ilustrado na Figura 4.11.

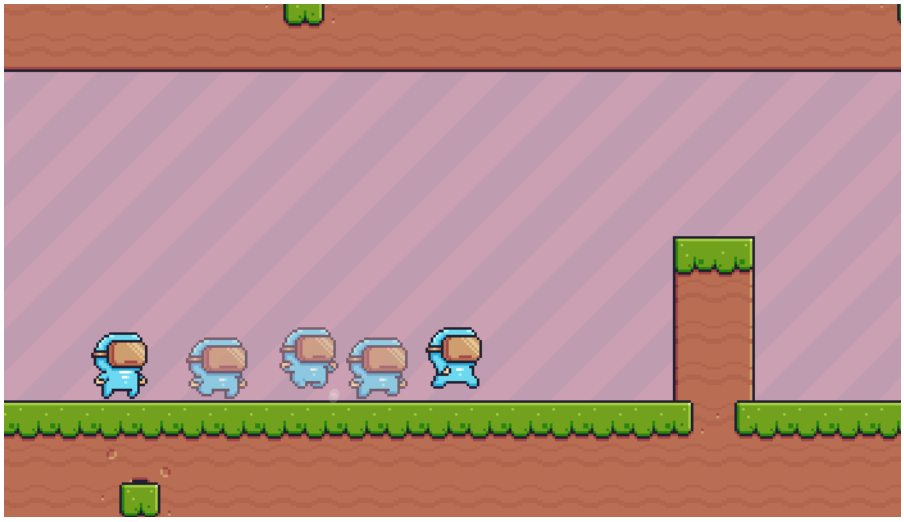

Figura 4.11: *Personagem andando.*

- *Jump* (Pular)

  Permite o deslocamento vertical do personagem, exibido na Figura 4.12.

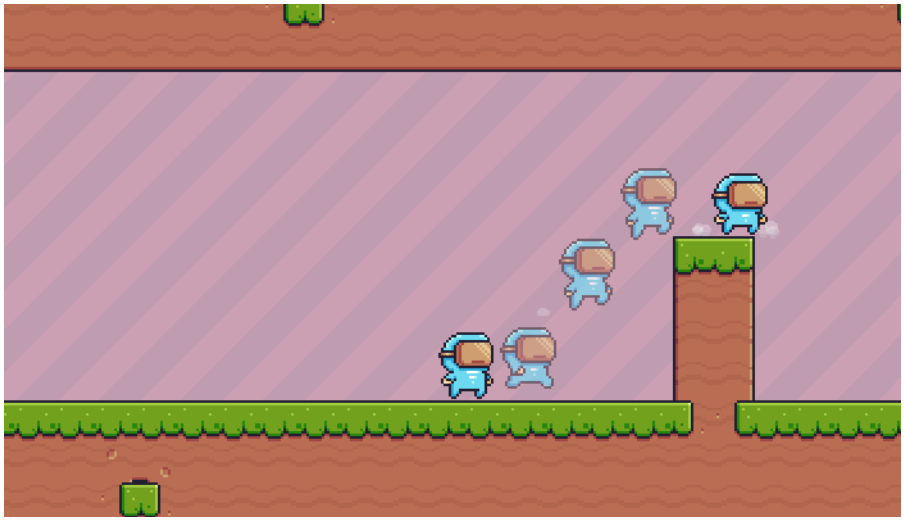

Figura 4.12: *Personagem pulando.*

4.3.2 Mecânicas de Movimentação

- *Double Jump* (Pulo Duplo)

  Permite que o jogador pule mais uma vez, no ar, conforme mostra a Figura 4.13.

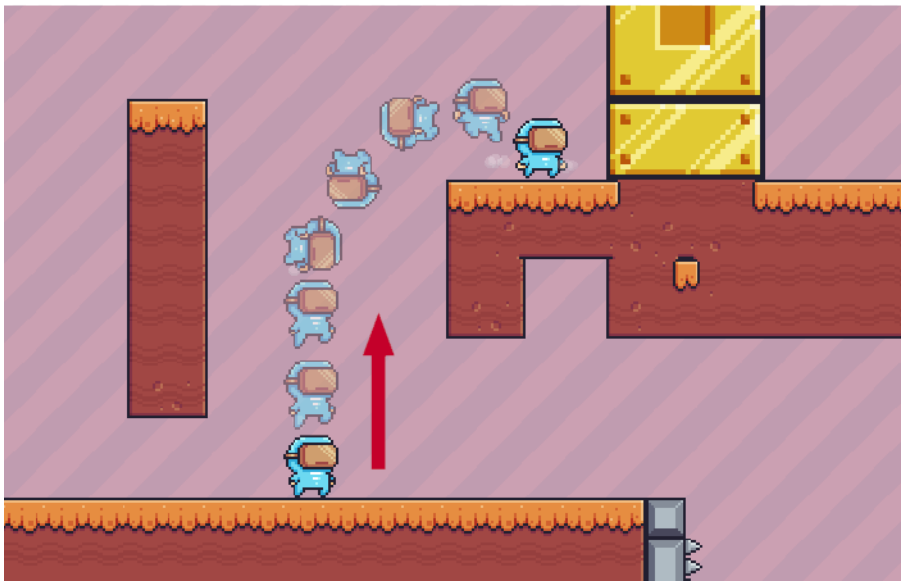

Figura 4.13: *Pulo duplo.*

- *Wall Slide* (Deslizar na parede)

  Permite que o jogador deslize na parede, como indica a Figura 4.14.

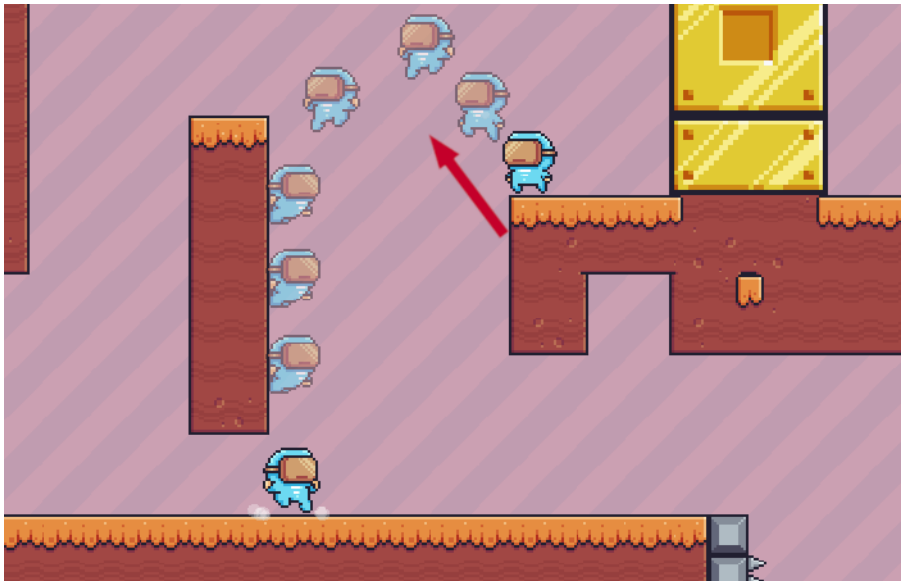

Figura 4.14: *Personagem deslizando na parede, com velocidade menor do que a da queda livre.*

- *Wall Jump* (Pular da parede)

  Permite que o jogador pule enquanto desliza na parede, o que está exposto na Figura 4.15.

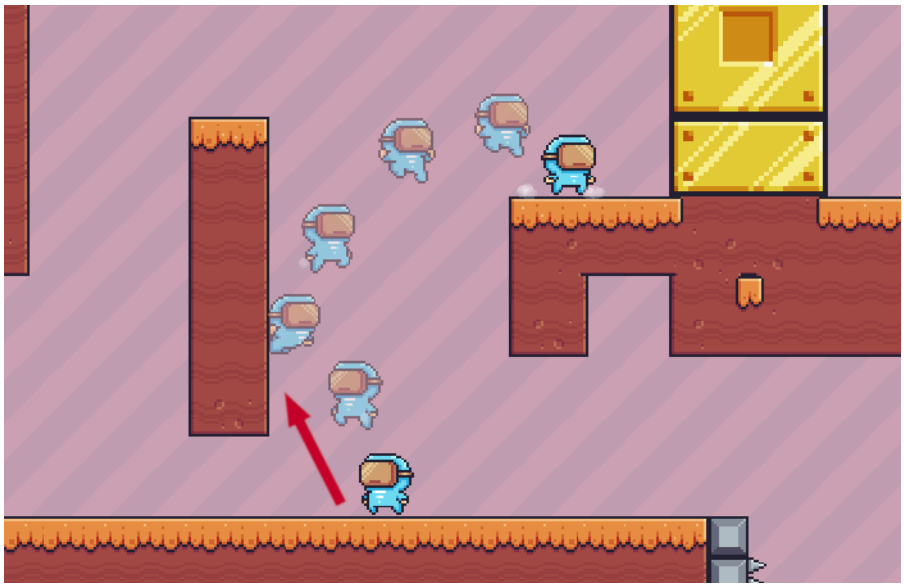

Figura 4.15: *Personagem pulando da parede.*

- *Dash* (Impulso)

  Personagem dá um impulso para frente, da maneira como a Figura 4.16 indica.

Figura 4.16: *Personagem utilizando o dash.*

- *Ethereal Dash* (Impulso Etéreo)

  Efeito passivo que altera o *dash*, fazendo com que o personagem não colida com obstáculos letais enquanto utiliza o *dash*, como retrata a Figura 4.17.

Figura 4.17: *Personagem passando pelas serras elétricas sem colidir com elas, enquanto utiliza o dash.*

- *Blink* (Teleporte)

  Permite ao jogador realizar um teleporte à curta distância, na direção indicada. É possível ultrapassar até mesmo portões, como evidencia a Figura 4.18.

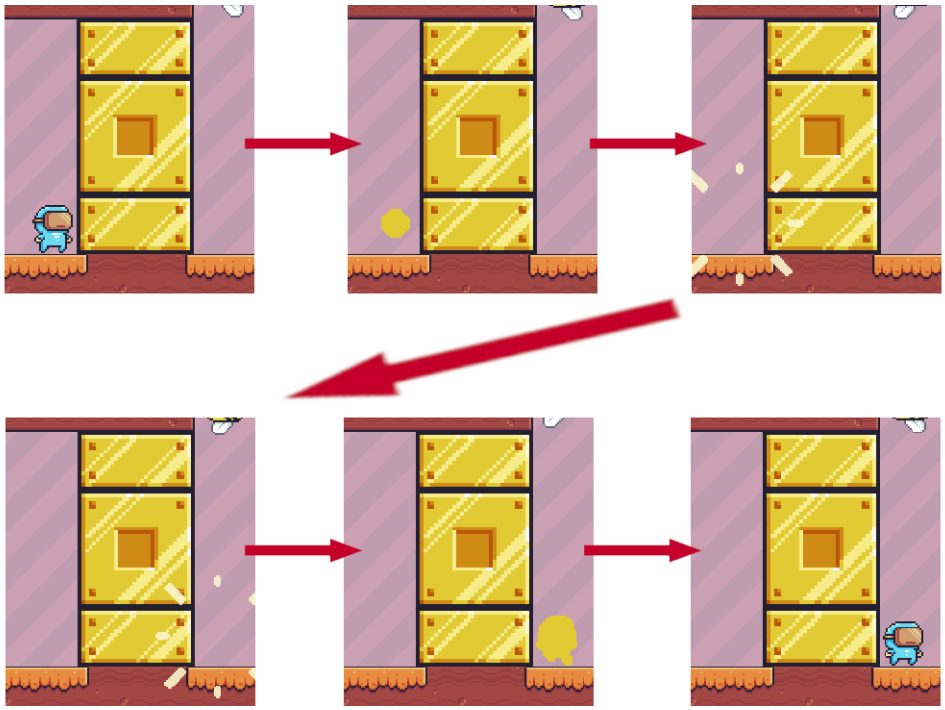

Figura 4.18: *Personagem se teleportando para o outro lado do portão.*

4.3.3 Mecânicas de Ataque

- *Attack* (Ataque)

  Mecânica básica de ataque. Permite ao jogador atacar em 4 direções (cima, baixo, esquerda, direita), criando um arco que atinge inimigos e obstáculos. O funcionamento da mecânica pode ser visto na Figura 4.19.

