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

Aplicação de animação procedural e Inteligências Artificiais independentes do jogador em jogos digitais

Arthur Vieira Barbosa,
Gabriel Sarti Massukado

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Ricardo Nakamura

São Paulo
2021

Agradecimentos

Trust your passion. Believe in your dream.
— Satoru Iwata

Aos meus pais, que sempre me apoiaram, aos amigos que fiz na faculdade, que me ensinaram muito e me ajudaram em tempos difíceis e ao prof. Nakamura que aceitou minha proposta de trabalho e me acompanhou durante o processo.

Gabriel

A todos os amigos de longa data e familiares que me acompanharam nessa jornada de graduação, aos professores que fizeram com que eu chegasse até aqui e aos amigos que fiz na universidade, especialmente àqueles do USPGameDev.

Arthur

Resumo

Arthur Vieira Barbosa, Gabriel Sarti Massukado. **Aplicação de animação procedural e Inteligências Artificiais independentes do jogador em jogos digitais**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

Este trabalho busca implementar entidades controladas por inteligência artificial e animadas proceduralmente em um jogo de plataforma 2D. Animação procedural é um processo que produz animação em tempo real com base em algoritmos geralmente utilizado em jogos tridimensionais, porém questiona-se suas aplicações para gerar animações responsivas em duas dimensões. Foi desenvolvido um *sandbox* jogável usando a *Godot Engine*, implementando mecânicas tradicionais de jogos de plataforma e criaturas com comportamentos aleatorizados, complementadas por animações procedurais baseadas em cinemática inversa e outros métodos. O produto final possui recursos para ser expandido em um jogo completo. Por fim, pondera-se como jogos de outros gêneros poderiam se beneficiar de técnicas semelhantes.

Palavras-chave: Animação procedural. Inteligência artificial em jogos. Jogos de plataforma. *Godot Engine*.

Abstract

Arthur Vieira Barbosa, Gabriel Sarti Massukado. **Aplication of procedural animation and player independent Artificial Intelligence in digital games**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

This work seeks to implement procedurally animated entities controlled by artificial intelligence into a 2D-platforming game. Procedural animation is a process in which animation is produced algorithmically and in real-time and is generally used in 3D games, however its applications in generating responsive 2D animations is something to be explored. A playable sandbox was developed with the Godot Engine, implementing traditional platforming game mechanics and creatures with randomized behaviors, complemented by procedural animations based on inverse kinematics and other methods. The final product has the infrastructure to be expanded into a full game. Finally, this work ponders how similar techniques can be used to benefit different genres of games.

Keywords: Procedural animation. Artificial intelligence in games. Platforming games. Godot Engine.

Lista de Abreviaturas

IA	Inteligência Artificial
FK	Cinemática direta (*Forward Kinematics*)
IK	Cinemática inversa (*Inverse Kinematics*)
FDD	Desenvolvimento guiado a funcionalidades (*Feature Driven Development*)
GDD	Documento de projeto do jogo (*Game Design Document*)
FABRIK	Cinemática inversa alcançando para frente e para trás (*Forward And Backward Reaching Inverse Kinematics*)
CRUD	Criação, consulta, atualização e destruição (*Create, Read, Update and Delete*)
IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo

Lista de Figuras

1.1 Entidade do jogo *Gonner 2* que segue os movimentos do jogador, animada proceduralmente. . . . 2

2.1 Entidade de *Hollow Knight* animada tradicionalmente. . . . 4
2.2 Entidade de *Rain World* animada proceduralmente. . . . 5
2.3 Modo de desenvolvedor de *Rain World*. . . . 5

3.1 Quadros individuais da animação de um personagem de *Hollow Knight*. . 7
3.2 Personagem de *Risk of Rain 2* é derrubado por um golpe e sua animação entra em *ragdoll physics*. . . . 8
3.3 Animação procedural no cabelo da personagem principal de *Celeste*. . . . 8

4.1 Diagrama representando uma configuração de quartos no *sandbox*. . . . 12
4.2 Rascunhos de duas das sete salas que foram implementadas no projeto final. 12

5.1 Representação de *game loop*. . . . 15
5.2 Lista com alguns dos nós da *Godot*, na interface de criação de um novo nó. 17
5.3 Imagem do editor de cenas da *Godot*, mostrando a árvore de uma das cenas do projeto. . . . 18
5.4 Diagrama de classes dos objetos que utilizam física. . . . 19
5.5 Tela de uso de *TileMap* na *Godot*. . . . 20
5.6 Representação gráfica das principais funções de atenuação usadas por *tweens* para interpolar valores. . . . 21

6.1 Aparência final da *Jelly*. . . . 25
6.2 Diagrama de estados da *Barracuda*. . . . 26
6.3 Aparência final da *Barracuda*. . . . 27
6.4 *Jamboard* de *brainstorms* sobre as ações do jogador. . . . 29
6.5 Diagrama de transição de estados do *Player*. . . . 30

7.1 *Player* segurando-se em uma quina. . . . 31

8.1 Cena de uma sala no editor de cenas da *Godot*. 38
8.2 Diagrama mostrando a organização original da *Factory*. 42
8.3 Diagrama mostrando a organização atual da *Factory*. 43
8.4 Vista da cena de *Factory* com a câmera de testes. 44

A.1 Rascunho das quatro criaturas originais: *Barracuda*, *Urchin*, *Shrimp* e *Jelly*. 51
A.2 Rascunho da aparência original do jogador, baseada em um axolote. . . . 52

B.1 Respostas da primeira pergunta aberta. 53
B.2 Respostas da segunda pergunta aberta. 54
B.3 Respostas da terceira pergunta aberta. 54
B.4 Respostas da primeira pergunta qualitativa. 55
B.5 Respostas da segunda pergunta qualitativa. 55
B.6 Respostas da terceira pergunta qualitativa. 55
B.7 Respostas da quarta pergunta qualitativa. 56

Lista de Programas

7.1 Implementação de FK na camada visual da *Barracuda*. 32
7.2 Implementação de FABRIK em *GDScript*. 34
7.3 Código da classe *FillPolygon2D*. 35

Sumário

1 Introdução **1**
1.1 Motivação e Justificativa 1
1.2 Objetivos 2

2 Jogos de plataforma e suas IAs **3**
2.1 *Rain World* 4

3 Animação procedural **7**
3.1 *Forward & Inverse Kinematics* 9

4 Planejamento de projeto **11**

5 Ferramentas e práticas **15**
5.1 *Godot Engine* 15
5.1.1 Noções básicas: nós, cenas e *scripts* 16
5.1.2 Sistema de física 17
5.1.3 *Tilemap* 18
5.1.4 *Pathfinding* 19
5.1.5 *Tween* 19
5.1.6 *Singletons* 20
5.2 Outras ferramentas 20

6 Criaturas e o jogador **23**
6.1 Classes base 23
6.1.1 Camada física e camada estética 24
6.2 Primeira criatura: *Jelly* 24
6.2.1 Animações 24
6.3 Segunda criatura: *Barracuda* 25
6.3.1 Animações 28

6.4 Jogador e seus controles 28

7 Técnicas de animação **31**
7.1 *Forward Kinematics* 32
7.2 *Inverse Kinematics* 32
7.3 *FillPolygon2D* 33

8 Gerenciadores e outras funcionalidades **37**
8.1 Salas 37
8.2 *Ledges* e Plataformas 38
8.3 *Conveyors* 38
8.4 Objetos 39
8.5 Tabelas de gerenciamento 40
8.5.1 Tabela de Criaturas 40
8.5.2 Tabela de Salas 41
8.6 *Factory* 41

9 *Playtesting e últimos ajustes* **45**
9.1 Lançamento do *playtest* 45
9.2 *Feedbacks* do *playtest* 45
9.3 Ajustes finais 46

10 Conclusão **49**
10.1 Resultados 49
10.2 Próximos passos 50

Apêndices

A Rascunhos e artes conceituais **51**

B Dados de *playtest* **53**

Anexos

A *Game Design Document* **57**

B Repositório no *Git* **59**

C Versões animadas **61**

Referências **63**

Capítulo 1

Introdução

1.1 Motivação e Justificativa

Os jogos digitais (mais popularmente conhecidos como *video games*, em inglês) estão diretamente relacionados com a forma que um computador é capaz de apresentar informação para o jogador, pois, nas palavras de Jesse Schell, "sem a experiência, um jogo não vale nada" [1]. Por mais que um software frequentemente esteja diretamente atrelado a alguma forma de interação entre usuário e máquina, os jogos digitais sempre tiveram destaque em sua ênfase aos recursos utilizados para isso.

Uma resposta adequada é parte essencial da interação humano-humano e também da interação humano-computador (Pérez-Quiñones e Sibert, 1996). Diferente de outros meios que usufruem de animações - como filmes e ilustrações animadas - a parte visual de um jogo está em constante mudança de acordo com a influência de um jogador. Embora isso geralmente já seja refletido diretamente por meio de um avatar que o representa, quando outras entidades do jogo reagem de forma perceptível, como no exemplo da Figura 1.1, é gerado um *feedback* que torna mais positiva a experiência de interação.

Jogos de plataforma, os quais Shigeru Miyamoto inicialmente chamou de "jogos atléticos" por consistirem principalmente de um personagem dando pulos pela tela (Gifford, 2010), são presentes na indústria desde seu surgimento. Entretanto, muitos deles optam por simplificar o comportamento de suas entidades que não são controladas pelo jogador, seja por decisões de *design* ou por restrições de orçamento ou recursos técnicos. Um aspecto atualmente pouco explorado é o uso de animações 2D geradas proceduralmente, ou seja, animações que são geradas em tempo real por meio de processos algorítmicos e que, consequentemente, permitiriam que elementos do jogo produzissem *feedbacks* mais dinâmicos e interativos à relações entre o jogador e o jogo. Ademais, a animação procedural abre novas possibilidades visuais nas entidades controladas pelo jogo. Mesmo com uma inteligência artificial simples, uma criatura animada proceduralmente, por exemplo, pode aparentar uma gama de comportamentos menos previsíveis do que

[1] "*Without the experience, the game is worthless*", em tradução livre (Schell, 2020, p. 10).

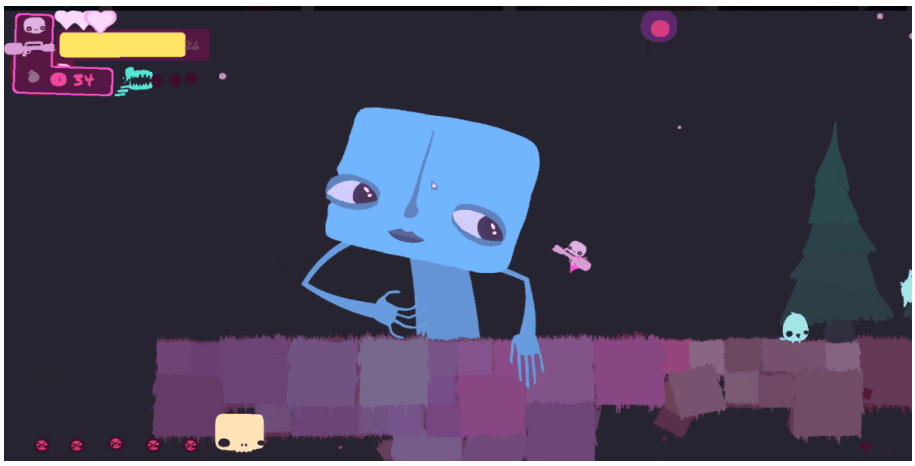

Figura 1.1: *Entidade do jogo Gonner 2 que segue os movimentos do jogador, animada proceduralmente. Versão animada disponível no Anexo C (Versões animadas).*

uma animada com um conjunto de poucas imagens.

1.2 Objetivos

O presente trabalho tem como objetivo o desenvolvimento de um jogo simples para estudar como IAs e animações procedurais podem criar uma experiência diferente de jogo de plataforma. O foco do jogo será em integrar elementos de plataforma tradicionais, como poder realizar vários tipos de movimentos para se locomover pela tela, com criaturas animadas proceduralmente e controladas por IAs mais complexas do que as normalmente encontradas em jogos desse gênero.

Além disso, discorrer-se-á sobre o processo de desenvolvimento e as decisões tomadas durante ele, também contextualizando alguns detalhes relevantes sobre jogos de plataforma, animação procedural e *game design*.

Capítulo 2

Jogos de plataforma e suas IAs

Um jogo de plataforma, termo cunhado após o lançamento de ***Donkey Kong*** (1981) [1] e outros jogos semelhantes por volta dos anos 80/90 (Kohler, 2016, p. 54), é um tipo de jogo digital em que o jogador controla um personagem cujas ações geralmente enfatizam a movimentação pela tela. Por meio delas, o personagem se desloca entre diferentes superfícies - "plataformas" - e interage com o mundo ao seu redor, exigindo que o jogador supere diversos obstáculos e desafios criados pelo *game designer*. Nas palavras de Elias, Garfield e Gutschera (2012, p.286), um jogo de plataforma é aquele em que o jogador navega um mapa - pulando, escalando e desviando de obstáculos.

Quando tratamos de IA em jogos de plataforma, estamos falando principalmente de obstáculos. Os inimigos de ***Super Mario Bros.*** (1985) [2] têm seu comportamento limitado a padrões simples, como por exemplo patrulhar uma área horizontal ou vertical ou disparar um projétil no jogador caso ele se aproxime e, por mais que isso pareça excessivamente simples, foi e ainda é suficiente para deixar a experiência de jogo mais interessante. Grande parte dos jogos de plataforma em duas dimensões ainda seguem modelos semelhantes, como por exemplo ***Hollow Knight*** (2017) [3], um título recente de sucesso mundial cuja maior parte dos inimigos faz uso de padrões facilmente identificáveis de movimentação (ver Figura 2.1). Isso acontece pois é um objetivo de design razoável: o jogador é recompensado por observar as ações dos inimigos, entendê-las e sobrepujá-las (Millington e Funge, 2009, p. 819).

Outro aspecto interessante a ser comentado é a falácia da complexidade, ou seja, a pretensão de que um jogo com uma IA mais complexa necessariamente resulta em uma experiência de jogo melhor. Criar uma IA boa para um jogo muitas vezes implica usar métodos mais simples ou que sejam mais esteticamente agradáveis para o jogador, mesmo que isso seja resultante de escolhas mal-vistas no meio acadêmico (Millington e Funge, 2009, p. 19). Mesmo assim, é interessante considerar que tipo de experiências de jogo

[1] Página de *Donkey Kong* na IGDB.

[2] Página de *Super Mario Bros.* na IGDB.

[3] Página de *Hollow Knight* na IGDB.

Figura 2.1: *Inimigo de Hollow Knight patrulhando uma área, animado tradicionalmente. Versão animada disponível no Anexo C (Versões animadas).*

podem ser criadas em jogos de plataforma ao se explorar as IAs de agentes não controlados pelo jogador com um pouco mais de profundidade.

2.1 *Rain World*

Rain World (2017) [4] é um jogo de plataforma e sobrevivência que utiliza IA para controlar diversas criaturas que interagem como presas ou predadores do jogador como pode ser visto na Figura 2.2. Além disso, cada criatura interage também com outras mesmo quando fora da tela do jogo, criando uma sensação de um mundo vivo e altamente interativo. Esse fator é somado com outras implementações interessantes, como múltiplos parâmetros de personalidade selecionados aleatoriamente para cada criatura, para criar uma experiência de jogo de plataforma diferente e inovadora. Usando animações proceduralmente geradas para suas criaturas e repleto de decisões de *design* únicas, ***Rain World*** foi uma das principais inspirações por trás desse trabalho.

Analisando as implementações do jogo (ver Figura 2.3), pode-se ver que ele divide as telas jogáveis em "quartos" e cada um deles possui dois estados: *realized* e *abstract*, isto é, concretizado e abstrato respectivamente, em inglês. Dessa forma, os sistemas internos de ***Rain World*** sabem quando precisam simular uma criatura por completo ou apenas simplificar seus movimentos e comportamentos, criando a ilusão de um mundo vivo que é descoberto pelo jogador constantemente. Isso permite que cada agente do jogo esteja perpetuamente sofrendo influências de seus próprios valores aleatórios e de interações com outras criaturas, mesmo quando fora do campo de visão do jogador. Mesmo com um mapa e mundo de jogo sempre igual, ***Rain World*** consegue diversificar cada passagem por um quarto e se renovar inúmeras vezes em uma única sessão de jogo, trazendo uma experiência única e sobremaneira diferente de outros jogos de plataforma que aplicam geração procedural em suas implementações - geralmente usada para aleatorizar

[4] Página oficial de *Rain World*.

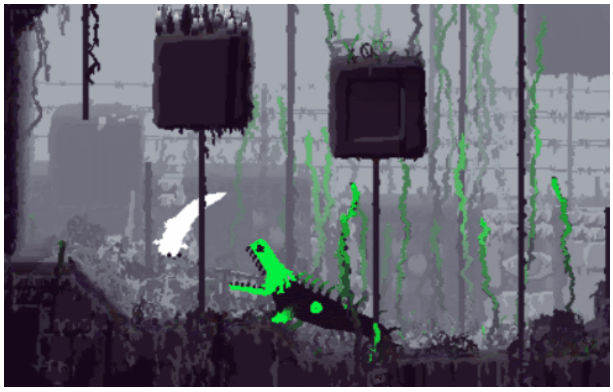

Figura 2.2: *Lagarto verde de Rain World reagindo ao jogador, ambos animados proceduralmente. Versão animada disponível no Anexo C (Versões animadas).*

combinações de níveis ou posições de itens importantes.

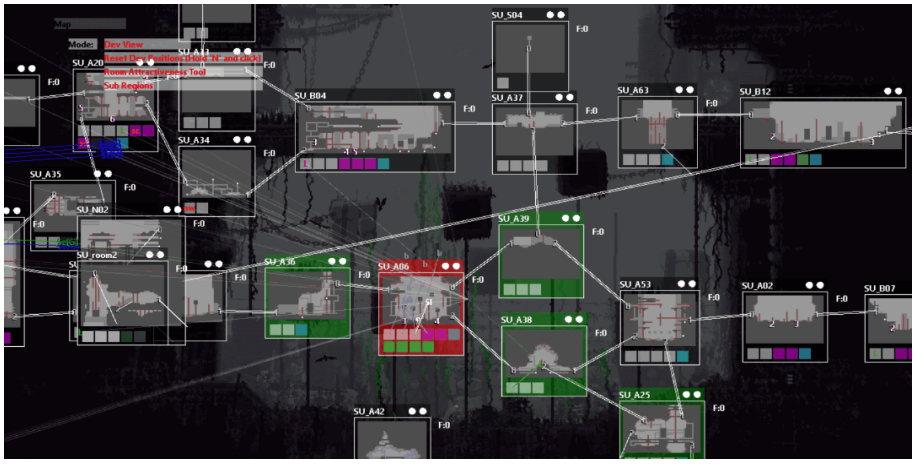

Figura 2.3: *Modo de desenvolvedor de Rain World. É possível ver o quarto do jogador (em vermelho) e os outros quartos concretizados (em verde), além de siglas que representam criaturas se movendo em quartos abstratos. Versão animada disponível no Anexo C (Versões animadas).*

Porém, por mais que esse sistema seja deveras engajante, ele tem um custo de recursos e uma consequência na experiência de jogo. Em primeiro lugar, o número grande de simulações com múltiplas possibilidades de resultados acontecendo em *background* não é otimizado o suficiente para um desempenho consistente. Frequentemente, entrar e sair de quartos gera quedas na taxa de quadros e o *pathfinding* ("busca de caminhos", em inglês) de criaturas fora da tela pode escalar de forma descontrolada e congelar o jogo temporariamente. Em segundo e último lugar, ***Rain World*** se torna um jogo bastante difícil de se entender e de se jogar devido à quantidade de aleatoriedade presente em seus sistemas. Por mais que a geração procedural possa ser usada para proporcionar desafios e experiências diferentes (Cunha, 2020, p. 2), um conjunto de comportamentos gerados aleatoriamente que se retroalimentam podem sair do controle do jogador com bastante facilidade. Mesmo ponderando que esse segundo fator seja intencional para representar a vulnerabilidade do personagem principal em uma situação de "presa-predador", seria

importante considerar suas possíveis implicações negativas na experiência de jogo.

Estudando com detalhe os sucessos e falhas de ***Rain World***, é possível melhor guiar o desenvolvimento de um jogo de plataforma que busca experimentar a implementação de mecânicas e dinâmicas incomuns no gênero.

Capítulo 3

Animação procedural

Para descrever animação procedural, é necessário explicar primeiro o processo tradicional de animação. Animação, como um termo geral, descreve uma sequência de quadros estáticos que, quando reproduzidos em sequência, geram ilusão de movimento ao serem representados em um meio de visualização bidimensional como pode ser visto na Figura 3.1. Em jogos, geralmente cada animação está associada a um certo estado de uma entidade ou elemento do jogo, como por exemplo andar, pular, atacar e assim por diante.

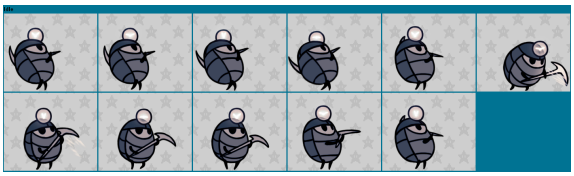

Figura 3.1: *Quadros individuais da animação de um personagem de Hollow Knight. Disponível em The Spriter's Resource, acesso em 18/12/2021.*

Em uma animação tradicional, quando se trata de *video games* 2D, cada quadro é desenhado individualmente e apenas a reprodução deles é controlada pelo jogo. Isso implica que, independente do meio utilizado para desenhar cada quadro, esse passo da animação é feito conscientemente e manualmente - o trabalho do programador é restrito a utilizar os quadros disponíveis. Enquanto isso, na animação procedural, cada quadro é gerado em tempo de execução por um processo algorítmico, ou seja, não é necessário elaborar as imagens de antemão. Isso permite que uma entidade de jogo tenha um número de estados maior ou uma transição de estados mais interessante visualmente, porém é diretamente atrelado com a capacidade do programador de escrever um código robusto e eficiente, que possa lidar com o processo de animação em tempo real.

Naturalmente, um *video game* não precisa se restringir a apenas um método de animação. Jogos em 3D são famosos por utilizar animações procedurais para facilitar o processo de animação de seus modelos, enquanto animações que precisam de maior detalhe são feitas utilizando o processo tradicional. Isso é facilmente identificável quando um modelo entra em "*ragdoll physics*" ("física de boneco de pano", em inglês), ou seja, seu

corpo e animações passam a ser controlados apenas por forças aplicadas sobre ele, como a gravidade no exemplo da Figura 3.2.

Figura 3.2: *Personagem de Risk of Rain 2 é derrubado por um golpe e sua animação entra em ragdoll physics. Versão animada disponível no Anexo C (Versões animadas).*

Ademais, jogos 2D também frequentemente optam por implementar animações procedurais em apenas partes de seus recursos visuais, como por exemplo a personagem controlada pelo jogador em ***Celeste (2018)*** [1]. Nela, todo o seu corpo tem animações tradicionais, enquanto seu cabelo é animado em tempo real para refletir os movimentos do jogador conforme ele se desloca pelas plataformas (ver Figura 3.3), como explicado por Berry (2018), um dos desenvolvedores do jogo.