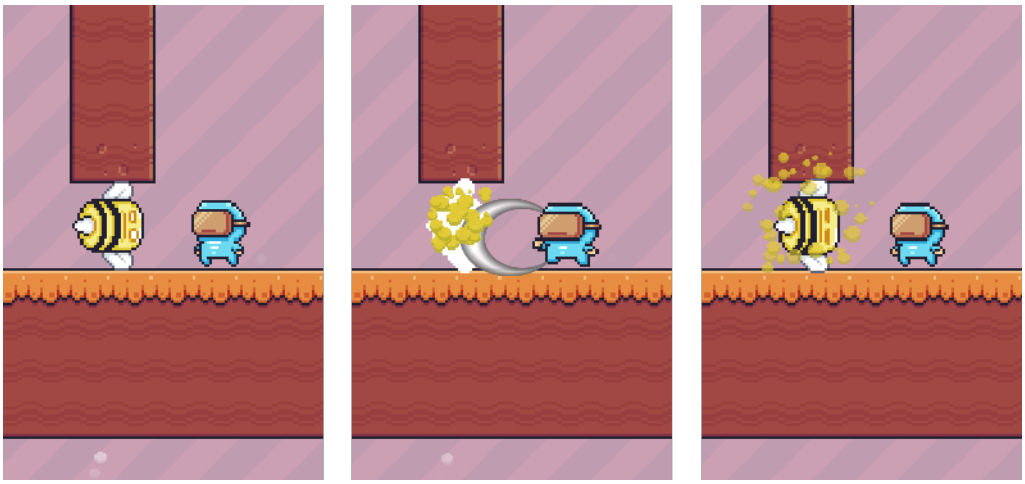

Figura 4.19: *Personagem atacando um inimigo.*

- *Range Boost* (Ampliação de alcance)

  Efeito passivo que amplia o alcance do ataque, podendo atingir até mesmo inimigos do outro lado do terreno, como mostra a Figura 4.20.

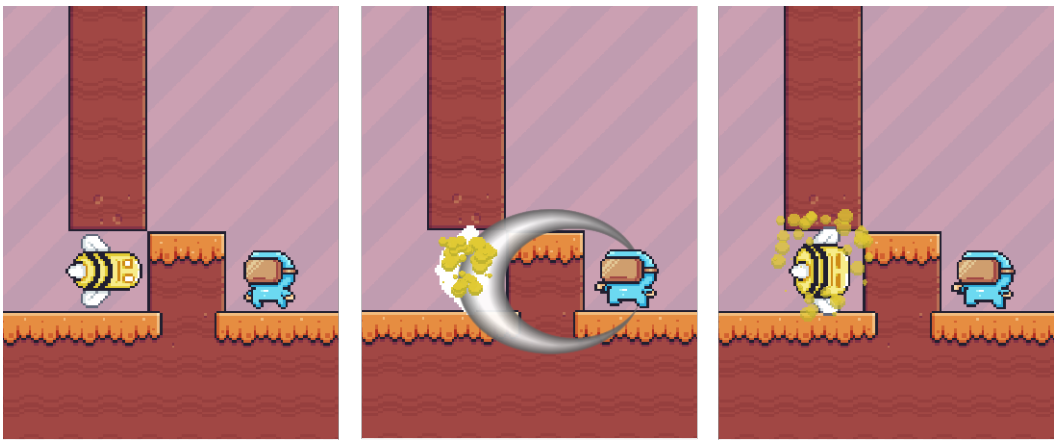

Figura 4.20: *Personagem atacando com alcance aumentado.*

- *Destroy Projectile* (Destruir projétil)

  Faz com que os ataques do jogador destruam projéteis, o que está exposto na Figura 4.21.

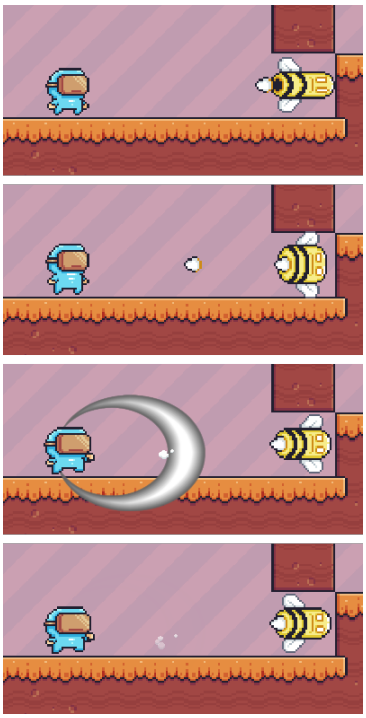

Figura 4.21: *Personagem destruindo um projétil lançado por um inimigo.*

- *Pogo Jump* (Pulo Pogo)

  Quando o personagem está no ar, ataca para baixo e atinge um obstáculo como uma serra elétrica ou um inimigo, ele é impulsionado para cima. Essa sequência de ações pode ser entendida melhor observando-se a Figura 4.22.

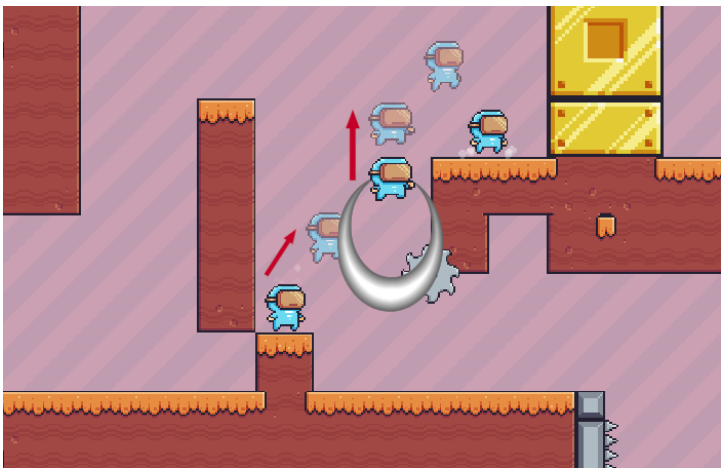

Figura 4.22: *Personagem pula e ataca para baixo, atingindo a serra elétrica. Então, ele é impulsionado para cima e consegue alcançar a plataforma alta.*

- *Gun Boots* (Botas armadas)

  Permite ao personagem atirar pelas botas em direção ao chão, dando-lhe um impulso para cima e, ao mesmo tempo, servindo como arma para combater seus inimigos, como ilustra a figura 4.23.

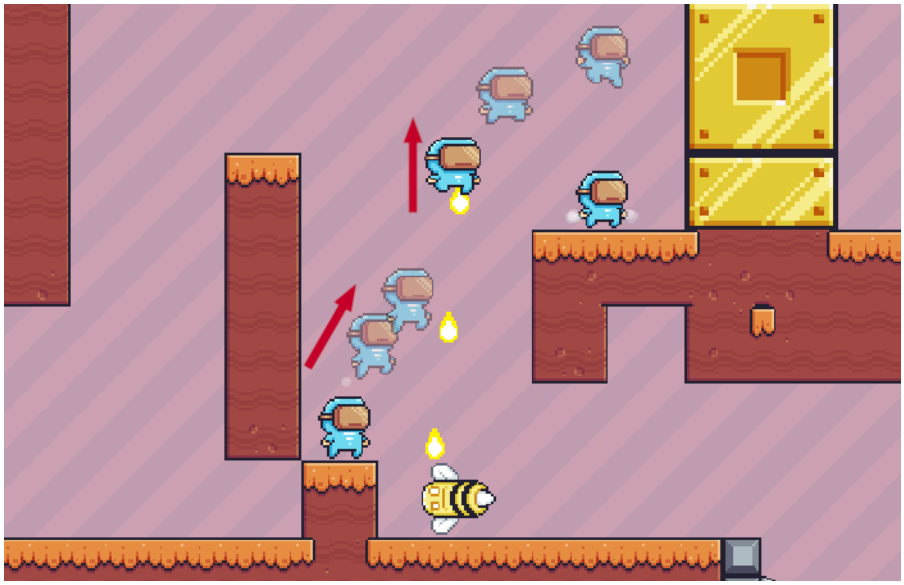

Figura 4.23: *Personagem pulando e atirando pelas botas, o que dá impulso para ele subir na plataforma alta e também atingir seu inimigo.*

- *Explosion* (Explosão)

  Cria uma explosão em uma de 4 direções (cima, baixo, esquerda, direita), que destrói obstáculos como inimigos, serras e portões. Uma colagem contendo essas 3 situações está apresentada na Figura 4.24.

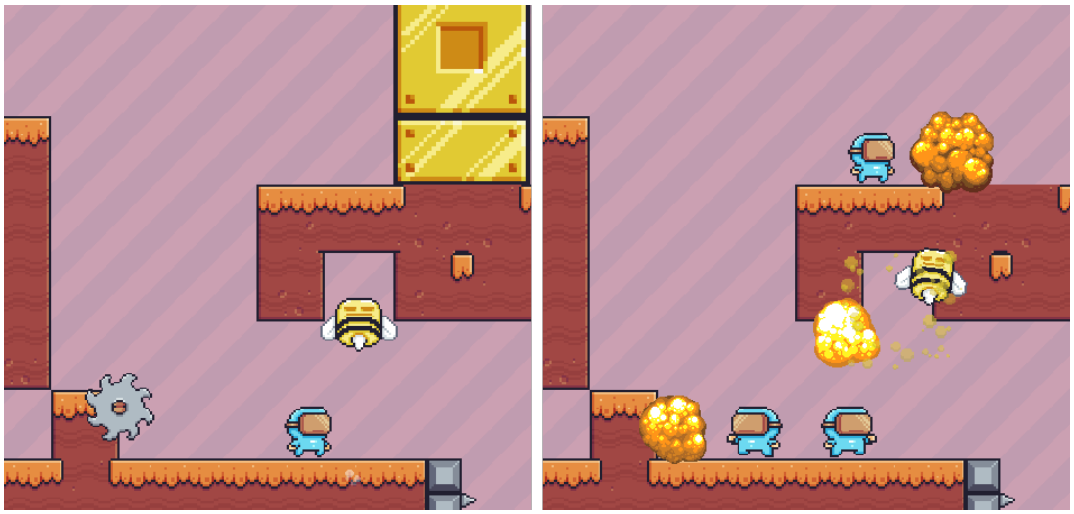

Figura 4.24: *Personagem explodindo diversos obstáculos.*

4.3.4 Mecânicas de Utilidade

- *Shield* (Escudo)

  O jogador pode ativar um escudo que é consumido ao entrar em contato com um projétil, protegendo o personagem. Esse sistema pode ser visto na Figura 4.25. O usuário só poderá reativar o escudo depois de 1 segundo.

Figura 4.25: *Personagem se defendendo de um projétil com o escudo.*

- *Parry* (Bloqueio)

  Efeito passivo que altera o comportamento do escudo: se o jogador ativá-lo bem na hora do impacto, além do projétil ser bloqueado, o escudo não será consumido e o jogador poderá voltar a utilizá-lo imediatamente, como ilustra a Figura 4.26.

Figura 4.26: *Personagem bloqueando múltiplos projéteis.*

- *Reflect Projectile* (Refletir projétil)

  Habilidade que altera o funcionamento do escudo, fazendo com que os projéteis bloqueados sejam refletidos de volta ao inimigo. A sequência de quadros da Figura 4.27 demonstra o funcionamento dessa mecânica.

Figura 4.27: *Personagem bloqueando um projétil, que é refletido, eliminando o inimigo.*

- *Spike Invulnerability* (Invulnerabilidade à espinhos)

  Efeito passivo que transforma os espinhos em obstáculos não-letais, como pode-se observar na Figura 4.28.

Figura 4.28: *Personagem não sendo afetado pelos espinhos.*

- *Saw Invulnerability* (Invulnerabilidade à serras)

  Efeito passivo que transforma as serras em obstáculos não-letais, como é mostrado na Figura 4.29.

Figura 4.29: *Personagem não sendo afetado pelas serras.*

- *Create Platform* (Criar plataforma)

  Possibilita ao jogador criar plataformas para alcançar terrenos altos e bloquear projéteis. A sequência de quadros da Figura 4.30 ilustra a utilização dessa mecânica.