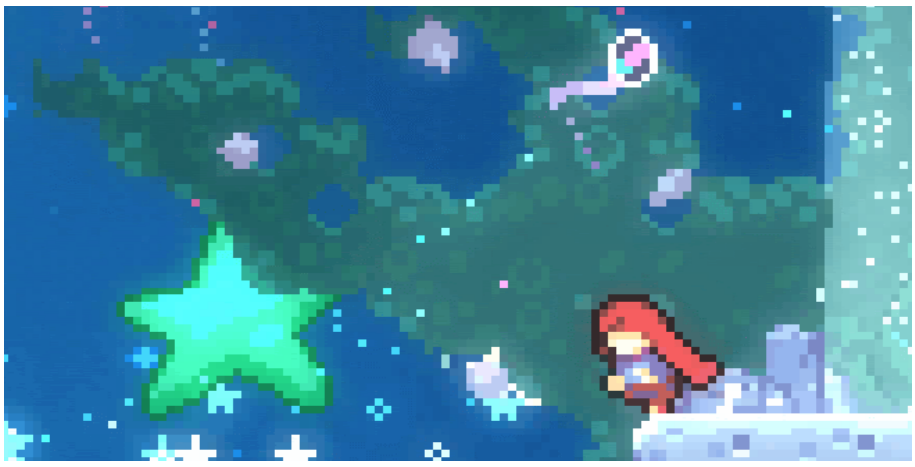

Figura 3.3: *Animação procedural no cabelo da personagem principal de Celeste. Versão animada disponível no Anexo C (Versões animadas).*

Vale notar que ambos os termos que foram utilizados até o momento para descrever tipos de animação não são categorias formalmente definidas e restritivas, ou seja, "animação tradicional" e "animação procedural" podem englobar diversas técnicas distintas de

[1] Página de *Celeste* na IGDB

animação cada uma. Entretanto, o termo "animação procedural" continuará sendo usado para se referir ao conceito explicado anteriormente e as técnicas utilizadas serão descritas de forma aprofundada quando necessário.

3.1 *Forward & Inverse Kinematics*

Durante o desenvolvimento do projeto, grande parte das animações procedurais elaboradas se basearam em dois métodos centrais: *Forward Kinematics* e *Inverse Kinematics*, ou seja, cinemática direta e inversa respectivamente em inglês. Originários do campo da robótica, os termos se referem ao uso de equações de cinemática para calcular ângulos e posições das juntas de um braço robótico. A cinemática frontal consiste do uso da angulação de cada junta para calcular a posição de um efetor final (do inglês "*end effector*") - isto é, o ponto final de um braço robótico por exemplo - enquanto a cinemática inversa busca encontrar cada ângulo das juntas do braço com base em seus tamanhos e da posição final (Saeed e Fadhil, 2020).

Esses processos tem aplicações diferentes, mas ambos podem ser utilizados para animar um modelo cujas partes são segmentadas, gerando resultados esteticamente agradáveis e mecanicamente funcionais. Sua relevância na animação é tamanha que motores de jogos como a *Unity*[2], que é considerada referência no mercado de jogos, e a *Godot Engine*, que é utilizada nesse trabalho e será detalhada na Seção 5.1, já possuem implementações de IK embutidas para auxiliar os desenvolvedores[3] [4]. Porém, como o presente trabalho pretende estudar mais a fundo os passos por trás da animação procedural, os métodos e *scripts* responsáveis pela animação serão implementados em sua totalidade - e descritos no texto nas seções relevantes.

[2] Página da *Unity*, acesso em 19/12/2021

[3] Inverse Kinematics na *Unity*.

[4] IK Chains, na *Godot Engine*.

Capítulo 4

Planejamento de projeto

Os primeiros passos para começar o desenvolvimento foram uma seção de *brainstorming* e a criação de um *Game Design Document*, ou seja, um documento que contém a visão completa do design do jogo (Brathwaite e Schreiber, 2009, p. 14). Mesmo com o entendimento de que o escopo do projeto provavelmente seria limitado pelos prazos do trabalho de conclusão de curso, foi decidido que o GDD deveria contemplar decisões de design do jogo completo, de forma que pudesse ser referenciado quando necessário para nortear o processo de desenvolvimento. No GDD foram definidos aspectos importantes como as mecânicas e ações do jogador, os tipos de criaturas controladas pela IA do jogo e as diretrizes básicas de seus comportamentos, assim como motivações estéticas e outras informações relevantes para o desenvolvimento do jogo. Ademais, foi decidida a ambientação do jogo: o jogador controlaria um robô que explora uma fábrica submarina, vasculhando seus espaços inundados e examinando as criaturas das profundezas. O GDD completo está disponível no Anexo A (*Game Design Document*), porém vale notar que nem todas as ideias conceituadas foram implementadas no projeto final.

Em resumo, seria construído um *sandbox* (caixa de areia, em inglês), ou seja, um conjunto de cenários de teste em que tanto desenvolvedores como jogadores pudessem testar o jogo, e implementadas nele as mecânicas e principais conceitos que foram planejados: criaturas que interagem entre o jogador e entre si e um personagem de jogador que pudesse interagir com elas, ambos animados proceduralmente e atuando em níveis de plataforma 2D. O *sandbox* seria composto de sete "salas" ou "quartos" (*rooms*, em inglês) separadas por "portas transportadoras" (*conveyors*[1], em inglês), sendo que cada sala seria preenchida com diferentes plataformas e obstáculos. Dessa forma, seria produzida uma experiência de jogo de plataforma e exploração para o jogador, ao mesmo tempo que seria criado um espaço para testar os comportamentos das criaturas em relação ao ambiente. Além disso, para aliviar o custo de processamento, foi determinado que apenas as criaturas na mesma sala que o jogador seriam simuladas por completo, enquanto as criaturas em salas adjacentes seriam simuladas sem a parte gráfica e as criaturas nas salas restantes seriam abstraídas para uma tabela que calcularia aleatoriamente interações mais simples.

[1] O termo *conveyors* será usado com maior frequência para se referir às portas transportadoras por tornar a leitura mais conveniente.

A Figura 4.1 mostra como as salas seriam conectadas - salas com maior número de salas adjacentes são coloridas em tons mais próximos ao vermelho, indicando que o custo de seu processamento seria maior.

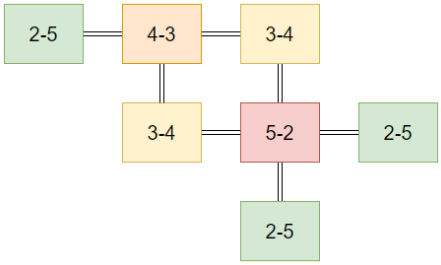

Figura 4.1: *Diagrama representando uma configuração de quartos no sandbox. O quarto central, denotado com o valor **5-2** indica que, se o jogador estiver nessa sala, cinco salas estão sendo simuladas (ela própria e suas quatro vizinhas) enquanto duas salas não-adjacentes são abstraídas para a tabela.*

A última parte do planejamento inicial foi o rascunho de artes conceituais e o desenvolvimento de protótipos em papel (ver Figura 4.2). Desenhando cada sala da *sandbox*, foi possível focar no *level design* nesse momento inicial, sem se preocupar necessariamente com a programação. Com isso, pode-se avaliar que tipo de impressão e experiência que o jogo propõe e conseguir opiniões externas, tornando o desenvolvimento a seguir mais eficiente e direcionado.

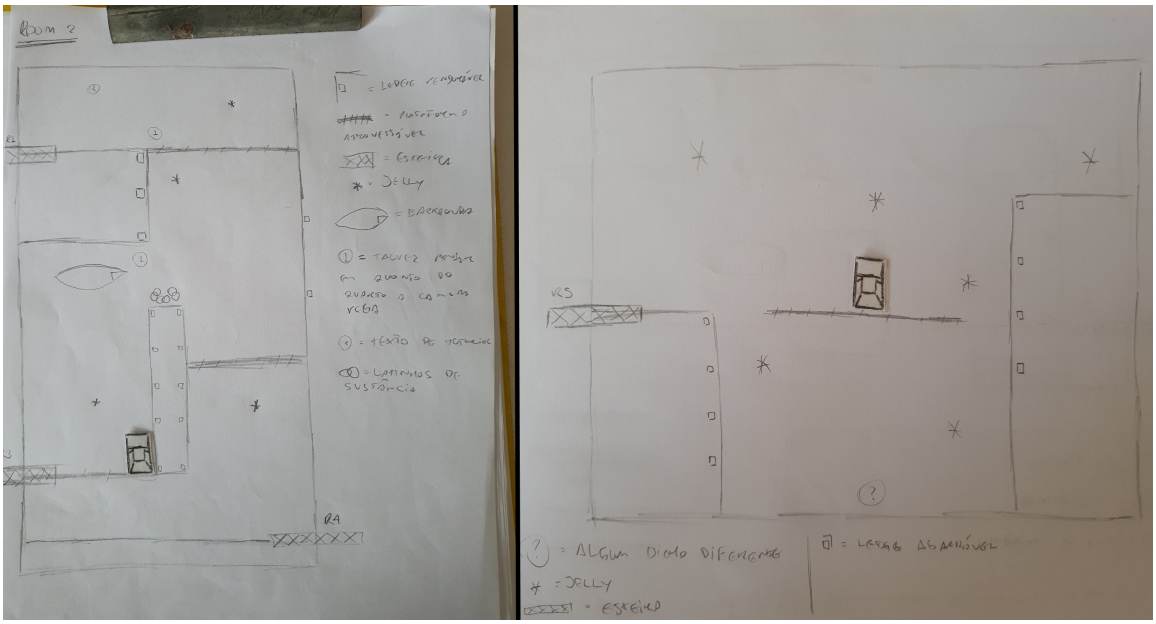

Figura 4.2: *Rascunhos de duas das sete salas que foram implementadas no projeto final. Cada sala foi desenhada considerando tanto elementos de plataforma que seriam interessantes para o jogador explorar como áreas que permitissem simulação das criaturas.*

Além disso, foram feitos rascunhos das sete salas, do personagem do jogador e de quatro criaturas, inspiradas visualmente em animais marinhos:

- *Jelly* - inspirada em águas-vivas, a *Jelly* é uma criatura completamente passiva que habita a fábrica submarina, sendo autossuficiente e servindo como alimento para outros seres vivos. Ela é simples em todos os aspectos possíveis e é usada para popular o mundo do jogo em geral.
- *Barracuda* - inspirada na espécie de peixe homônima, a *Barracuda* é uma criatura agressiva que ataca tanto o jogador como as outras criaturas que passam por seu campo de visão, sendo veloz e violenta. Ela também pode fugir quando sente que não conseguirá triunfar em combate e é encontrada em números bem menores pelo ambiente.
- *Urchin* (descartada) - inspirado em ouriços-do-mar, o *Urchin* é uma criatura que passa a maior parte do tempo parada e é envolvida em espinhos, que a protegem de predadores.
- *Shrimp* (descartada) - inspirado em camarões-pistola, o *Shrimp* é uma criatura que também passa a maior parte do tempo parado, mas caça de forma agressiva e territorial criaturas que se aproximam.

Ao rever o escopo do projeto por volta do mês de setembro, tanto o *Urchin* como o *Shrimp* foram desconsiderados para que houvesse um foco maior em outros aspectos do projeto, como será detalhado na Seção 9.2 *Feedbacks* do *playtest*. Além disso, é possível ver as artes conceituais de todas as criaturas originalmente planejadas no Apêndice A (Rascunhos e artes conceituais), com alguns detalhes como o planejamento dos comportamentos, camadas físicas e camadas estéticas de cada um.

Capítulo 5

Ferramentas e práticas

Este capítulo detalha as principais ferramentas utilizadas durante o desenvolvimento do projeto.