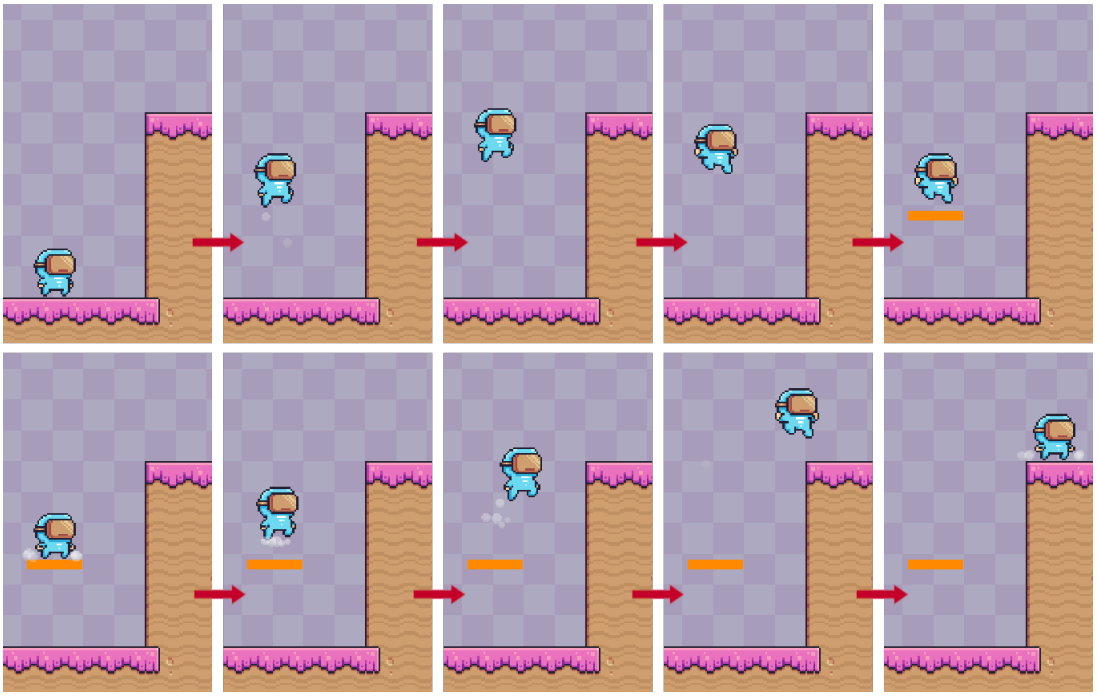

Figura 4.30: *Personagem pula, cria a plataforma enquanto está no ar e pula a partir dela para atingir a plataforma alta.*

4.4 Mecânicas de Progressão

O jogador progride em um dado nível buscando alcançar o troféu (condição de vitória) e evitando o contato com os obstáculos letais (condição de derrota). Para ajudá-lo nessa jornada, há a presença de bandeiras que salvam seu progresso na fase.

- Condição de vitória

  O jogador satisfaz a condição de vitória se conseguir alcançar o troféu, exibido na Figura 4.31. Ele só pode ser adquirido depois que todos os desafios forem superados.

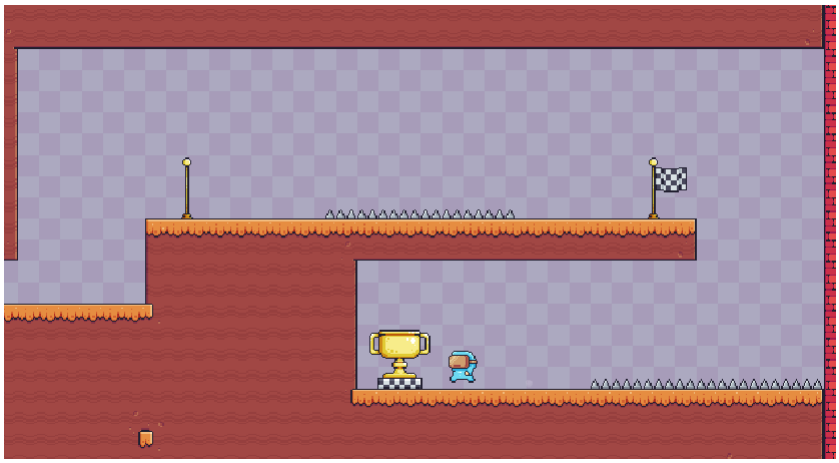

Figura 4.31: *Personagem andando em direção ao troféu, depois de ter vencido os desafios da fase.*

- Condição de derrota

  O jogador satisfaz a condição de derrota se ele colidir com um obstáculo letal, como na Figura 4.32. Uma breve animação de morte ocorre e o nível é reiniciado.

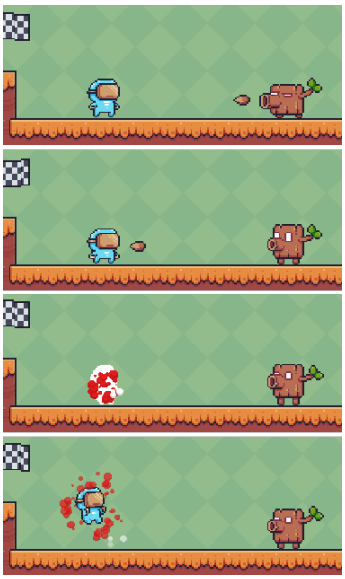

Figura 4.32: *Personagem sendo derrotado por entrar em contato com um projétil.*

- *Checkpoints*

  Quando o jogador é derrotado, o nível é reiniciado e o personagem normalmente reaparece no início da fase. Porém, se ele tiver ativado alguma das bandeiras espalhadas pelo nível, ele reaparecerá no local da bandeira. Ou seja, as bandeiras (também conhecidas como *Checkpoints*[1]) salvam seu progresso no nível. Este artifício, que pode ser visto na Figura 4.33, também é utilizado como recompensa pelo jogador ter vencido um desafio.

Figura 4.33: *Bandeira inativada, à esquerda, e ativada, à direita.*

4.5 Interface

A tela do jogo, exibida na Figura 4.34, apresenta apenas dois elementos de interface: no topo, há um indicador que informa o número do nível, e na esquerda, há um painel que mostra as mecânicas que o jogador possui no momento, além da opção de expandi-lo para informar quais botões acionam as habilidades.

Observação: no contexto do jogo, o termo *mecânica* foi substituído por *habilidade* (ou, *skill*), pois este é um termo mais familiar para um jogador comum.

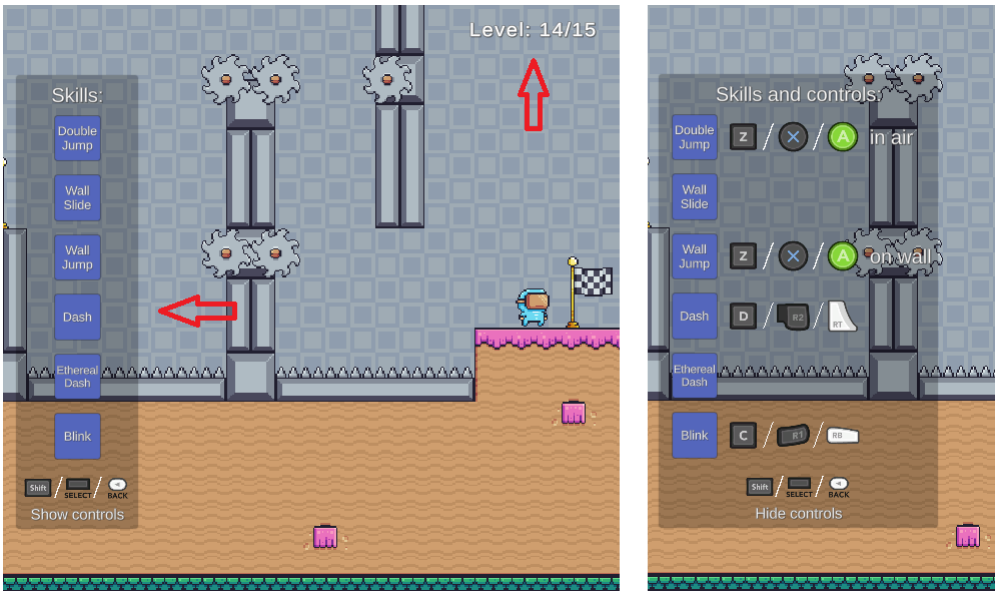

Figura 4.34: *Indicador de nível e painel de habilidades. À direita, painel de habilidades expandido, informando os controles.*

[1] https://pt.wikipedia.org/wiki/Jogo_salvo#Checkpoints

Além da interface da tela de jogo, há o menu de seleção de mecânicas, apresentado na Figura 4.35. Nele, o jogador pode navegar com o mouse, o teclado ou o controle para escolher seu conjunto de 6 mecânicas. Ao selecionar uma mecânica, são mostradas informações sobre ela e também um vídeo de demonstração.

Foi seguido o sistema de árvore de habilidades, no qual algumas mecânicas possuem requisitos para poderem ser escolhidas, como a *Reflect Projectile* que requer a *Parry*, que, por sua vez, requer a *Shield*:

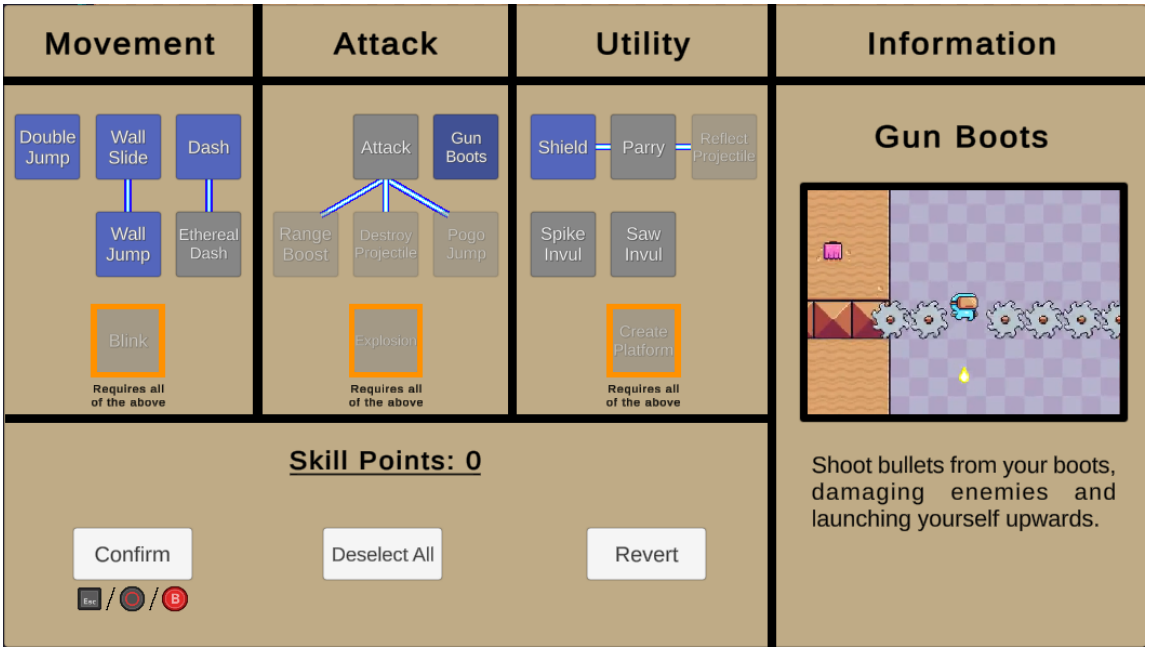

Figura 4.35: *Menu de escolha de mecânicas, com as mecânicas escolhidas na cor azul, a mecânica selecionada em azul escuro e as demais na cor cinza.*

Cada categoria possui uma mecânica especial (borda laranja) que requer todas as outras da mesma categoria.

4.6 Level Design

Apesar das mecânicas de jogo serem o foco deste projeto, isoladas elas não possuem muito sentido. A integração com o *level design* providencia o contexto necessário para que seu potencial possa ser bem aproveitado. Portanto, uma grande atenção foi dada para o desenvolvimento do *level design* neste trabalho. Projetar os níveis para se adequarem à utilização das mais de 20 mecânicas desenvolvidas foi um dos maiores desafios do projeto.

O desenvolvimento das mecânicas foi baseado nos níveis em que o jogador poderia escolher o conjunto de 6 mecânicas, dentre as 18 disponíveis. Porém, disponibilizar o jogo para os usuários contendo apenas esses níveis, com todos os recursos disponíveis, seria uma má aplicação das boas práticas de *level design*. No livro *Fundamentals of Game Design*, o autor *Ernest Adams* diz: "Não disponibilize todos os recursos do jogo de uma só vez. Se

por acaso o jogador selecionar, por acidente, uma habilidade que você ainda não introduziu, que produz um efeito na tela que ele não entende, isso só vai confundir o jogador. Desative os recursos até que o tutorial os introduza."[Ada10].

No contexto dos videogames, tutorial (ou, níveis de tutorial) são níveis que ensinam o jogador como jogar. Antigamente, as mídias de distribuição (disquetes e cartuchos) não podiam armazenar níveis para ensinar o jogador, portanto, os jogos geralmente vinham sem tutorial; a maneira de se ensinar como jogar era feita por meio de um manual físico. Hoje em dia, com os avanços da tecnologia, essa prática se tornou extremamente incomum. Os jogadores estão acostumados a iniciar o jogo imediatamente, sem ter que ler nada, aprendendo a jogar na prática [Ada10].

4.6.1 Níveis de tutorial

Foram desenvolvidos 13 níveis de tutorial, como mostra a tabela da Figura 4.36.

Nível 1:	Walk, Jump, Double Jump
Nível 2:	Dash
Nível 3:	Ethereal Dash
Nível 4:	Wall Slide
Nível 5:	Wall Jump
Nível 6:	Blink
Nível 7:	Attack, Range Boost, Destroy Projectile
Nível 8:	Pogo Jump
Nível 9:	Gun Boots
Nível 10:	Explosion
Nível 11:	Shield, Parry, Reflect Projectile
Nível 12:	Create Platform
Nível 13:	Spike Invul, Saw Invul

Figura 4.36: *Tabela indicando as mecânicas ensinadas em cada nível de tutorial. Algumas mecânicas relacionadas foram agrupadas no mesmo nível.*

Um nível de tutorial não é simplesmente um nível fácil, ou curto. Ele deve conter uma experiência roteirizada, que explica como realizar as ações disponíveis ao jogador, quais são os principais desafios do jogo e como utilizar a interface [Ada10].

Diversas diretrizes de *level design* foram seguidas para desenvolver o tutorial:

- "Ensine gradualmente, por meio da experiência"[Ber16].

  Cada mecânica foi ensinada de maneira gradual: uma de cada vez. O jogador aprendia por meio da própria experiência de jogo, utilizando as mecânicas na prática, como mostra a Figura 4.37.

Figura 4.37: *Mecânicas ensinadas gradualmente: se mover, pular, pular alto e pulo duplo.*

- Pouco texto

  Existem várias maneiras de comunicar as orientações ao jogador, como a utilização de locução ou de um personagem dentro do jogo que conversa com o jogador. No projeto, foram utilizados imagens e textos impressos no fundo dos níveis, com foco em concisão, seguindo a boa prática de não interromper o fluxo do jogo com textos muito grandes ou *pop-ups* [Cre12] [Ada10]. A colagem da Figura 4.38 exibe exemplos dessa concisão.

Figura 4.38: *Exemplos de mensagens concisas de tutorial, explicando como realizar ações utilizando tanto o teclado quanto controles.*

- Não sobrecarregar o jogador

  No início do jogo, foram omitidos os elementos de interface como o painel de habilidades e o menu de seleção de mecânicas, pois não iriam ser utilizados imediatamente.

  Somente no momento em que o jogador adquire a primeira mecânica que o painel de habilidades aparece, como pode ser visto na Figura 4.39. Esse artifício tem a função de chamar a atenção do jogador para o novo elemento. Após os níveis de tutorial, o mesmo ocorrerá com o menu de seleção de mecânicas.

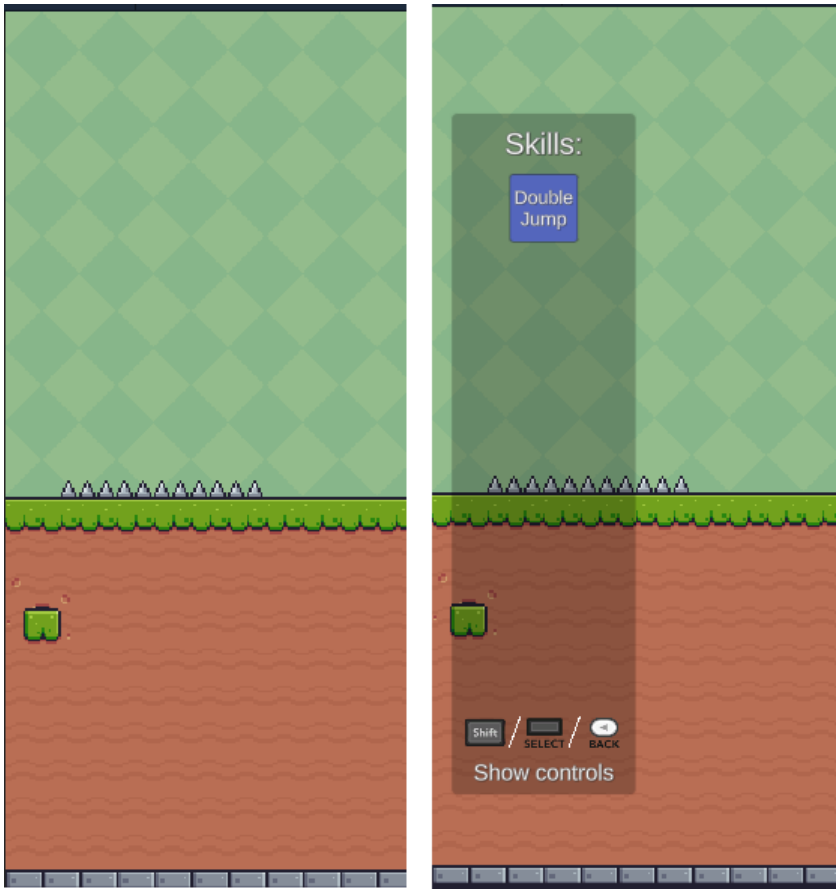

Figura 4.39: *Extremidade esquerda da tela antes e de depois do pulo duplo ser adquirido, evidenciando o aparecimento do painel de habilidades.*

- Manter engajamento do jogador

  Muitos jogos apresentam seus tutoriais de maneira enfadonha, por meio de textos gigantescos ou restringindo demais a liberdade do jogador. Uma boa prática de *design* de tutoriais é fazer com que o usuário nem perceba que está aprendendo, mantendo-o livre e engajado com um tutorial interativo, contendo desafios interessantes [Cre12] [Ber16].

- Princípio do Isolamento

  Toda mecânica que é introduzida ao jogador segue o princípio do isolamento, no

qual o personagem é posto em um ambiente isolado, seguro e controlado, sem punições [Gam14], como ilustrado na Figura 4.40.

Esse ambiente é projetado de maneira a exigir que, para prosseguir, o jogador tenha que utilizar corretamente a mecânica introduzida. Assim, se o jogador passar para a próxima seção, o *designer* tem a garantia de que a mecânica foi aprendida.

Figura 4.40: *Introdução da mecânica de pulo duplo. O jogador a adquire encostando no objeto quadrado e imediatamente precisa utilizá-la para subir no terreno elevado. Note que o ambiente é seguro: se o jogador errar o pulo, ele não é punido.*

Logo em seguida, a segurança é removida e um desafio leve é apresentado, exigindo novamente a utilização da mecânica introduzida, mas num contexto mais desafiador, conforme exibido na Figura 4.41.

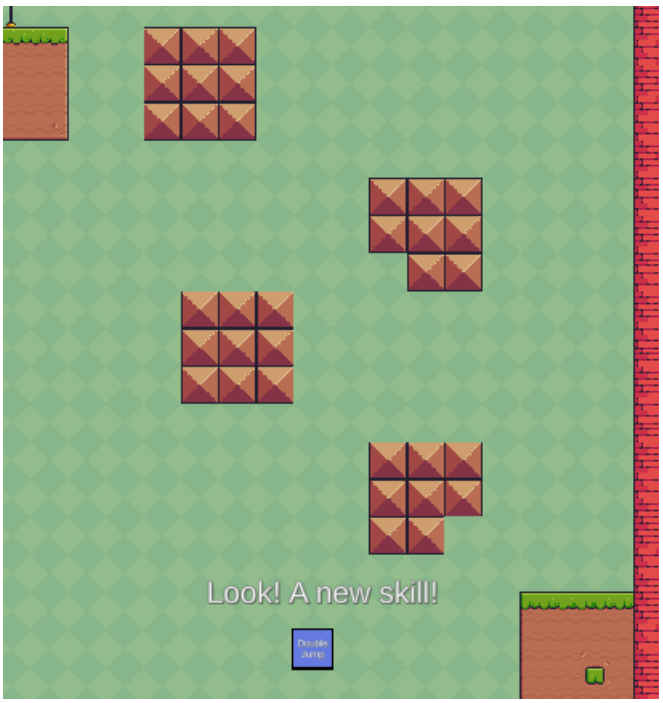

Figura 4.41: *Desafio aumenta: o jogador precisa utilizar o pulo duplo para alcançar 4 terrenos elevados seguidos. A segurança é removida: se errar o pulo, cairá e perderá progresso.*

Por último, é proposto um desafio mais difícil, geralmente envolvendo a interação com outras mecânicas e exigindo um maior domínio da habilidade aprendida, do modo como é demonstrado na Figura 4.42.

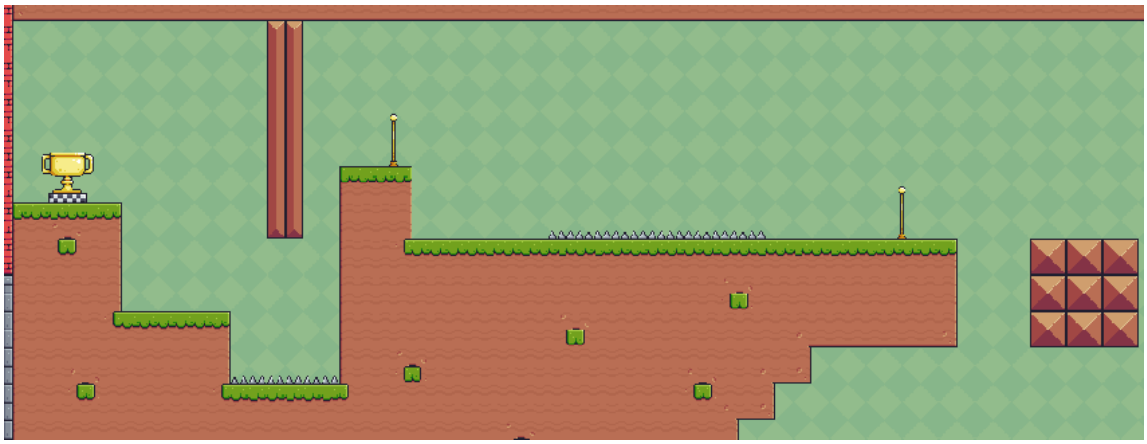

Figura 4.42: *Desafio final: agora, há interação com obstáculos (espinhos) e, no segmento final, uma técnica mais apurada é exigida, na qual o jogador deve utilizar o pulo duplo somente quando estiver bem perto de enconstar nos espinhos.*

4.6.2 Níveis comuns

Após o tutorial, o jogador é apresentado aos níveis em que o menu de escolha de mecânicas é liberado. Foram desenvolvidos 2 níveis comuns, bem mais longos e desafiadores do que os de tutorial.

Em um jogo de plataforma convencional, os níveis são projetados com o conhecimento de que o personagem principal tem acesso a um conjunto fixo de mecânicas. Em *Mechanical Playground*, o jogador pode escolher conjuntos muito diferentes de mecânicas, tornando o processo de *level design* bastante desafiador.

Para que esse processo fosse factível dentro do prazo estabelecido, o *design* foi feito pensando em 4 arquétipos de personagens, cada um contendo um conjunto fixo de mecânicas:

- *Ninja*: personagem ágil. Possui as 6 mecânicas de movimentação: *Double Jump*, *Wall Slide*, *Wall Jump*, *Dash*, *Ethereal Dash* e *Blink*.
- *Rambo*: personagem que destrói tudo em sua frente. Possui as 6 mecânicas de ataque: *Attack*, *Range Boost*, *Destroy Projectile*, *Pogo Jump*, *Gun Boots* e *Explosion*.
- *Tanker*: personagem defensivo. Possui as 6 mecânicas de utilidade: *Shield*, *Parry*, *Reflect Projectile*, *Spike Invulnerability*, *Saw Invulnerability* e *Create Platform*.
- Generalista: personagem balanceado. Possui mecânicas das 3 categorias mas não tem acesso a nenhuma mecânica especial (*Blink*, *Explosion* e *Create Platform*).

Com estes arquétipos em mente, foram estipuladas as seguintes diretrizes para o desenvolvimento dos níveis comuns:

1. Todos os arquétipos devem conseguir superar os obstáculos, cada um da sua maneira.

2. Algumas seções serão mais fáceis para um arquétipo do que para outro, porém, será almejado um equilíbrio de dificuldade.

Sempre que uma seção de um nível era projetada, pensava-se nas mecânicas necessárias para cada arquétipo superar os obstáculos. Se algum dos arquétipos não conseguisse passar da seção, ela era alterada. Nenhuma seção poderia exigir as 3 mecânicas especiais, pois isso excluiria o arquétipo generalista. Um rascunho demonstrando esse processo está apresentado na Figura 4.43.