5.1 *Godot Engine*

A natureza interativa dos jogos eletrônicos faz com que os seus softwares utilizem um padrão de código em comum, conhecido como *game loop* (Nystrom, 2014). Nesse padrão, ilustrado na Figura 5.1, o programa continuamente recebe a entrada do usuário, atualiza o estado do jogo de acordo e então renderiza o resultado para que o jogador possa ver o novo estado. A cada execução desse laço é dado o nome de *frame* (traduzido como "quadro", em português).

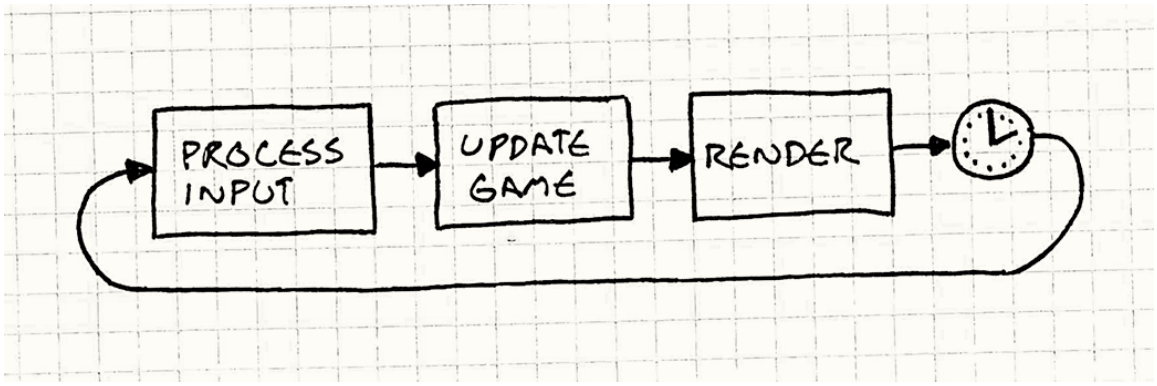

Figura 5.1: *Representação de game loop. Disponível em Game Loop · Sequencing Patterns · Game Programming Patterns, acesso em 19/12/2021.*

Com o intuito de simplificar o desenvolvimento, foram desenvolvidos arcabouços e motores de jogos que além de implementar o *game loop* também possuem componentes capazes de fornecer funcionalidades comumente utilizadas na maioria dos jogos, como a renderização de gráficos, reprodução de sons, simulação física, entre outros (Gregory, 2019, pp. 11-13). Os motores de jogos, do inglês *game engines*, tem o mesmo objetivo

dos arcabouços, que são uma técnica de reutilizar tanto código quanto padrões de *design* (Johnson, 1997), porém enquanto os arcabouços apenas definem interfaces que abstraem os componentes de mais baixo nível das aplicações, os motores de jogos além disso fornecem as implementações prontas de tais componentes (Richards, 2010). Muitos desses motores também são aplicações completas, com suas próprias interfaces gráficas, editores de texto, ferramentas de depuração e outras funções.

A *Godot Engine* [1] é uma *game engine* leve, versátil, *open source* sob a licença MIT e mantida por uma comunidade independente de colaboradores. Por esses motivos, em conjunto com a familiaridade e facilidade de uso dos autores deste trabalho, ela foi a escolhida para esse projeto dentre os outros diversos motores disponíveis. Assim, serão apresentados nas seções seguintes os conceitos e componentes da *Godot Engine* - que será referenciada daqui para frente apenas como *Godot* - mais importantes para o entendimento desta monografia.

5.1.1 Noções básicas: nós, cenas e *scripts*

Nós são os elementos básicos de um jogo feito na *Godot*, e ela já providencia vários com funcionalidades diferentes para a implementação de jogos 2D e 3D de diversos tipos (ver Figura 5.2). Um nó sempre tem as seguintes propriedades:

- Tem um nome;
- Tem propriedades editáveis;
- Pode receber um *callback* pra ser processado a cada *frame*;
- Pode ser estendido para ter mais funções;
- Pode ser adicionado a outro nó como seu filho.

Quando temos um conjunto de nós relacionados uns aos outros chamamos essa estrutura de **árvore** e o nó que não é filho de nenhum outro é o **nó raiz** dessa árvore.

Ao criarmos uma árvore no editor da *Godot* produzimos uma cena (ver Figura 5.3), que tem as seguintes propriedades:

- Sempre tem um nó raiz;
- Pode ser salva no disco e carregada dele, no formato ".tscn";
- Pode ser instanciada (como será visto a seguir).

Dentro do editor da *Godot* pode-se rodar qualquer cena, porém as consequências mais importantes dessa estrutura são que um jogo pode conter diversas cenas e que rodar o jogo equivale a executar uma cena principal, que pode conter ou fazer a transição para outras cenas. Quando adicionamos uma cena pronta a outra, seja pelo editor ou durante a execução do jogo, chamamos isso de instanciação. Ao instanciarmos uma cena, decidimos qual nó da cena original vai ser o pai de seu nó raiz e então essa cena e seus nós passam a

[1] *Página da Godot Engine*, acesso em 19/12/2021.

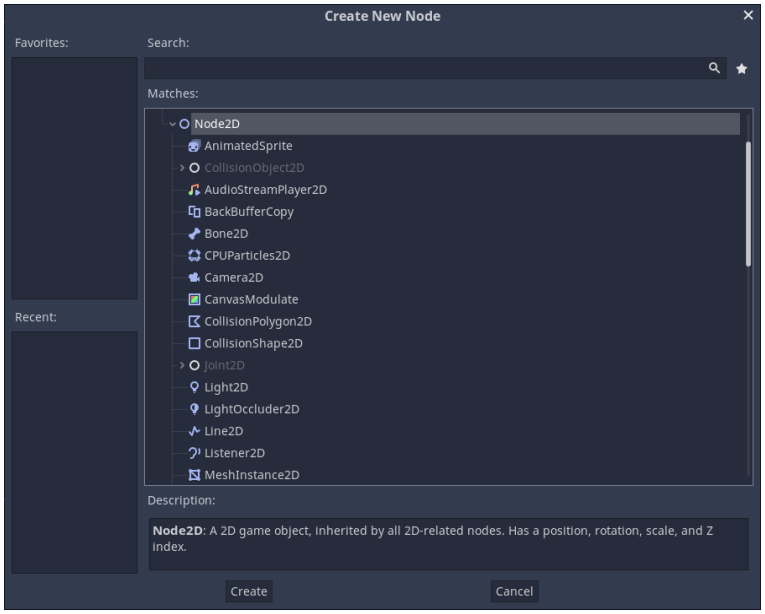

Figura 5.2: *Lista com alguns dos nós da Godot, na interface de criação de um novo nó.*

fazer parte da árvore da cena original.

Como foi dito antes, os nós providenciados pela *engine* podem ter seus comportamentos estendidos. Pelo ponto de vista do usuário isso é feito pela adição de *scripts* aos nós, adicionando novas funcionalidades ou modificando as já existentes. A *Godot* suporta oficialmente quatro linguagens de *scripting*:

- *GDScript*: uma linguagem feita propriamente para o uso da *Godot*, gerando vantagens como facilidade de uso e alta integração com o editor;
- *VisualScript*: assim como *GDScript*, porém se utiliza de programação visual com uma linguagem de blocos e conexões, podendo deixar o código mais acessível a não-programadores;
- C#: adicionada por ser a linguagem de escolha de outras *game engines* relevantes, como a *Unity*;
- C++: a melhor escolha em função de desempenho, pode ser usada tanto no projeto inteiro quanto apenas nas partes que apresentam *bottlenecks* de performance.

Foi utilizada a linguagem *GDScript* neste projeto devido às vantagens citadas somadas à experiência prévia dos autores.

5.1.2 Sistema de física

Durante o desenvolvimento de jogos frequentemente precisamos saber quando dois objetos entram em contato ou se intersectam, e então gerar um resultado a partir disso. Esses conceitos são conhecidos como detecção de colisão e resposta de colisão respectivamente. A *Godot* providencia um motor de física com diversos nós para lidar com tais mecânicas tanto em 2D quanto em 3D. Para representar um objeto que permite detectar colisão em um

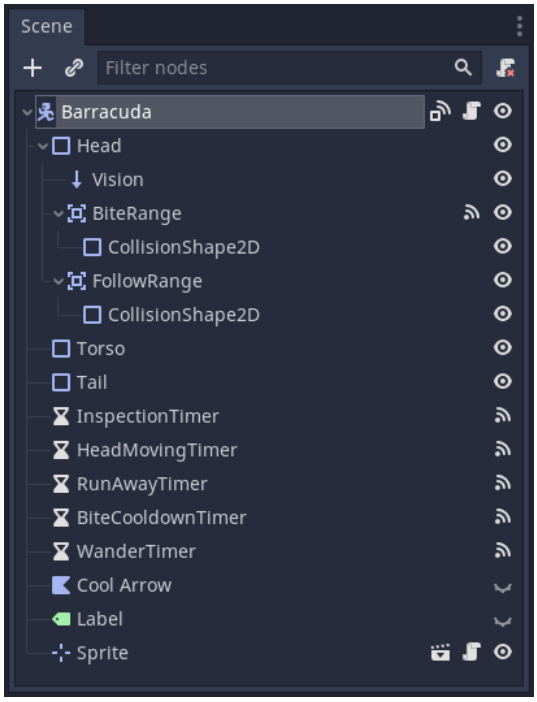

Figura 5.3: *Imagem do editor de cenas da Godot, mostrando a árvore de uma das cenas do projeto.*

jogo 2D, temos quatro nós que tem como classe base o nó *CollisionObject2D*, que contém apenas informação sobre os formatos de tais objetos. Esses quatro nós são:

- *Area2D*: um nó que proporciona detecção de outras áreas e corpos, porém não interage fisicamente com eles;
- *StaticBody2D*: um corpo que participa da detecção de colisão mas não se move em resposta, usado geralmente para parte do terreno ou objetos que não necessitam de um comportamento dinâmico;
- *RigidBody2D*: um nó que implementa física simulada e não é controlado diretamente - uma força é aplicada a ele e o motor de física calcula o movimento resultante;
- *KinematicBody2D*: um corpo que proporciona detecção de colisão mas não participa da resposta no motor de física - todos os movimentos e respostas de colisão precisam ser implementados pelo usuário.

No projeto foram utilizados os quatro nós, como pode ser visto na Figura 5.4.