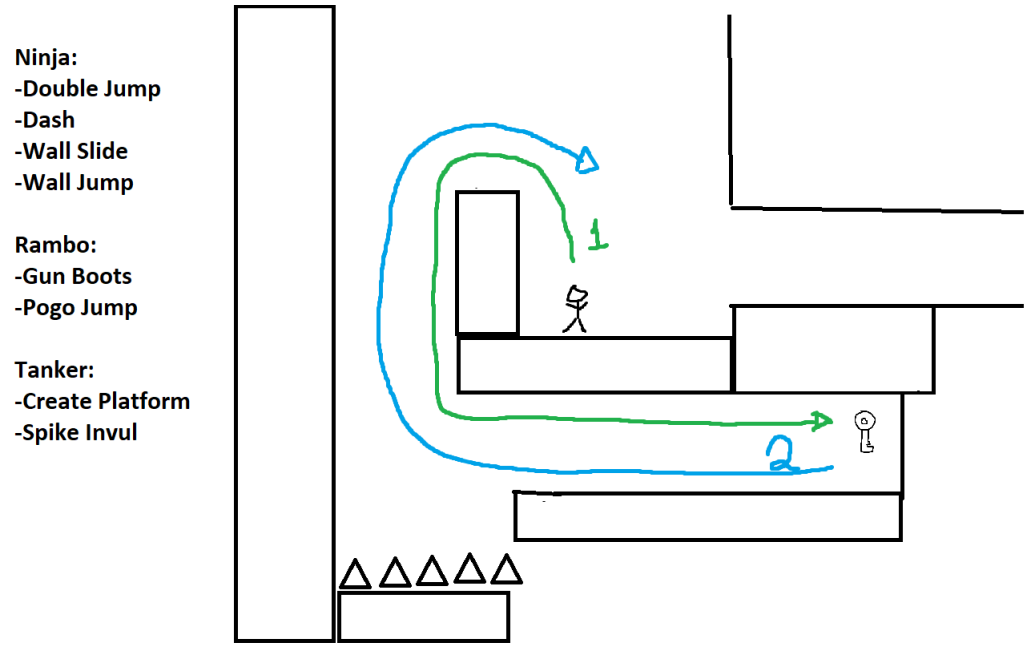

Figura 4.43: *Rascunho feito durante o design de uma seção em que o personagem tem que passar por um terreno alto, evitar os espinhos e voltar. As mecânicas exigidas para cada arquétipo estão listadas à esquerda.*

Quando era verificado que uma seção beneficiava um dos arquétipos, outra seção era alterada para prejudicá-lo, visando o balanceamento. Na seção exibida na Figura 4.44, por exemplo, há uma área na qual o arquétipo *Ninja* consegue evitar diversos obstáculos utilizando o *Blink*. Para equilibrar, no mesmo nível há uma área que impossibilita o uso do *Blink*, como pode ser visto na Figura 4.45.

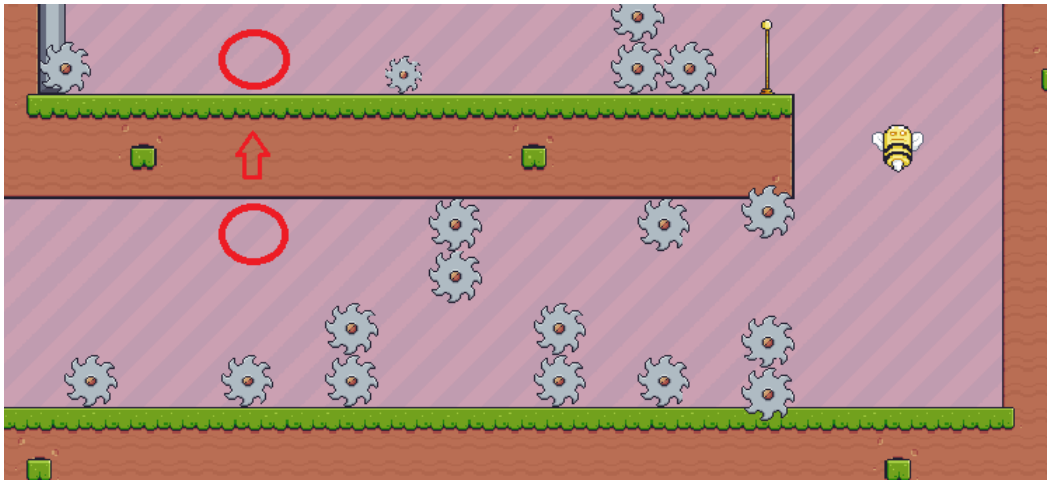

Figura 4.44: *Seção que beneficia a escolha das mecânicas de movimento, ao permitir que vários obstáculos sejam evitados com o uso do Blink (Teleporte).*

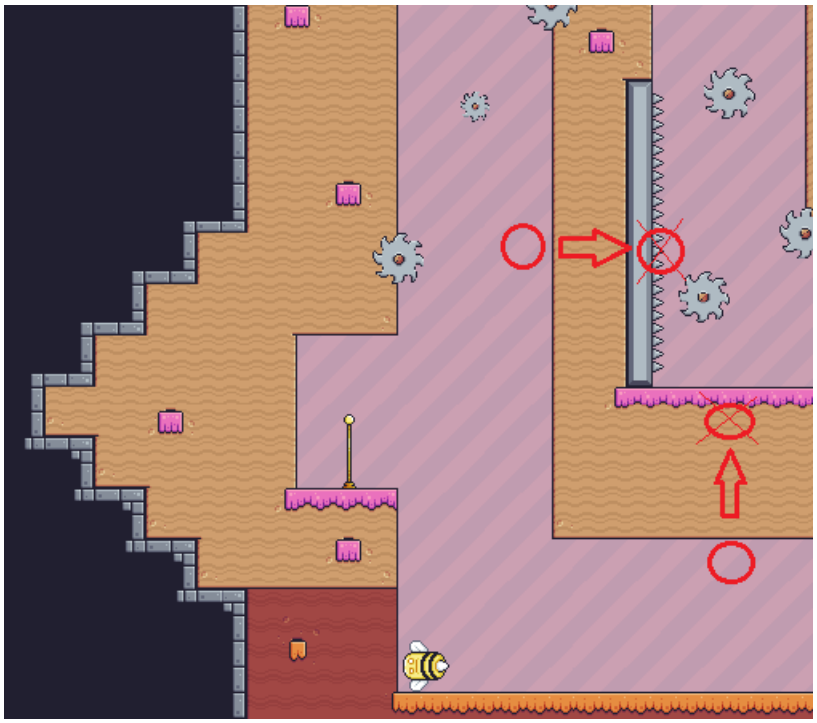

Figura 4.45: *Seção que prejudica a escolha das mecânicas de movimento, ao impedir a utilização do Blink (Teleporte).*

Capítulo 5

Desenvolvimento

O desenvolvimento de um jogo consiste na implementação das decisões que foram feitas na etapa de design. No caso de um jogo digital, a concretização do projeto se dá pela programação de um software.

A maioria dos jogos catalogados neste projeto foram feitos nas décadas de 80 e 90, principalmente para as plataformas *NES* e *SNES* e, nessa época, os desenvolvedores tinham que programar todos os sistemas do jogo, incluindo componentes complexos como a física, a síntese de imagens e o recebimento de entradas. Além disso, era utilizada a linguagem de programação *Assembly* [Wik20a] [Wik20b], conhecida por não ser muito acessível.

Nos últimos anos, a área de desenvolvimento de jogos foi democratizada com o advento de ferramentas chamadas de motores de jogo (*game engines*) de acesso gratuito, que providenciam uma implementação de diversos sistemas fundamentais, permitindo ao desenvolvedor focar na lógica do jogo em si [Wik21c].

Neste capítulo, será apresentado o motor de jogo e as demais ferramentas utilizadas no projeto, assim como um detalhamento da arquitetura do código e dos princípios seguidos na criação do jogo.

5.1 Ferramentas

A ferramenta mais utilizada no projeto foi o motor de jogo *Unity*, que utiliza a linguagem de programação *C#*. Este motor é um dos mais populares atualmente [Cru18] e foi utilizado no desenvolvimento de diversos jogos de sucesso, como *Hollow Knight* e *Cuphead* [Wik21d], que inclusive já foram citados nos capítulos anteriores e serviram como forte inspiração para este trabalho. Sua interface pode ser observada na Figura 5.1.

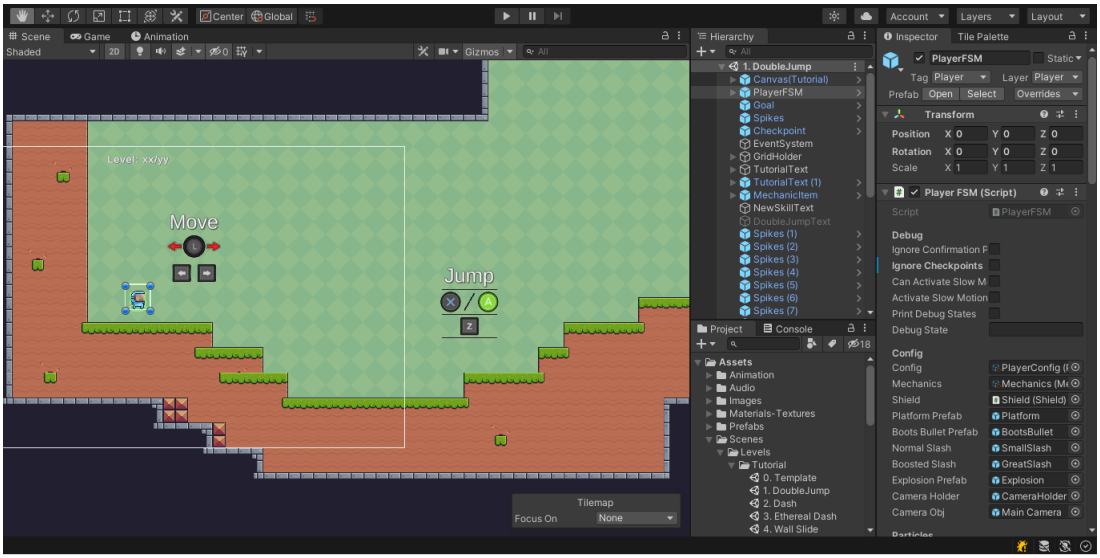

Figura 5.1: *Interface gráfica da Unity.*

As demais ferramentas utilizadas foram: *Visual Studio Code* para a escrita do código; *Git*, *GitHub* e *GitKraken* para o versionamento do código; *MSPaint* no *design* dos protótipos de níveis e de interface; *Trello* para a organização de tarefas; *GIMP* e *Audacity* para ajustar imagens e áudios, respectivamente; *Google Docs*, *Google Meet*, *Discord* e *e-mail* para a comunicação com o orientador e a plataforma *Itch.io*, para o lançamento do jogo.

5.2 Repositório

No estado final do projeto, encontram-se 72 arquivos referentes à lógica de jogo, contendo cerca de 3500 linhas de código, além de 8 arquivos, contendo 600 linhas, relativos aos testes de unidade e integração. Vale notar que esses números não incluem linhas em branco nem linhas de comentário.

Este resultado foi obtido por meio de 548 *commits*, que podem ser acessados no repositório de código.

5.3 Mecânicas

5.3.1 Abordagem direta

O início do processo de programação das mecânicas foi marcado por uma abordagem direta, que pode ser considerada ingênua: não houve a escolha de uma arquitetura de código bem estruturada. A ideia era desenvolver as mecânicas de maneira simples e direta.

Essa abordagem funcionou muito bem enquanto haviam poucas mecânicas. O primeiro protótipo foi desenvolvido rapidamente e era composto apenas de plataformas e do personagem principal, que podia andar, pular e executar o pulo duplo.

Em seguida, foram feitas mais 4 mecânicas: *dash*, *wall slide*, *wall jump* e *attack*.

A cada mecânica nova, a interação entre as classes aumentava, tornando o desenvolvimento cada vez mais difícil. Na criação da mecânica de pulo, por exemplo, é necessário identificar se o personagem está no chão, então cria-se a variável *isGrounded* para verificação na classe *Jump*. Ao desenvolver o *dash* (impulso), é necessário impedir o personagem de andar e pular enquanto está sendo impulsionado, logo, cria-se a variável *isDashing* para verificação na classe *Walk* e na classe *Jump*. O personagem só pode executar o pulo na parede se estiver deslizando nela, então cria-se o *isWallSliding* para verificação na classe *Wall Jump*, etc. Isso evidencia um alto nível de acoplamento entre as mecânicas, o que acabava gerando as seguintes dificuldades:

- Legibilidade: durante a leitura de um arquivo, encontravam-se diversas checagens de variáveis booleanas, frequentemente pertencentes à outras mecânicas, o que prejudicava o entendimento do código.
- *Debugging*: quando um *bug* ocorria no jogo, era difícil detectar exatamente onde no código poderia estar o problema, já que haviam segmentos de mecânicas em vários lugares.
- Extensibilidade: ao adicionar uma nova mecânica, era necessário refatorar o código de todas as outras, de maneira a se adequarem à essa nova classe.

Na Listagem 5.1 a seguir, observam-se dois segmentos de código evidenciando o acoplamento entre as mecânicas *walk*, *dash*, *wall slide* e *wall jump*:

```
// Segmento da clase Walk
void ProcessWalkRequest() {
    float xInput = Input.GetAxisRaw("Horizontal");
    int direction = CalculateDirection(xInput);
    SetWalkAnimation(direction);

    if (player.disableControls) return;
    if (player.isDashing) return; // Dash
    if (player.isWallJumping) return; // WallJump
    if (player.isGluedOnTheWall) return; // WallSlide

    WalkAction(direction);
}

// Segmento da classe Dash
void Update() {
    if (player.isGrounded || player.isWallSliding) {
        canDash = true;
    }
}
```

Listagem 5.1: *Segmentos de código mostrando alto nível de acoplamento.*

Diante deste problema, foi buscada uma alternativa para melhorar a estrutura do código.

Identificou-se que utilizar o padrão de *design State*[1] para a criação de uma máquina de estados solucionaria todos os problemas previamente citados.

5.3.2 Máquina de estados

Uma máquina de estados finita (*finite state machine*, ou, *FSM*) é uma máquina abstrata que contém um número finito de estados e, em um dado momento, ela pode estar apenas em exatamente um estado [Nys14].

A FSM é definida por uma lista de estados possíveis, as condições de transição entre os estados e o estado inicial, como ilustra a Figura 5.2.

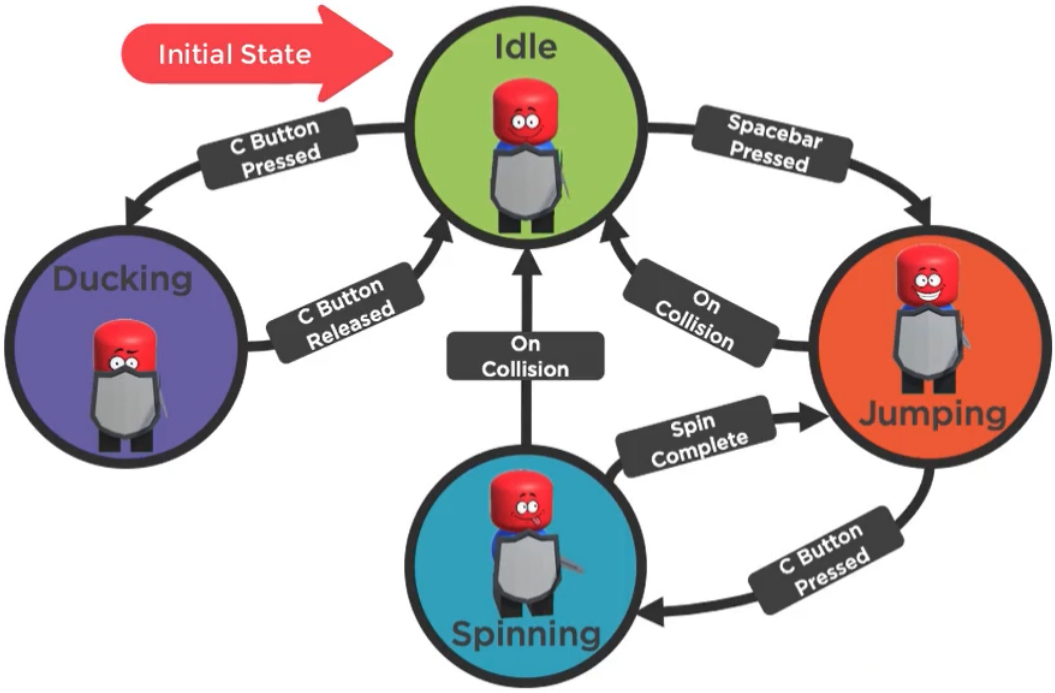

Figura 5.2: *Ilustração de uma FSM contendo 4 estados, suas transições e o estado inicial [Uni19].*

Sua implementação se dá por meio de 3 elementos:

- Contexto: é a classe que representa a máquina. Ela gerencia os estados, mantendo sempre apenas um deles ativo, como pode se observar na Listagem 5.2.

[1] https://gameprogrammingpatterns.com/state.html

```
public class PlayerFSM : MonoBehaviour {
    private PlayerBaseState currentState;

    // Lista de estados possíveis
    public readonly GroundedState Grounded = new GroundedState();
    public readonly JumpingState Jumping = new JumpingState();
    public readonly FallingState Falling = new FallingState();

    // Estado inicial: Grounded
    private void Start() {
        TransitionToState(Grounded);
    }

    private void Update() {
        currentState.Update(this);
    }

    public void TransitionToState(PlayerBaseState state) {
        currentState = state;
        currentState.EnterState(this);
    }
}
```

Listagem 5.2: *Classe que gerencia a máquina de estados do personagem principal.*

- Estado abstrato: define um modelo que encapsula comportamentos comuns de todos os estados concretos. Como mostra a Listagem 5.3, é nessa classe base que ficam os métodos que checam se é necessário fazer a transição entre estados.

```
public abstract class PlayerBaseState {
    public abstract void EnterState(PlayerFSM player);
    public abstract void Update(PlayerFSM player);

    // Condições de transição
    public virtual bool CheckTransitionToGrounded(PlayerFSM
    ↪ player) {
        if (player.isGrounded) {
            player.TransitionToState(player.Grounded);
            return true;
        }
        return false;
    }
}
```

Listagem 5.3: *Classe abstrata que estabelece o modelo para a criação dos estados concretos.*

- Estado concreto: classe que herda da classe abstrata e representa um estado concreto, implementando os comportamentos específicos de cada ação. Um segmento de

código resumido de um dos estados concretos está apresentado na Listagem 5.4.

```
public class PlayerDoubleJumpingState : PlayerBaseState {
    public override void EnterState(PlayerFSM player) {
        player.animator.Play("PlayerDoubleJump");
        player.canDoubleJump = false;
        DoubleJumpAction(player);
    }

    public override void Update(PlayerFSM player) {
        base.ProcessMovementInput(player);

        if (base.CheckTransitionToFalling(player)) return;
        if (base.CheckTransitionToDashing(player)) return;
        if (base.CheckTransitionToAttacking(player)) return;
        if (base.CheckTransitionToBlinking(player)) return;
        if (base.CheckTransitionToWallSliding(player)) return;
    }

    void DoubleJumpAction(PlayerFSM player) {
        player.rb.velocity = new Vector2(player.rb.velocity.x,
        ↪  player.config.doubleJumpForce);
    }
}
```

Listagem 5.4: *Classe que representa o estado concreto em que o personagem está realizando o pulo duplo.*

O personagem começa o jogo no estado *Grounded*, o que significa que ele está parado no chão. Como este é o estado atual, sua função *Update* será executada. É na função *Update* dos estados concretos que são executadas as checagens de transição. Como no estado *Grounded* há a checagem de transição para o estado *Jumping*, se o jogador pressionar o botão de pular, a transição ocorrerá com a chamada da função *TransitionToState*, que altera o estado atual da máquina e imediatamente chama a função *EnterState* do novo estado. No caso do *Jumping*, é nesta função de entrada que a ação de pular é executada. Novamente, as checagens de transição serão executadas a todo *frame* e assim por diante.

A classe abstrata *PlayerBaseState* possui métodos gerais de checagem de transição pois eles serão utilizados nas diversas classes concretas que herdam dela. Além disso, esses métodos são virtuais, possibilitando que algum estado concreto possa reescrevê-los, se o comportamento de transição for diferente do geral. Por exemplo, a transição para o estado *Wall Sliding* acontece, em geral, quando o personagem está sendo ativamente pressionado contra uma parede, ou seja, o jogador está apertando o botão direcional no sentido da parede. Porém, foi projetado que quando o personagem está sendo impulsionado pelo *Dash* e encosta em uma parede, ele deve ir para o estado *Wall Sliding*. Como, nesse caso, o personagem estará sendo impulsionado automaticamente pelo funcionamento do *Dash*, o jogador não estará apertando o botão direcional contra a parede, então a função de transição geral não funcionará. Portanto, é necessário que, na classe do *Dash*, haja a reescrita dessa

função de transição, contendo uma simples verificação de se o personagem está tocando em uma parede. Na Figura 5.3 são apresentadas todas as mecânicas da máquina de estados do jogador.

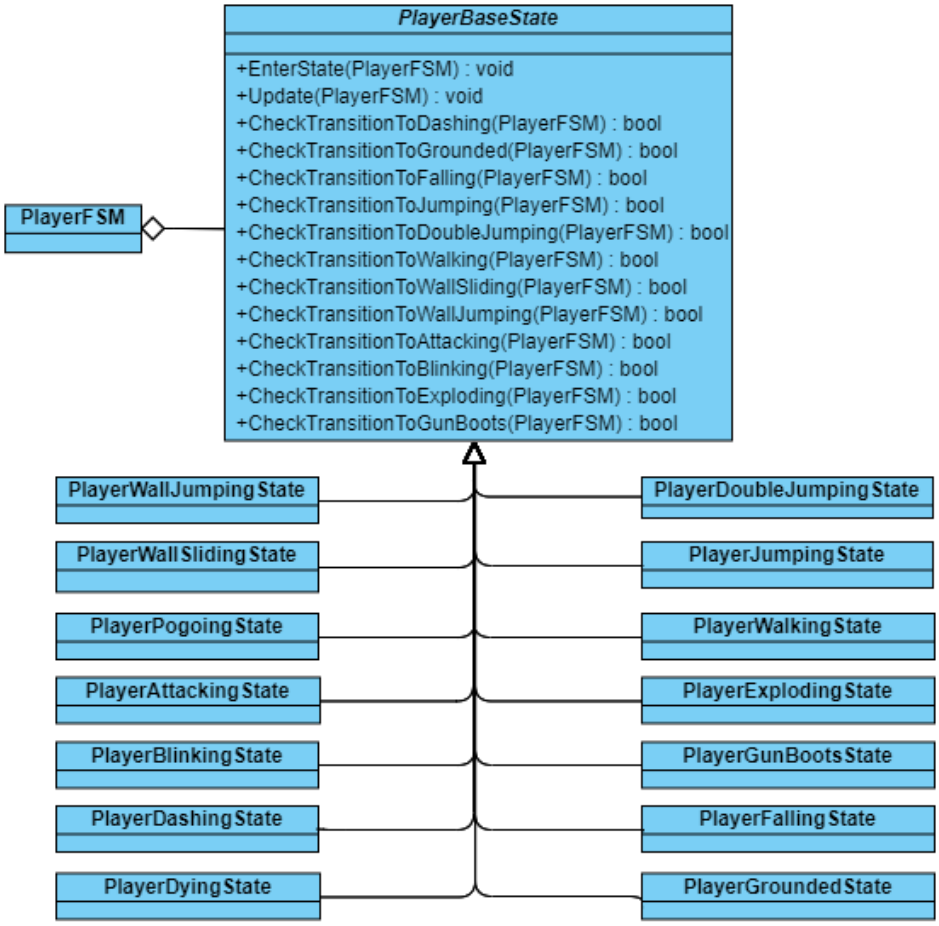

Figura 5.3: *Diagrama UML contendo todas as mecânicas desenvolvidas dentro da estrutura de máquina de estados do jogador. Apenas as operações da classe base estão apresentadas no diagrama.*

Com essa nova estrutura de código, aqueles problemas apresentados pela abordagem direta foram solucionados:

- Legibilidade: com um código mais modular, a leitura ficou muito mais fácil. Sem as checagens de booleanos vindos de outras classes, o código de cada mecânica ficou autocontido.
- *Debugging*: agora, quando um *bug* ocorre no jogo, pode-se identificar exatamente por quais estados o jogador passou, sendo muito mais fácil localizar em qual lugar do código está o problema.
- Extensibilidade: quando uma mecânica nova é adicionada, pouca coisa muda na estrutura geral. Se um estado existente pode transicionar para o novo estado, uma única linha de código será inserida em sua classe, que será a chamada do método de

checagem de transição. Se um estado existente não permitir transição para o estado novo, seu código simplesmente permanecerá o mesmo.

Somente por meio desta abordagem o projeto foi possível. Diversos problemas que tomavam muito tempo de desenvolvimento na abordagem direta se tornaram extremamente mais fáceis de serem administrados, como restringir o jogador de poder andar durante um *dash* ou garantir que o personagem só poderá pular da parede se estiver deslizando nela. Antes, era necessário uma grande quantidade de checagens de booleanos, agora, se um estado não permitir transição para outro, essa transição simplesmente não ocorrerá. Por exemplo, se apenas o estado *wall slide* tiver transição para o estado *wall jump*, já está garantido que a ação de pular da parede só ocorrerá nesse contexto: não é necessário adicionar checagens em outras classes. Se não há transição do estado *dash* para o estado *walk*, quaisquer tentativas de fazer o personagem andar durante um *dash* serão automaticamente ignoradas pelo código.