5.1.3 *Tilemap*

Uma técnica bem estabelecida para a criação de níveis em jogos 2D é a utilização de *tiles*, elementos gráficos que representam parte do cenário. A esse conjunto de tiles dispostos numa grade damos o nome *tilemap*. Os *tilemaps* permitem a criação rápida de novos *layouts* de fases e a modificação dos existentes, agilizando o processo de *level design* (desenho de nível, em inglês). Na *Godot* o nó responsável por tal finalidade é o *TileMap* (ver Figura 5.5), que além de permitir desenhar os *tiles* ainda possibilita adicionar outras funcionalidades a eles, como formas de colisão e navegação para auxiliar os sistemas de

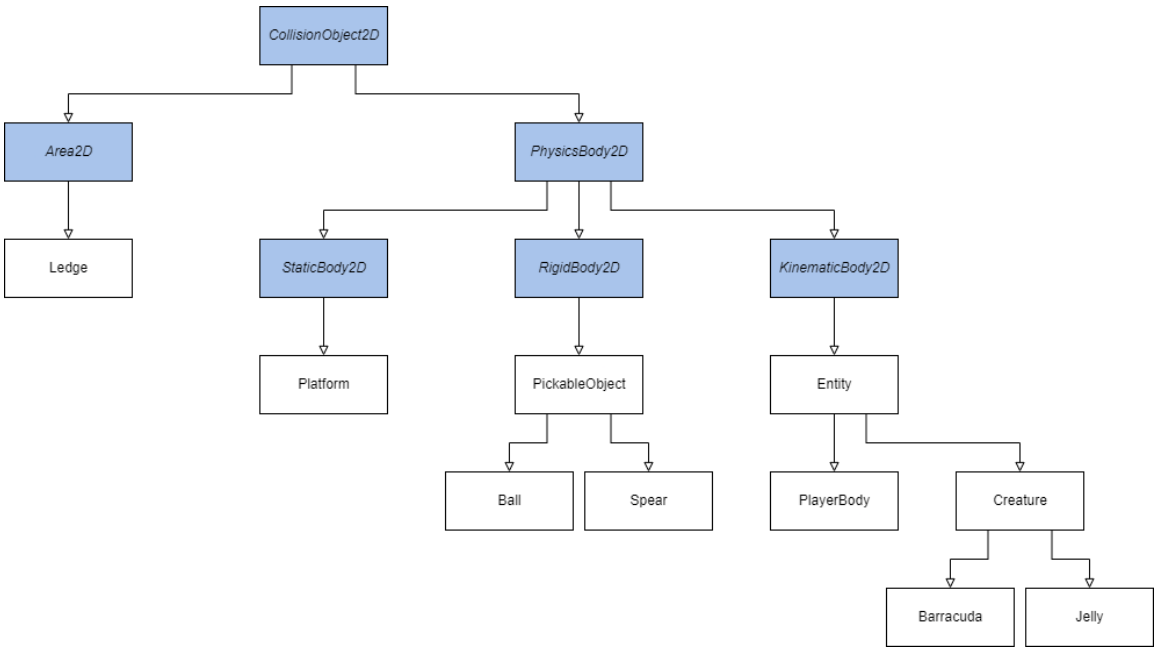

Figura 5.4: *Diagrama de classes dos objetos que utilizam física. Representa-se em azul as classes integradas na Godot Engine e em branco as implementadas especificamente para o projeto.*

física e *pathfinding* respectivamente.

5.1.4 *Pathfinding*

Pathfinding no contexto do desenvolvimento de jogos é a aplicação de inteligência artificial para se traçar o menor caminho possível entre dois pontos. Uma implementação de *pathfinding* geralmente consiste de um grafo representando um mapa do jogo, um algoritmo de busca e uma função heurística para guiar a busca (Botea *et al.*, 2013). O nó utilizado da *Godot* que implementa essa funcionalidade foi o *Navigation2D*, que utiliza o algoritmo de busca A* [2] para providenciar métodos de se achar caminhos dentro de áreas definidas por nós *NavigationPolygonInstance* filhos dele. Neste trabalho foi aproveitada a característica do *TileMap* de ter polígonos de navegação associados aos *tiles* para prover as informações ao *Navigation2D*, e o uso desse nó foi de extrema importância para implementar a inteligência artificial das criaturas, como veremos numa seção futura.

5.1.5 *Tween*

O nome *tween* vem de *in-betweening*, uma técnica de animação que consiste em especificar quadros-chave e deixar com que o computador interpole os valores para os quadros que aparecem entre eles. Assim, *tweens* são úteis quando se precisa que uma propriedade numérica de um elemento seja interpolada por valores que podem ser previamente desconhecidos. Na animação procedural, os valores de propriedades que representam um movimento são geralmente decididos em tempo de execução e dependentes de fatores externos ao componente que está sendo animado, portanto o uso

[2] Leia mais sobre esse algoritmo em Wikipedia - A* search algorithm

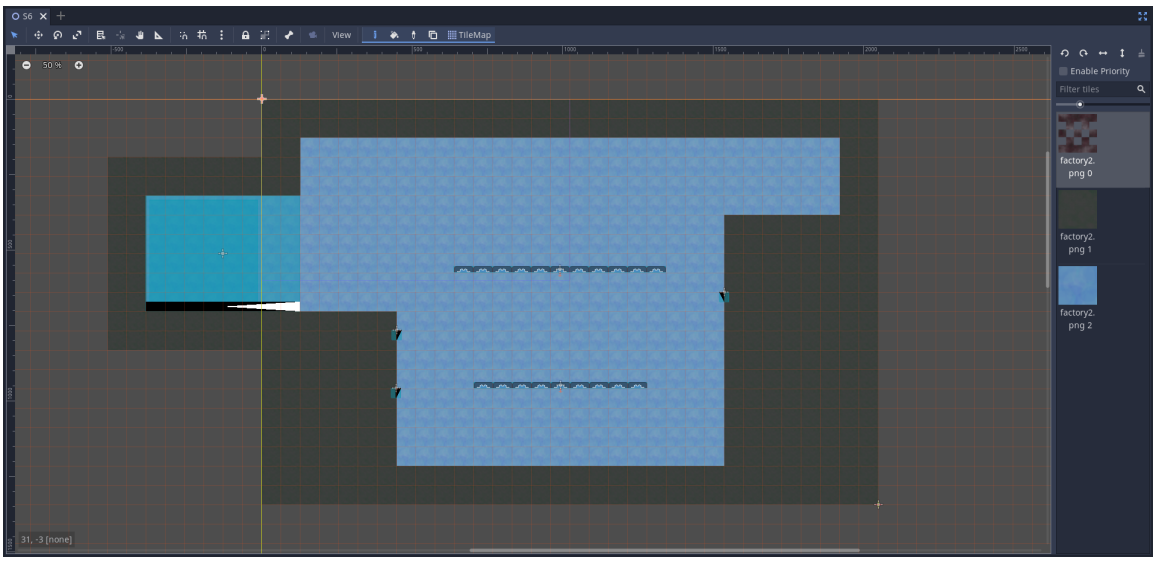

Figura 5.5: *Tela de uso de TileMap na Godot, mostrando as tiles utilizadas para desenhar os níveis no projeto.*

de *tweens* é frequente nesse projeto para fazer com que as animações ocorram suavemente.

Na *Godot*, a implementação dessa técnica se dá pelo nó *Tween*, que possui métodos para interpolar valores de atributos de um dado nó. Isso é feito passando ao método o nó, o nome de sua propriedade que se quer interpolar, um valor inicial, um valor final, e uma duração. Por padrão a interpolação feita é linear com o tempo, mas adicionalmente podem ser passadas outras constantes que definem a forma que a interpolação vai tomar, resultando em transições diferentes que podem ser vistas na Figura 5.6.

5.1.6 *Singletons*

Quando é necessário que uma informação persista entre cenas diferentes, a *Godot* se utiliza do padrão de projeto *singleton* criado pela ilustre *"Gang of Four"* (Gamma *et al.*, 1995). Assim é possível criar cenas e as definir na *engine* como *Singletons*, também chamados de *AutoLoads*, que vão ser sempre instanciadas no começo da execução do jogo e continuarão instanciadas independentemente de qual cena está sendo rodada. É importante notar que apesar de terem o mesmo nome e propósito do padrão de projeto, os *Singletons* na *Godot* não são verdadeiramente como os propostos pela *"Gang of Four"*, pois nada impede o usuário de instanciar mais de uma cena definida como *Singleton*. Neste trabalho *Singletons* são utilizados para implementar objetos de gerenciamento de salas e criaturas, como veremos no Capítulo 8 Gerenciadores e outras funcionalidades.

5.2 Outras ferramentas

Além da *Godot*, outras ferramentas que não precisam de uma descrição tão detalhada foram utilizadas no projeto. Além disso, houve algumas práticas e princípios seguidos que também são relevantes e estão registrados aqui.

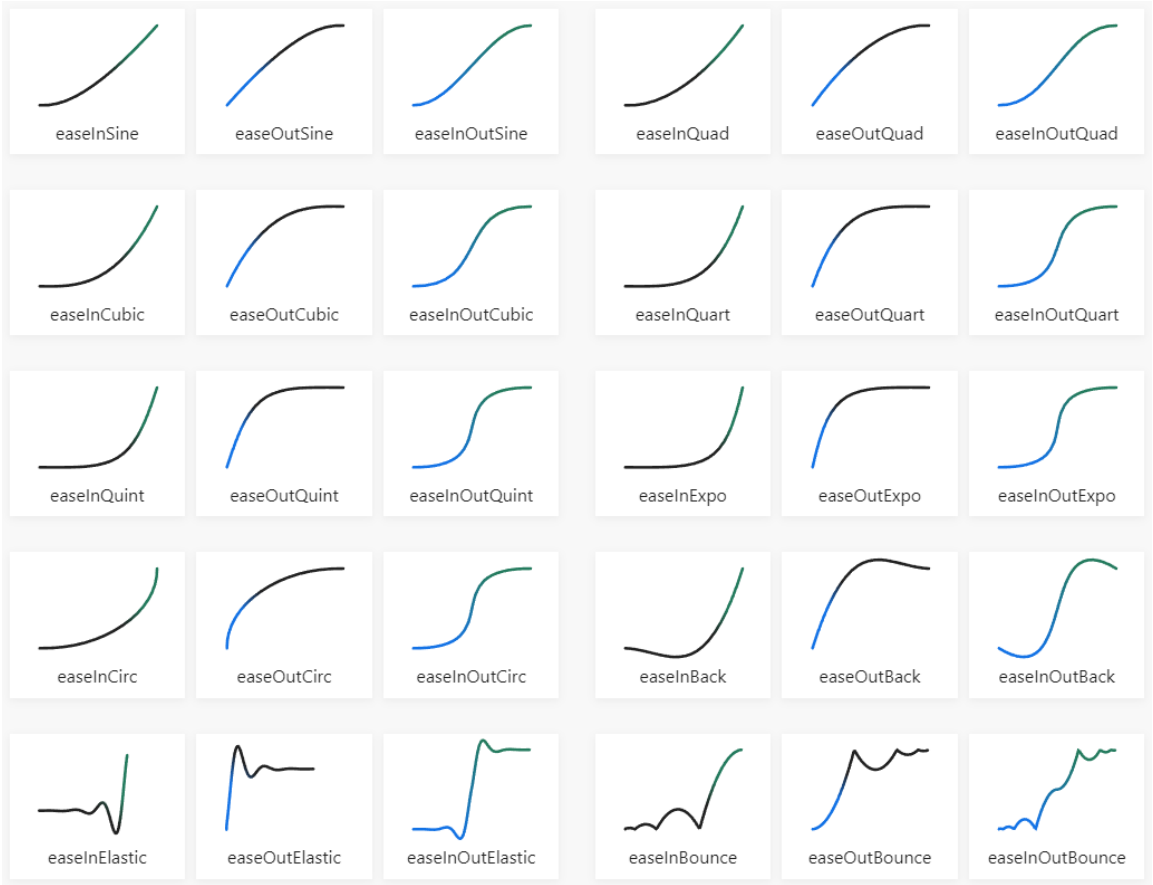

Figura 5.6: *Representação gráfica das principais funções de atenuação usadas por tweens para interpolar valores. Disponível em easings.net, acesso em 19/12/2021.*

- *Git*[3]: Uma ferramenta de versionamento de projeto universalmente conhecida. Ele foi utilizado em conjunto ao *GitLab*[4] para manter o repositório do projeto, que está disponível no Anexo B (Repositório no *Git*).
- *Google Jamboard*[5]: Uma lousa digital que pode ser editada simultaneamente por várias pessoas. Foi usada principalmente durante a fase de *brainstorming*, para decidir e rascunhar mecânicas e implementações.
- Metodologia Ágil[6] e *Feature Driven Development*[7]: Durante o desenvolvimento, foram seguidos princípios e práticas embasadas na metodologia ágil e no FDD, que também se baseia nos métodos ágeis. Em suma, o processo de desenvolvimento se organizou por meio da definição de um modelo inicial de jogo que deveria ser feito e foi dividido em várias funcionalidades atômicas, organizadas em *issues* no *GitLab*. Ambos os

[3] https://git-scm.com/

[4] https://about.gitlab.com/

[5] https://jamboard.google.com/

[6] Manifesto Ágil, acesso em 19/12/2021.