Esta nova abordagem também acabou solucionando um problema relacionado às animações. A *Unity* possui, internamente, uma máquina de estados para lidar com animações, como pode ser observado na Figura 5.4. Na abordagem direta, essa máquina refletia os problemas do código: grande confusão, muitas variáveis de controle e *bugs* difíceis de serem identificados:

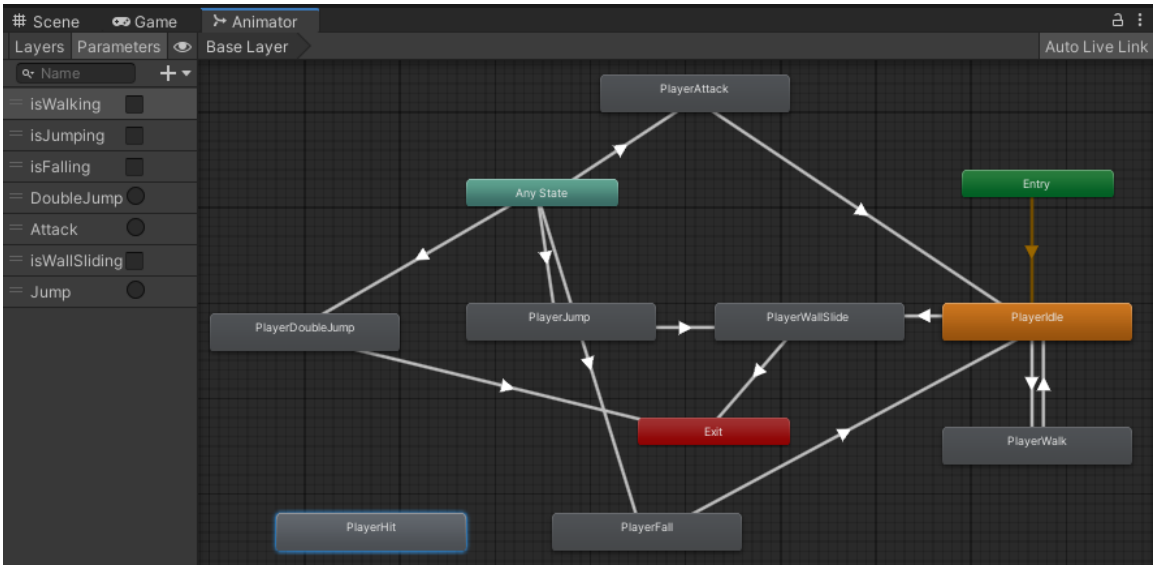

Figura 5.4: *Máquina de estados interna da Unity referente às animações do personagem principal, antes da reestruturação do código. Nota-se um alto nível de complexidade e desorganização.*

Com a reestruturação do projeto, o gerenciamento das animações pôde ser feito pela própria máquina de estados implementada no código: quando o personagem entra no estado de pulo, a animação de pulo é tocada; quando ele entra no estado do pulo duplo, a animação de pulo duplo é tocada, e assim por diante. Além dessa forma ser extremamente mais simples, ela também resolveu os *bugs* que haviam anteriormente.

5.3.3 Mecânicas fora da máquina de estado

Nem todas as mecânicas foram implementadas dentro da estrutura da máquina de estados. Existem mecânicas projetadas para que o jogador possa ativá-las a qualquer momento, em conjunto com qualquer outra mecânica, como o escudo e a criação de plataformas. Portanto, elas foram implementadas de maneira isolada, em classes separadas da máquina. O acesso à essas mecânicas é feito por meio de referências na classe do *player*.

Além disso, nem toda mecânica precisou de uma classe própria. A habilidade *Ethereal Dash*, por exemplo, pôde ser implementada como uma simples propriedade da classe *Dash*. O mesmo ocorreu com o *Range Boost*, que se integrou à classe *Attack*. As mecânicas de invulnerabilidade à espinhos e serras foram implementadas utilizando as classes dos próprios obstáculos: quando um nível é carregado, os obstáculos são criados e eles checam se a mecânica de invulnerabilidade está ativada no jogador; se estiver, o obstáculo se torna não-letal.

Os inimigos *Bee*, *Trunk* e *Spiky* também foram implementados utilizando máquinas de estados. Como todos se comportam da mesma maneira em relação à sofrerem ataques e serem eliminados, há uma classe superior na hierarquia chamada *Enemy*:

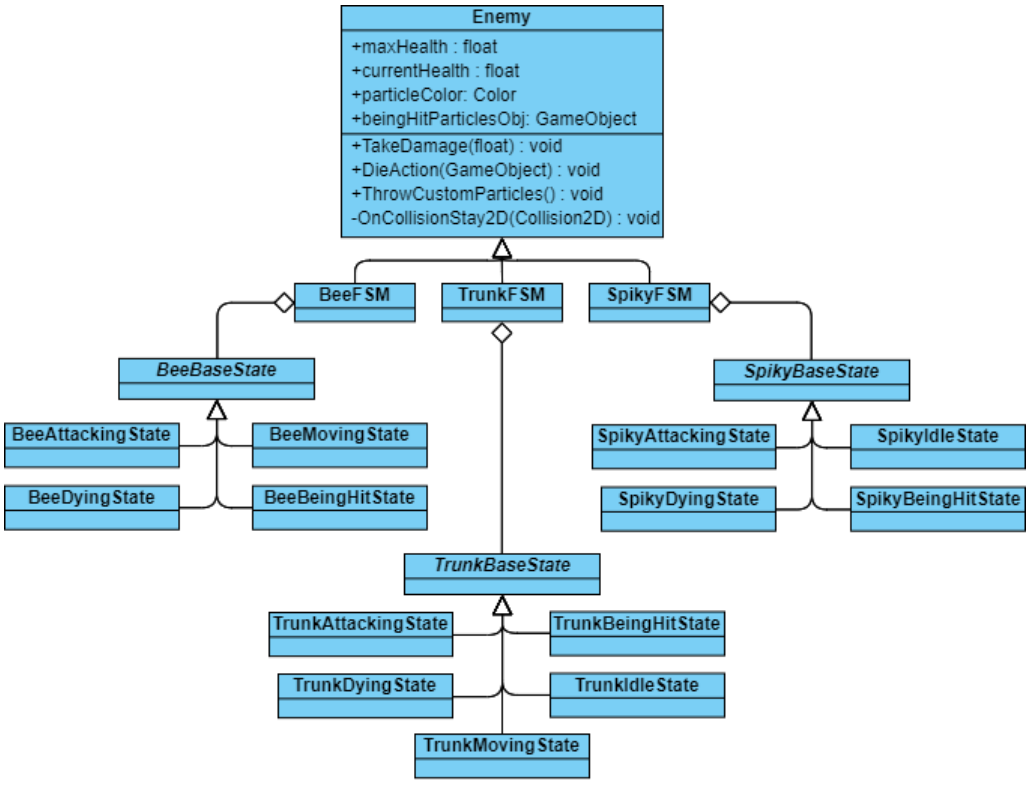

Figura 5.5: *Diagrama UML demonstrando as associações entre as classes de inimigos, dentro da estrutura de máquina de estados. Apenas as operações da classe Enemy estão apresentadas.*

5.4 Game Feel

Segundo *Steve Swink*, autor do livro *Game Feel: A game designer's guide to virtual sensation*, o conceito de *game feel* é definido como "O controle em tempo real, de objetos virtuais, em um espaço simulado, com interações enfatizadas por polimento"[Swi09].

Na indústria dos videogames, muitas funcionalidades relacionadas à melhorar o *game feel* são criadas a partir da observação de padrões de comportamento entre os jogadores. O jogo *Celeste*, que foi catalogado e serviu de inspiração para este trabalho, implementa vários destes artifícios e a sua criadora os demonstrou recentemente [Tho20].

Como este projeto foca na possibilidade do jogador se expressar, diversas funcionalidades foram desenvolvidas com o propósito de melhorar a experiência e o *game feel* do jogador. Algumas se dedicaram a aprimorar o controle dos objetos virtuais, enquanto outras focaram no polimento visual. Nenhum destes elementos desenvolvidos é estritamente necessário para o jogo funcionar, porém, sua adição contribui para a experiência do jogador, muitas vezes sem ele perceber.

A seguir estão descritas algumas dessas funcionalidades:

- *Coyote Timer*: foi identificado que os jogadores costumam pular apenas quando estão na borda das plataformas. Como o personagem se movimenta muito rapidamente, às vezes ele já saiu da plataforma, mas o jogador não percebe, aperta o botão de pulo e nada acontece, deixando-o frustrado. O *Coyote Timer*, nomeado a partir do coiote do desenho animado "Papa-Léguas", permite que o personagem pule mesmo depois de ele ter saído do chão (apenas por alguns milissegundos) diminuindo a frustração do jogador.

- *Bunny Hop*: outro hábito que os jogadores têm é de ficarem apertando continuamente o botão de pulo, para se moverem "saltitando". Porém, muitas vezes o botão é apertado alguns milissegundos antes do personagem encostar no chão, fazendo com que o personagem não pule da maneira como o jogador imaginava. Para solucionar este problema, foi feito um *Input Buffering*, detectando o acionamento do botão enquanto o personagem está no ar e, se ele encostar no chão depois de alguns milissegundos, o personagem pula instantaneamente.

- Gravidade na queda: este artifício, que vêm sendo implementado desde os primeiros jogos da série *Mario*, faz com que o personagem seja submetido a uma gravidade maior depois de atingir o pico do pulo ou depois que o jogador solta o botão de pulo. Isto resulta em uma movimentação mais dinâmica; sem este artifício, o pulo fica com uma sensação "flutuante".

- *Screen Shake*: quando o jogador utiliza a mecânica de explosão, ou quando ele morre, ocorre um leve estremecimento da tela. Isso amplifica o impacto dessas ações, aumentando a sensação de imersão do jogador no ambiente do jogo.

- Partículas: ao atacar um inimigo, são emitidas diversas partículas das cores dos próprios inimigos, realçando o impacto da ação e servindo como um *feedback* ao jogador de que o seu ataque foi bem-sucedido.

5.5 Testes

Neste projeto, foram desenvolvidos testes unitários e de integração. A *Unity* possui uma ferramenta de testes bastante robusta, que permite a execução dos testes de integração simulando os cenários de teste como se estivessem sendo jogados em tempo real. Por esta razão, foi dado um foco muito maior aos testes de integração, os quais foram utilizados para simular a utilização das mecânicas.

Durante o processo de reestruturação do código para adaptá-lo à máquina de estados, os testes foram essenciais para garantir que as mecânicas continuariam funcionando depois da refatoração. A lista de testes executados na interface da Unity pode ser vista na Figura 5.6.

Figura 5.6: *Testes de integração, garantindo o bom funcionamento das mecânicas.*

Capítulo 6

Resultados

Após a conclusão do desenvolvimento, o jogo *Mechanical Playground* foi publicado na plataforma *Itch.io*.

A *Unity* permite a exportação do jogo para diversas plataformas, como *Windows*, *Linux* e uma versão *Web* que pode ser jogada no próprio navegador, sem precisar fazer o *download* de arquivos. Foi feito um esforço extra para que esta versão *Web* fosse desenvolvida, pois isso significaria um acesso bem mais facilitado para o usuário. As três versões foram disponibilizadas no site, além de um vídeo de demonstração do jogo. Algumas estatísticas providenciadas pela plataforma estão expostas na Figura 6.1.

Figura 6.1: *Estatísticas da plataforma Itch, mostrando que o jogo foi jogado mais de 350 vezes.*

Junto com o lançamento, foi elaborado um formulário contendo o *link* para o jogo e perguntas à respeito da experiência de jogá-lo. A intenção foi permitir que jogadores reais pudessem avaliar a qualidade do produto final e se o objetivo do projeto foi bem-sucedido. Normalmente, os membros do *USPGameDev* testam os seus jogos com usuários reais nos eventos presenciais promovidos pelo grupo, nos quais é possível entender com mais detalhes a interação do jogador com o produto. Porém, devido ao distanciamento social vivenciado atualmente, esta opção não foi viável.

O formulário foi compartilhado no grupo do *USPGameDev* e em outros grupos contendo entusiastas de jogos, que são o público-alvo. A pesquisa obteve 19 respostas anônimas e o seu resultado pode ser acessado por meio deste link.

Sobre a qualidade, 90% julgaram o jogo divertido ou muito divertido e, apesar de haverem críticas, a maioria dos entrevistados achou a dificuldade justa e os controles intuitivos.

O objetivo do projeto era permitir que o jogador pudesse se expressar por meio da seleção de mecânicas, que foram chamadas de "habilidades" dentro do contexto do jogo.

Dos 19 respondentes, 16 conseguiram passar pelos níveis de tutorial e chegar na parte em que podem escolher as mecânicas. **Todos** os 16 avaliaram que conseguiram expressar sua maneira favorita de jogar, como observa-se no gráfico da Figura 6.2.

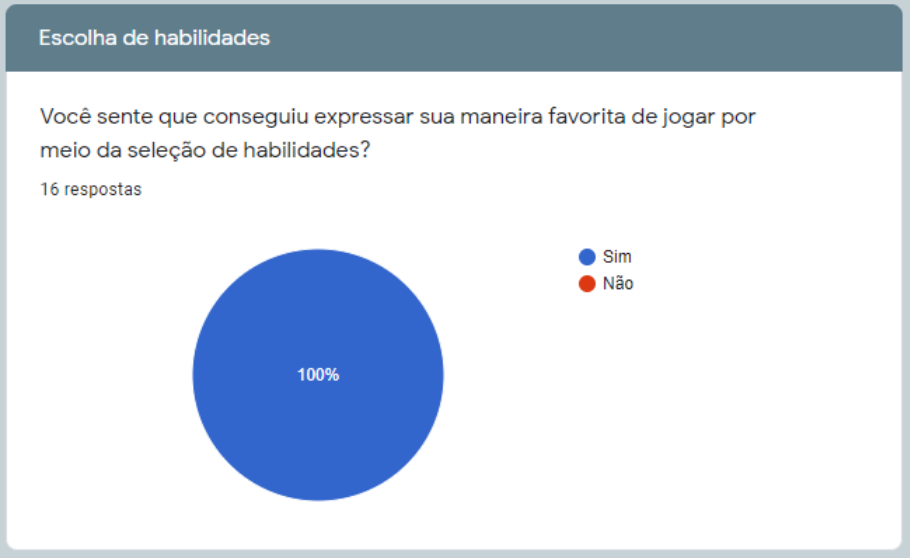

Figura 6.2: *Avaliação extremamente positiva da capacidade de expressão do jogador por meio da seleção de mecânicas.*

Houve uma distribuição razoável entre os grupos de mecânicas selecionados, observando-se uma maior adesão das mecânicas de utilidade e uma baixa preferência pelas mecânicas de combate, como retrata a Figura 6.3.