[7] Princípios do FDD disponíveis em *Feature Driven Development (FDD) and Agile Modeling*, acesso em 19/12/2021.

desenvolvedores se reuniam uma ou mais vezes por semana para debater e reavaliar o estado atual do projeto, além de estabelecer metas para o cumprimento das *issues* atuais. Também houve momentos em que foi praticado o *pair programming* em algumas das *issues* mais extensas ou em momentos de necessidade.

Capítulo 6

Criaturas e o jogador

6.1 Classes base

Para facilitar o desenvolvimento, foram implementadas duas classes que serviriam de base para todas as entidades do jogo, herdando do nó *KinematicBody2D*. Optou-se por essa herança pois fazia sentido implementar a física e os movimentos conforme fosse necessário, ao invés de já possuir regras de física a se seguir e ter de adaptá-las durante o desenvolvimento - principalmente dado que as criaturas teriam movimentação baseada em animais aquáticos. Nenhum método originário do *KinematicBody2D* é utilizado nessas classes base, apenas nas classes filhas - mas é importante que elas já herdem desse nó para evitar complicações ou dependências cíclicas, pois usam atributos e métodos do nó pai de *KinematicBody2D*, o *Node2D*.

A primeira classe base é a *Entity* (entidade, em inglês), utilizada para todas as criaturas e para o jogador. Ela basicamente permite que qualquer agente do jogo possa navegar pelas salas e passar pelas *conveyors*, que conectam uma sala a outra. Já a segunda classe, que herda de *Entity*, é a *Creature* (criatura, em inglês). Ela possui um número maior de métodos que serão utilizados pelas criaturas em si: *setters* e *getters* de atributos relevantes para os gerenciadores de entidades, funções abstratas para serem sobrescritas (como por exemplo as funções que recebem sinais e precisam de respostas específicas dependentes da espécie da criatura) e funções relacionadas com a localização na sala. Uma função particularmente importante é a *disable_physics()*, geralmente utilizada após a morte da criatura para garantir que ela não será mais simulada, mesmo quando o quarto onde ela estava pela última vez for carregado. Isso ocorre pois mesmo criaturas mortas continuam sendo acompanhadas pelo gerenciador devido à mecânicas que haviam sido planejadas mas acabaram por não serem implementadas, como a adição de novas criaturas para repopular o ambiente com base nas criaturas que já haviam morrido.

Vale destacar uma variável central às criaturas, o inteiro *health* (saúde em inglês). O valor de *health* indica o quão saudável a criatura está, indicando que ela está morta em valores menores ou iguais a zero. Outro valor chamado *threat* (ameaça, em inglês) havia

sido planejado como um indicador universal de o quão perigosa uma criatura era, porém acabou sendo descartado por ser pouco intuitivo e pouco útil em geral.

6.1.1 Camada física e camada estética

Embora não seja necessariamente algo presente nas classes base, toda criatura do jogo tem uma camada física e uma camada estética. A camada física é uma cena que herda da classe base de criatura, ou seja, *Creature.tscn*, enquanto a camada estética é uma cena com um nó de *Position2D* como raíz, que representa o ponto onde o *sprite*[1] animado da criatura estará ancorado na parte física. Na primeira, estão todos os requisitos e implementações para o funcionamento mecânico da criatura - o código que governa seu comportamento e com o qual outros scripts e módulos se comunicam. Já a segunda é composta principalmente por outros nós *Position2D*, usados para representar pontos articulados da criatura, e nós *Polygon2D*, que desenham polígonos na tela. O nó pai possui um script que rege o comportamento dos polígonos e posições que compõem o *sprite* animado da criatura.

A única referência a essas camadas na classe base é a função *toggle_aesthetics()*, que é usada para ligar e desligar o processamento da camada estética quando a criatura está fora do quarto em que o jogador se encontra.

6.2 Primeira criatura: *Jelly*

O primeiro elemento do jogo a ser desenvolvido foram as *Jellies*. O objetivo dessa entidade é bem simples: ter uma base de criatura o mais simples o possível, que já tivesse funcionalidades básicas, como se mover de forma aleatória e ser vulnerável a outras criaturas. É importante ter uma "entidade controle" que tenha chances muito baixas de gerar *bugs* por si própria pois ela pode ser utilizada para testar comportamentos de outras criaturas e outros objetos de jogo com facilidade, agilizando o desenvolvimento em geral. Também é relevante o fato de que ela pode ser instanciada em números altos sem custo significativo no processamento.

A *Jelly* é governada por uma máquina de estados extremamente simples, implementada usando apenas nós de *Timer* da *Godot*. Ela alterna entre estar parada e se mover em uma direção aleatória, simplesmente rebatendo em qualquer outro objeto ou entidade que possa colidir.

6.2.1 Animações

Inicialmente, a Jelly não possuia nenhuma animação e era representada apenas por um *sprite* estático, pois a prioridade inicial do projeto foi desenvolver os sistemas base do jogo e apenas a animação do jogador e da Barracuda, devido aos seus comportamentos

[1] Em desenvolvimento de jogos, um *sprite* é uma imagem 2D usada para representar algo na tela.

Figura 6.1: *Aparência final da Jelly. Versão animada disponível no Anexo C (Versões animadas).*

serem mais complexos e comunicáveis. Porém, após receber os *Feedbacks* do *playtest*, foi implementada uma animação relativamente simples. A *Jelly* agora possui duas partes que se movem: seus tentáculos e suas bordas (ver Figura 6.1).

As bordas da *Jelly*, ou seja, as partes triangulares no extremos inferiores direito e esquerdo, se movem para cima e para baixo buscando imitar o movimento típico de uma água-viva nadando e são coordenados com os seus movimentos na tela. Já os seus tentáculos utilizam cinemática inversa para se colocarem em posições que simulam o seu arrasto na água conforme a criatura se mexe. Quando ela está parada, os tentáculos procuram uma posição de descanso escolhida semi-aleatoriamente com base na posição de descanso anterior. Os procedimentos utilizados para conseguir esses resultados serão detalhados no Capítulo 7 Técnicas de animação.

6.3 Segunda criatura: *Barracuda*

O segundo elemento a ser implementado foram as Barracudas. O conceito dessa criatura era ser um predador tentando caçar tanto o jogador como outras criaturas, que tivesse noção de como se localizar pelas salas e também estados mais complexos. A Barracuda possui uma máquina de estados mais elaborada, mais próxima do que tinha sido planejado originalmente para as criaturas durante a fase de planejamento (ver Figura 6.2): seu comportamento consiste em passear aleatoriamente pelo mapa e, quando uma possível presa é avistada, é iniciada uma perseguição - entretanto, a Barracuda também pode se assustar e fugir ao ser atacada por um predador, o que impede que ela reaja a presas como de costume.

O funcionamento dessa entidade passou por duas implementações principais. Inicialmente, seu estado *idle* (inativo, em inglês) limitava-se a apenas patrulhar uma certa área horizontal, de forma semelhante ao comportamento padrão de muitos inimigos em jogos de plataforma. Isso implicava que a Barracuda só conseguiria se mover para outras salas quando avistasse alguma presa (incluindo o jogador) e esta decidisse se locomover

para uma sala adjacente. Esse comportamento acabou rejeitado pois, por uma perspectiva estética, isso produzia a ideia de que a Barracuda era uma criatura territorial - rótulo que pertencia ao *Shrimp*, antigamente - e por uma perspectiva mecânica, fazia mais sentido que sua locomoção padrão utilizasse alguma forma de *pathfinding* assim como sua locomoção de perseguição ou fuga. Portanto, a segunda iteração do código da Barracuda dividiu o antigo estado *idle* em dois novos: outro *idle* e o novo *wander* (vagar, em inglês). Em *idle*, a Barracuda somente fica parada, oscilando levemente para cima e para baixo e ocasionalmente movendo sua cabeça para cima e para baixo, procurando presas. Esse estado é interrompido ou quando uma presa é avistada, ou quando ela sofre um ataque, ou quando o seu *Timer* interno acaba. Nesse último caso, a criatura entra no estado *wander*, em que um ponto aleatório da sala atual é escolhido e é definido um caminho até ele usando o nó de *Navigation2D* da sala atual. A Barracuda segue o caminho e retorna ao estado *idle*.

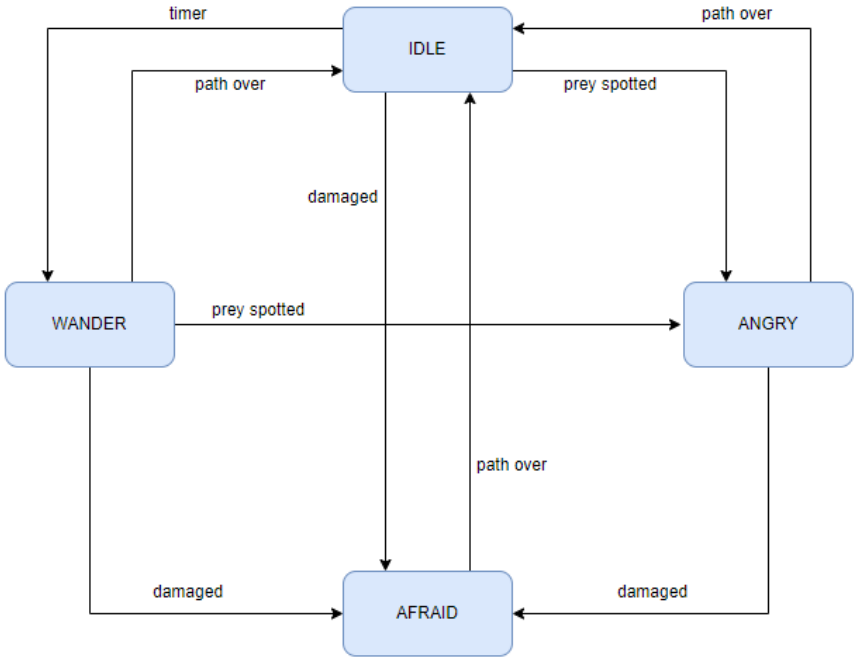

Figura 6.2: *Diagrama de estados da Barracuda. Existem funções à parte que rodam em vários dos estados mostrados, como por exemplo a função check_vision(), que avalia se a Barracuda está vendo alguma presa.*

Os outros dois estados, *angry* e *afraid* (“brava” e “amedrontada”, em inglês), funcionam de maneiras semelhantes ao estado *wander*, mas com alguns passos a mais. Uma *Barracuda* pode entrar no estado *angry* a qualquer momento que ela não esteja *afraid* e passa obrigatoriamente a ter um “alvo” - ou seja, uma referência à presa avistada. Ela então utiliza o *Navigation2D* para traçar um caminho até a posição em que o alvo foi avistado e quando o final do caminho é alcançado, ela observa a área ao redor, buscando a presa e repetindo o processo se necessário. Para que a presa realmente seja mordida, é necessário que ela entre no alcance da *Barracuda*, uma área circular em torno de sua cabeça. Já o

estado *afraid* pode ser iniciado a partir de qualquer outro estado, quando a *Barracuda* é atacada. Caso ela esteja *afraid* e não tenha um caminho definido, ela escolhe um ponto distante[2] de sua posição atual e segue um caminho em direção a ele, em uma velocidade mais alta do que a de perseguição. Uma vez que ela termina sua fuga, ela retorna ao estado *idle*.

A implementação do estado *angry* se baseou no fato de que um comportamento de determinar um alvo e seguí-lo indefinidamente poderia muitas vezes ser injusto com o jogador e gerar uma experiência desagradável - e também criar a possibilidade de uma *Barracuda* rapidamente matar todas as outras criaturas que estivessem na mesma sala que ela. Outra forma considerada era um vetor ou fila de "memória" e expandisse a área de detecção de presas da *Barracuda* apenas para criaturas que estivessem na memória - uma funcionalidade que também poderia ser usada por outras criaturas, conforme fosse conveniente - porém ela foi desconsiderada durante a reavaliação de prioridades do projeto. Outra funcionalidade que acabou ficando fora do escopo final era o conceito de uma criatura que intimidasse as *Barracudas* com sua presença e as colocasse no estado *afraid* instantaneamente, podendo ajudar o jogador a passar por uma área mais perigosa por exemplo.

Um último aspecto notável da *Barracuda* são os valores de personalidade. Cada instância de *Barracuda* tem dois valores de personalidade gerados aleatoriamente para diferenciar seu comportamento de outras de sua espécie, armazenados nas variáveis *restlessness* e *curiosity* ("inquietação" e "curiosidade", em inglês). Esses valores são usados em operações aritméticas simples para determinar quanto tempo de espera cada um dos *Timers* que ativam comportamentos possui, como por exemplo o tempo entre cada passeio pela sala ou o tempo da fuga depois de ser atacada. Eles sempre são determinados usando outros números aleatórios em conjunto, geralmente gerados a partir intervalos que foram definidos empiricamente conforme eram feitos testes internos.

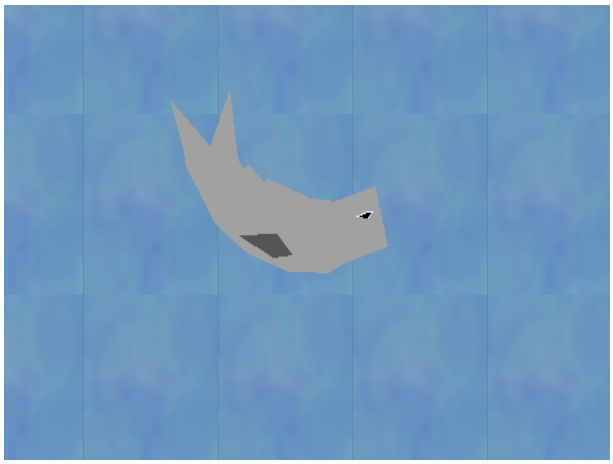

Figura 6.3: *Aparência final da Barracuda. Versão animada disponível no Anexo C (Versões animadas).*

[2] Detalhes sobre o que um ponto distante significa na Seção 8.1 Salas.

6.3.1 Animações

A *Barracuda* possui apenas duas animações, porém elas ilustram um pouco das possibilidades da animação procedural. Quando a *Barracuda* está em movimento, ela ondula seu rabo para criar a ilusão de que está nadando, como esperado (ver Figura 6.3). Porém, a velocidade dessa animação varia proporcionalmente à velocidade de movimento da entidade, usando valores definidos na camada física, que variam de acordo com o seu estado atual. Com isso, como três estados da *Barracuda* envolvem movimentação, mais da metade de seus estados podem ser animados com apenas um algoritmo - e ainda assim possuírem animações diferenciadas. Com uma animação tradicional, simplesmente alterar a velocidade de reprodução dos quadros provavelmente teria resultados estranhos ou visualmente desagradáveis. Já nos momentos em que a *Barracuda* está parada, partes de seu corpo se movem lentamente para cima e para baixo, criando um aspecto de flutuabilidade. Conforme a sua saúde diminui, cada pedaço do corpo tende para baixo com mais força, simulando um esforço maior para se manter flutuando.

Como explicado anteriormente, as animações da *Barracuda* mudam sua intensidade conforme ela é atacada pelo jogador ou por outras criaturas. Enquanto isso implicaria produzir um conjunto de imagens novas caso ela fosse animada tradicionalmente, nesse caso basta adicionar um valor na fórmula que determina quanto ela move seu rabo ao nadar, por exemplo. Todavia, é importante deixar explícito que isso não é um menosprezo ou demérito às animações tradicionais e sim um incentivo para a experimentação de elementos procedurais na animação.

6.4 Jogador e seus controles

Na fase de ideação do jogo o personagem do jogador, que por conveniência será referido apenas como *Player* daqui para a frente, era de uma espécie de salamandra chamada axolote, como pode ser visto em sua arte conceitual no Apêndice A (Rascunhos e artes conceituais). Esse fato originou o nome do repositório do projeto, do inglês *axolotl*, e se deu por ser uma criatura aquática que seria consistente com a ambientação escolhida. Mais adiante a ideia foi descartada em favor do personagem robô atual, pois uma criatura aquática exigiria uma física muito diferente da esperada da maioria dos jogos de plataforma, o que poderia diluir o foco da experiência do jogador quando o objetivo era avaliar as IAs em conjunto com a animação procedural.

Durante a ideação também foram pensados os controles e definidas as ações que o *Player* seria capaz de fazer (ver Figura 6.4), representadas no formato de verbos - uma técnica conhecida em *game design* (Dotsenko, 2017). Foram estabelecidas tanto as ações principais, como andar e pular, quanto as que serviriam como forma de interação do personagem com o ambiente, como agarrar *ledges* (quinas, em português) e pegar, soltar e arremessar objetos. Para aumentar a gama de mecânicas do *Player* também foram incluídas ações que o dariam mais mobilidade, algumas comuns no gênero de jogos de plataforma como pular a partir de paredes e deslizar nelas (do inglês, *wall jump* e *wall slide* respectivamente) (Rodrigues, 2021). Elas permitiram uma maior liberdade para desenhar níveis maiores e mais complexos,

com o intuito de melhorar a experiência de exploração para o jogador.

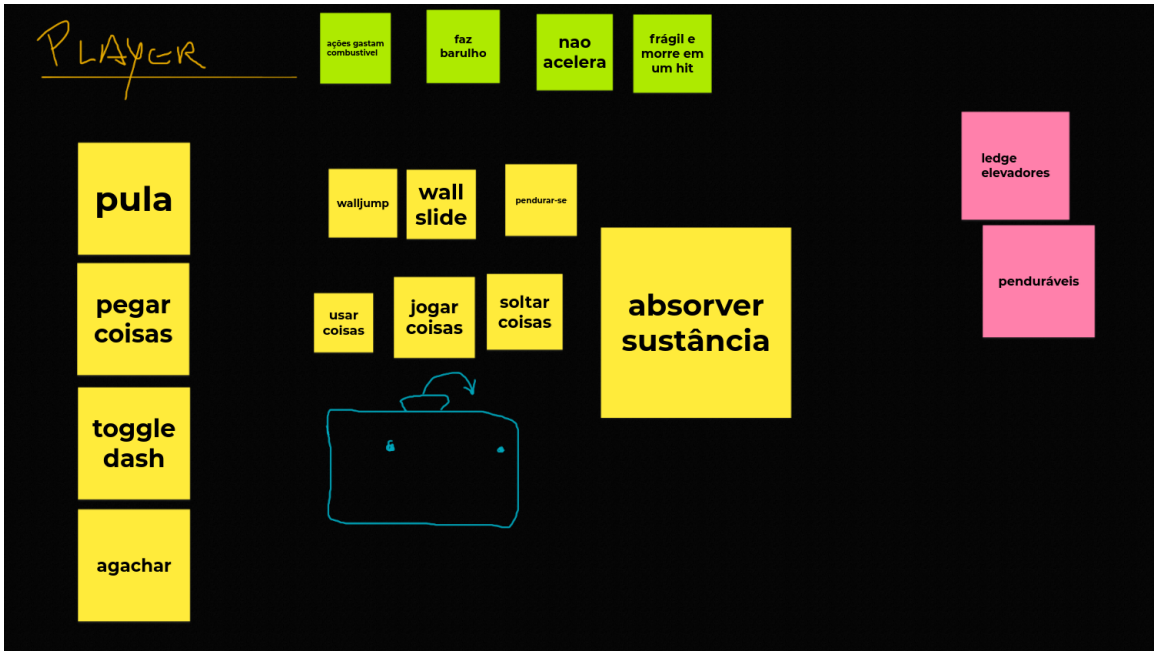

Figura 6.4: *Jamboard de um dos brainstorms, definindo a maioria das ações do Player incluindo algumas que foram cortadas.*

O comportamento da física do *Player* está na cena *PlayerBody.tscn* e seu *script* implementa uma máquina de estados simples (ver Figura 6.5). Esse *script* também é responsável por receber informações da resposta de colisões e de *inputs* do jogador para atualizar variáveis do *Player*, como estado atual, posição, velocidade e aceleração. Também foram definidas algumas constantes para valores de velocidade e aceleração do *Player* em diversos estados, de forma a tornar esses valores fáceis de serem modificados e permitir várias iterações de testes até que o controle do movimento fosse prazeroso ao jogador.

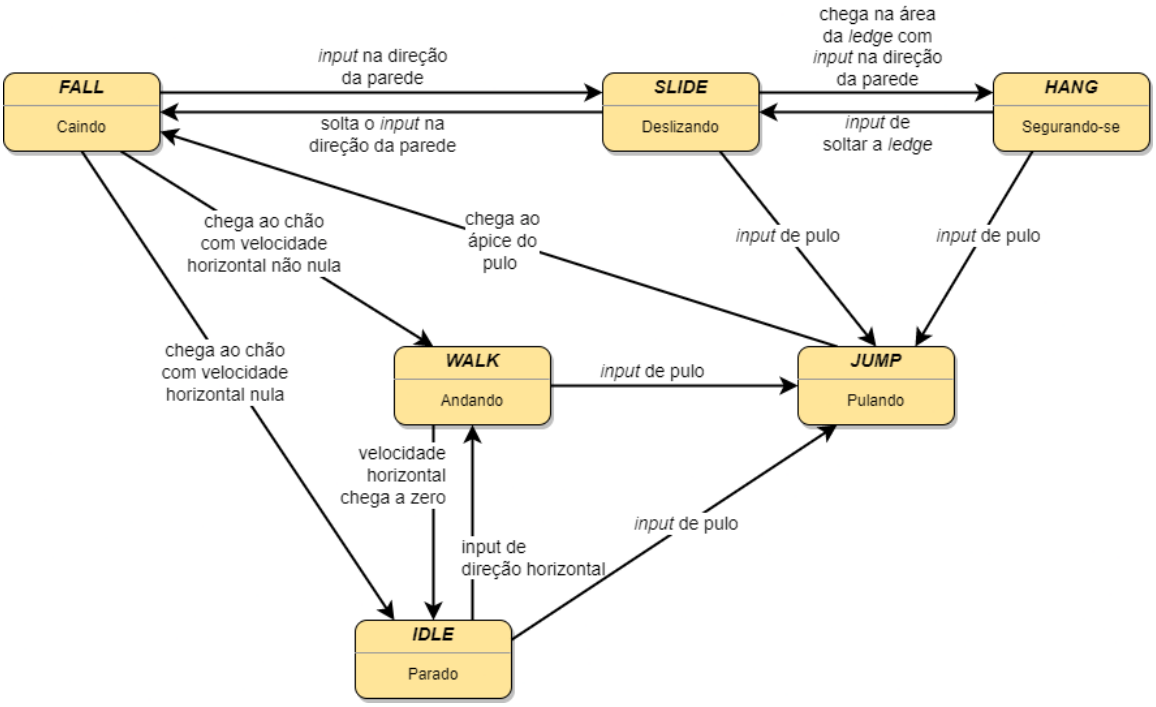

Figura 6.5: *Diagrama de transição de estados do Player.*

Capítulo 7

Técnicas de animação

Durante o desenvolvimento do projeto, várias técnicas foram utilizadas para animar proceduralmente as entidades do jogo. Esta seção detalha os algoritmos e cenas desenvolvidos para atingir os resultados visuais desejados (ver Figura 7.1).