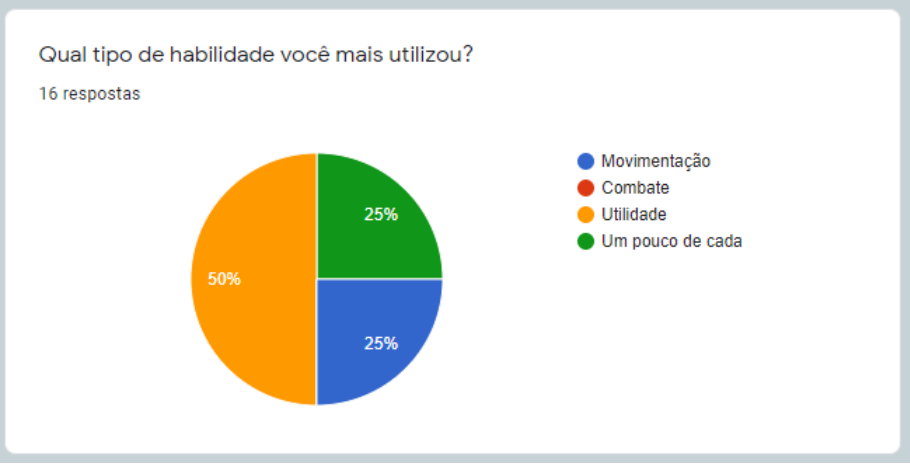

Figura 6.3: *Distribuição da escolha dos grupos de mecânicas. As mecânicas de utilidade foram as mais escolhidas pelos jogadores.*

Foi observado que a mecânica de *pogo-jump* se mostrou frustrante para pelo menos um terço dos jogadores. Além disso, houve comentários negativos em relação à *bugs* em que os controles não funcionavam corretamente, e também sobre a restrição de só poder

utilizar uma mecânica de cada vez nos níveis de tutorial: alguns jogadores gostariam de poder manter as mecânicas adquiridas nas fases anteriores.

O momento favorito da experiência de 40% dos jogadores envolveu a possibilidade de escolher as mecânicas, enquanto que 30% destacaram elementos de *level design* como seus aspectos prediletos do jogo.

Além das informações adquiridas por meio das respostas do formulário, houve mais uma fonte de *feedback*. A própria plataforma *Itch.io* destaca automaticamente, em sua página inicial, os jogos recém-lançados. Por causa disso, apesar do jogo só ter sido ativamente compartilhado por meio do formulário, centenas de usuários nativos da plataforma acessaram o jogo espontaneamente, e alguns ainda deixaram comentários muito positivos, como os da Figura 6.4.

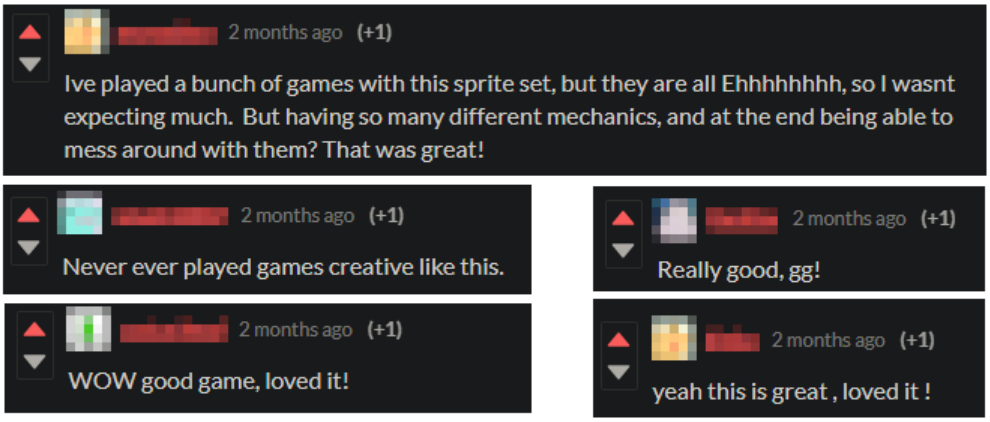

Figura 6.4: *Comentários de usuários espontâneos do Itch.io sobre o jogo Mechanical Playground.*

Capítulo 7

Conclusões

O processo de desenvolvimento do jogo *Mechanical Playground* revelou ser uma forma muito eficiente de aplicar os conhecimentos adquiridos durante a graduação e, ao mesmo tempo, também representou uma ótima oportunidade de aprendizagem na prática. O início da programação utilizando uma abordagem ineficiente, a descoberta de um padrão de projeto promissor e a sua posterior implementação bem-sucedida, evidenciam o enorme progresso ocorrido durante a elaboração do trabalho.

Poucos *bugs* foram relatados e a maioria das reclamações se concentrou na parte de *design* do jogo, mostrando que a parte de programação foi muito bem executada.

Ainda assim, futuramente, pretende-se trabalhar em cima dos problemas de *design* como a baixa preferência pelas mecânicas de combate, em especial o *Pogo-jump*. Assim que for viável, também será muito interessante a observação de pessoas jogando na prática, para que seja possível capturar as emoções e reações em tempo real.

Diante do expressivo número de 100% dos entrevistados satisfeitos com a capacidade de expressão por meio da escolha de mecânicas e observando todo o *feedback* positivo em relação à qualidade do jogo em si, pode-se concluir que o objetivo do projeto foi atingido com sucesso.

Considerando também as críticas e sugestões de melhorias apontadas, há bastante espaço para uma continuação do desenvolvimento e até mesmo um possível lançamento comercial no futuro.

Apêndice A

Códigos-fonte

O código completo do jogo está disponível no repositório do *GitHub*. Como o *Script* de *WebScraping*, feito na seção de catalogação para filtrar os jogos de plataforma, não fez parte do jogo em si, não está no repositório, mas pode ser observado a seguir:

```
from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
from selenium.common.exceptions import TimeoutException
import pandas as pd

def main():
  my_url =
   ↪  "https://howlongtobeat.com/user.php?n=Sorcker&s=games&custom=1"
  driver = webdriver.PhantomJS()
  driver.get(my_url)

  delay = 15
  try:
    myElem = WebDriverWait(driver,
     ↪  delay).until(EC.presence_of_element_located((By.CLASS_NAME,
     ↪  'user_game_list')))
    print("Pagina esta pronta!")

    games = driver.find_elements_by_class_name('text_teal')
    links = []
    platform_games = []
    for game in games:
      links.append(game.get_attribute("href"))

    print("Links foram copiados.")
    for link in links:
      driver.get(link)
```

```
        game_title = parse_game_title(driver.title)
        print("Analisando: " + game_title)
        infos = driver.find_elements_by_class_name('profile_info')
        for info in infos:
          if "latform" in info.text:
            print(game_title + " eh de plataforma!!")
            print("")
            platform_games.append(game_title)

      df = pd.DataFrame({'Jogos de Plataforma':platform_games})
      df.to_csv('platform_games.csv', index = False, encoding = 'utf-8')

    except TimeoutException:
      print("Tempo excedido para carregamento da pagina.")


def parse_game_title(page_title):
    split_title = page_title.split(' ')
    game_title = ""
    for c in range(3, len(split_title)):
      word = split_title[c]
      if word != '|':
        game_title += word + " "
      else:
        break

    return game_title[:-2].encode('ascii', 'ignore')


main()
```

Referências

[1UP06] 1UP. *Playing With Power: Great Ideas That Have Changed Gaming Forever*. https://web.archive.org/web/20060617150943/http://www.1up.com/do/feature?pager.offset=1&cId=3151392. Acessado em: 2021-02-19". 2006 (ver p. 13).

[AD12] Ernest Adams e Joris Dormans. *Game Mechanics: Advanced Game Design*. Berkeley, CA: New Riders Games, 2012, pp. 1, 221 (ver pp. 6, 9).

[Ada10] Ernest Adams. *Fundamentals of Game Design*. Berkeley, CA: New Riders Games, 2010, pp. 359, 376, 375 (ver pp. 9, 38, 39).

[Ber16] Nicolae Berbece. *This is a Talk About Tutorials, Press A to Skip*. https://youtu.be/VM1pV_6IE34?t=320. Acessado em: 2021-02-26". 2016 (ver pp. 38, 40).

[Cra03] Chris Crawford. *Chris Crawford on Game Design*. New Riders Games, 2003, p. 8 (ver p. 1).

[Cre12] Extra Credits. *Tutorials 101 - How to Design a Good Game Tutorial - Extra Credits*. https://youtu.be/BCPcn-Q5nKE?t=26. Acessado em: 2021-02-26". 2012 (ver pp. 39, 40).

[Cru18] Tech Crunch. *Unity CEO says half of all games are built on Unity*. https://techcrunch.com/2018/09/05/unity-ceo-says-half-of-all-games-are-built-on-unity/. Acessado em: 2021-03-04". 2018 (ver p. 45).

[EST08] Simon Egenfeldt-Nielsen, Jonas Heide Smith e Susana Pajares Tosca. *Understanding Video Games: The Essential Introduction*. New York, NY: Routledge, 2008, p. 64 (ver p. 13).

[Gam14] School of Game Design. *What the isolation principle is*. https://schoolofgamedesign.com/project/what-the-isolation-principle-is/. Acessado em: 2021-02-27". 2014 (ver p. 41).

[HLZ04] Robin Hunicke, Marc LeBlanc e Robert Zubek. "MDA: A Formal Approach to Game Design and Game Research". Em: *Game Developers Conference* (2004), p. 3 (ver p. 6).

[Nys14] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014, pp. 178, 183, 126 (ver pp. 5, 48).

[Pau17] Tyler Paulley. "The Official Rulebook for Choice in Video Games: An Examination of Choice in Modern Narrative Games". Bellarmine University, 2017, p. 9 (ver p. 2).

[Sch08] Jesse Schell. *The Art of Game Design: A Book of Lenses*. Morgan Kaufmann, 2008, pp. 34–37, xxiv (ver pp. 5, 8).

[Str20] Bjarne Stroustrup. *C++ Applications*. https://www.stroustrup.com/applications.html. Acessado em: 2021-02-17". 2020 (ver p. 5).

[Swi09] Steve Swink. *Game Feel: A game designer's guide to virtual sensation*. Burlington, MA: Morgan Kaufmann, 2009, p. 6 (ver p. 54).

[Tho20] Maddy Thorson. *A short thread on a few Celeste game-feel things*. https://twitter.com/MaddyThorson/status/1238338574220546049. Acessado em: 2021-03-08". 2020 (ver p. 54).

[Uni19] Unity. *Finite State Machines - Unity Learn*. https://learn.unity.com/tutorial/finite-state-machines. Acessado em: 2021-03-06". 2019 (ver p. 48).

[Wik20a] Wikipedia. *NES Programming*. https://en.wikibooks.org/wiki/NES_Programming. Acessado em: 2021-03-04". 2020 (ver p. 45).

[Wik20b] Wikipedia. *Super NES Programming*. https://en.wikibooks.org/wiki/Super_NES_Programming. Acessado em: 2021-03-04". 2020 (ver p. 45).

[Wik21a] Wikipedia. *2.5D*. https://en.wikipedia.org/wiki/2.5D. Acessado em: 2021-02-14". 2021 (ver p. 6).

[Wik21b] Wikipedia. *Fighting game*. https://en.wikipedia.org/wiki/Fighting_game. Acessado em: 2021-02-14". 2021 (ver p. 7).

[Wik21c] Wikipedia. *Game Engine*. https://en.wikipedia.org/wiki/Game_engine. Acessado em: 2021-03-04". 2021 (ver p. 45).

[Wik21d] Wikipedia. *List of Unity games*. https://en.wikipedia.org/wiki/List_of_Unity_games. Acessado em: 2021-03-04". 2021 (ver p. 45).

[Wik21e] Wikipedia. *Platform game*. https://en.wikipedia.org/wiki/Platform_game. Acessado em: 2021-02-14". 2021 (ver p. 7).

[Wik21f] Wikipedia. *Rendering (computer graphics)*. https://en.wikipedia.org/wiki/Rendering_(computer_graphics). Acessado em: 2021-02-13". 2021 (ver p. 6).