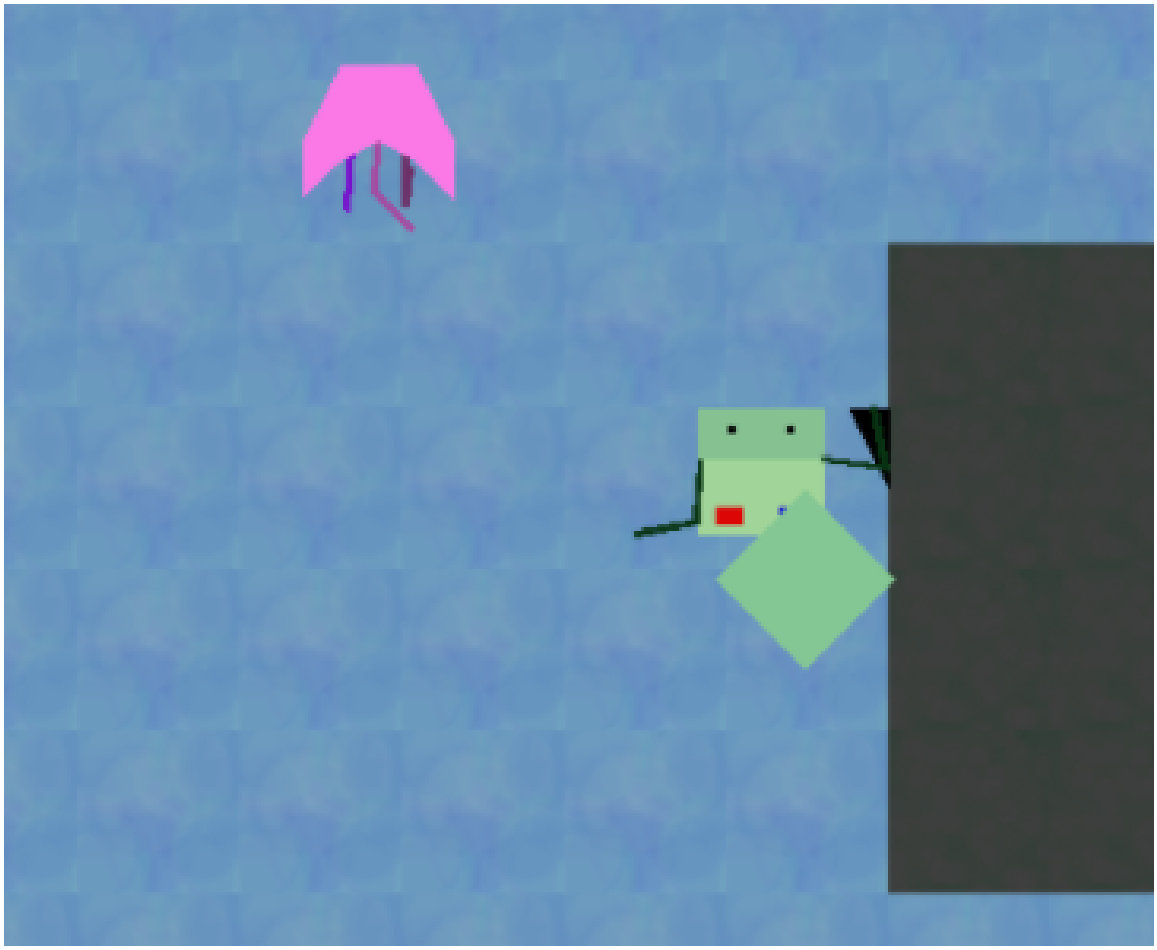

Figura 7.1: *Player segurando-se em uma quina. Seu corpo é animado com base em cinemática frontal, enquanto seus braços são animados com base em cinemática inversa. Versão animada disponível no Anexo C (Versões animadas).*

7.1 *Forward Kinematics*

Como já foi introduzido na Seção 3.1 *Forward & Inverse Kinematics*, o processo de FK é utilizado quando se tem a posição e rotação da primeira junta de um operador (um braço ou cauda, no contexto das criaturas implementadas) e, utilizando equações de cinemática, é calculada a posição do efetor final, ou seja, a última junta do operador. As equações de cinemática frontal foram adaptadas para a animação da *Barracuda*, para que movendo a primeira seção de seu corpo, imediatamente após sua cabeça, o resto do corpo se movesse de acordo. O algoritmo adaptado considera apenas a rotação de cada segmento (ou junta), pois, devido à forma como a árvore de nós é construída na *Godot*, a posição de um nó filho já é baseada na posição de seu pai. Desse modo, construindo a hierarquia de nós corretamente para a camada estética da *Barracuda*, foi possível simplificar o algoritmo de FK.

Programa 7.1 Implementação de FK na camada visual da *Barracuda*.

```
1   for i in range(joints.size()):
2      var joint = joints[i]
3      var r = delta*THETA*speed*wiggle_direction(joint.rotation)*health
4         if i == 0:
5            # first one gets info from parent
6            joint.rotation += rotation + r
7         else:
8            # any other one gets info from the one before
9            var prev_joint = joints[i-1]
10           joint.rotation = prev_joint.rotation + r
```

O trecho de código acima é parte do *process()* da camada estética da *Barracuda*. Nele, o valor de *r* é determinado usando valores da camada física (como *health* e *speed*), constantes empíricas como *THETA* e o valor retornado pela função *wiggle_direction()* ("direção de meneio" em inglês), que decide em que ponto do movimento o *sprite* está no momento e ajusta sua direção de acordo. Então, ele é somado com a rotação do segmento anterior, para cada segmento que não é o primeiro, atingindo assim a rotação desejada para cada um.

7.2 *Inverse Kinematics*

O processo de IK, também introduzido anteriormente na Seção 3.1 *Forward & Inverse Kinematics*, é utilizado para determinar a posição de cada junta do operador à partir da posição final desejada - e é um problema significativamente mais complicado que aquele apresentado pela cinemática frontal. A cinemática inversa normalmente precisa de derivações para encontrar as posições e rotações desejadas, então existem diversos métodos iterativos para implementá-la. Além disso, assim como na FK, a estrutura de árvore dos nós e cenas precisa ser considerada para que os algoritmos tradicionais de IK sejam adaptados corretamente. No projeto, tanto os tentáculos da *Jelly* como os braços do personagem do jogador utilizam cinemática inversa.

A primeira tentativa de implementação de IK foi utilizando um método baseado na lei dos cossenos, feita diretamente no *process()* da camada estética das entidades. Em um operador com dois segmentos[1], é possível traçar um triangulo usando o tamanho de cada segmento e a distância até o ponto desejado, permitindo o cálculo dos ângulos necessários[2]. Entretanto, essa implementação produziu resultados não desejados, como oscilações entre posições e dificuldades em encaixar com os padrões de desenvolvimento da *Godot*, e portanto ela foi abandonada.

A implementação definitiva de IK utilizada foi uma adaptação do algoritmo FABRIK, descrito por Andreas Aristidou e Joan Lasenby em ***FABRIK: A fast, iterative solver for the Inverse Kinematics problem*** (Aristidou e Lasenby, 2011). O algoritmo foi escrito em um *script* à parte, para que pudesse ser carregado e utilizado por qualquer cena do projeto. Entretanto, como a adaptação para *GDScript* segue bastante à risca o código original, foram necessárias algumas convenções para as cenas que fossem utilizá-la para animar seus *sprites*. Essa abordagem foi mais bem sucedida do que a anterior, pois por mais que ter de organizar a cena fora do padrão de árvore seja contra-intuitivo, ela poupou a necessidade de fazer grandes mudanças no algoritmo original e garantiu o seu funcionamento sem maiores problemas. No Programa 7.2, temos sua implementação como presente no arquivo *Fabrik.gd* do projeto.

Nessa implementação, o algoritmo trabalha usando apenas *floats* e coordenadas cartesianas, sem operar com nenhum nó da *Godot*. Isso ocorre pois ele sempre recebe um vetor de pares de coordenadas que são extraídas de nós *Position2D* e depois retornadas para redefinir a posição global de cada um deles. Isso supõe que todos eles estejam usando o mesmo referencial para determinar sua posição local, implicando que todos sejam filhos diretamente de um mesmo nó - e nesse ponto contradizendo a intuição de organizar os nós em forma de árvore.

O algoritmo FABRIK apenas encontra os pontos desejados, então ainda é necessário implementar a animação propriamente dita. Para isso, cada membro animado possui uma posição alvo que ele deseja atingir e é atualizada a cada quadro. Conforme essa posição alvo muda, o FABRIK é rodado para cada ponto intermediário durante o processo, atualizando a posição global de cada nó *Position2D* em questão. Em seguida, é desenhada uma *Line2D* usando a posição local de cada uma das *Position2D* atualizadas, concluindo assim a animação.

7.3 *FillPolygon2D*

O último elemento central das animações elaboradas é o *FillPolygon2D* (“preencher polígono” em inglês), uma classe construída a partir do nó *Polygon2D* da *Godot*. O

[1] É possível também generalizar esse método para operadores com mais do que dois segmentos, como mostrado em vídeo em *How to make a Limbo-style spider in Godot using procedural animation*

[2] Pode-se encontrar uma explicação bastante compreensiva do tema em *Inverse Kinematics in 2D*

Programa 7.2 Implementação de FABRIK em *GDScript*.

```
1   const TOL := 1.0
2
3   func fabrik(joints: Array, links: Array, target: Vector2) -> PoolVector2Array:

4       # This algorithm is a GDScript recreation of Aristidou and Lasenby's
5       # FABRIK algorithm, an iterative IK solver
6
7       var n := joints.size()
8       var dist_target = target.distance_to(joints[0])
9       var total_link_length := float_array_sum(links)
10
11      if dist_target > total_link_length:
12          # oops can't reach that
13          for i in range(0, n-1):
14              var p = joints[i]
15              var d = links[i]
16              var r = target.distance_to(p)
17              var lambda = d/r
18              joints[i+1] = (1 - lambda) * p + lambda * target
19      else:
20          # yay I can reach this
21          var b = joints[0]
22          var dif = target.distance_to(joints[n-1])
23
24          while dif > TOL:
25              joints[n-1] = target
26
27              for i in range(n-2, -1, -1): # forward reaching
28                  var pii = joints[i+1]
29                  var pi = joints[i]
30                  var d = links[i]
31                  var r = pii.distance_to(pi)
32                  var lambda = d/r
33                  joints[i] = (1 - lambda) * pii + lambda * pi
34
35              joints[0] = b # assure the root stays the same
36
37              for i in range(0, n-1): # backward reaching
38                  var pii = joints[i+1]
39                  var pi = joints[i]
40                  var d = links[i]
41                  var r = pii.distance_to(pi)
42                  var lambda = d/r
43                  joints[i+1] = (1 - lambda) * pi + lambda * pii
44
45              dif = target.distance_to(joints[n-1])
46
47      return PoolVector2Array(joints)
```

FillPolygon2D é simplesmente um polígono desenhado na tela, assim como a classe da qual herda seus atributos, porém cada um de seus vértices tem sua posição definida em tempo real. Por mais que seja uma implementação bastante simples, intercalar *FillPolygons* com os *Polygons* normais da *engine* permite conectar todas as partes que se movem da camada estética de uma criatura, de modo que ela realmente pareça apenas um corpo em movimento.

Para funcionar, cada *FillPolygon2D* tem seus vértices atrelados a nós *Position2D* no editor de cenas. Então, cada vértice é reposicionado em tempo de execução conforme as *Position2D* são movidas pela tela, animando o polígono. É importante garantir que os pontos do vértices sempre formem um polígono, porém essa responsabilidade é da classe que utiliza uma instância de *FillPolygon2D*. A seguir temos o código da classe, no qual *neighbor_vertices* são os pares de coordenadas que serão transformados nos vértices do polígono a ser preenchido, guardados no vetor *vertexes*.

Programa 7.3 Código da classe *FillPolygon2D*.

```
1   extends Polygon2D
2   class_name FillPolygon2D
3
4   export (Array, NodePath) var neighbor_vertexes
5   var vertexes := []
6
7   func _ready():
8      for i in neighbor_vertexes.size():
9         vertexes.append(get_node(neighbor_vertexes[i]))
10
11
12  func _process(_delta):
13     var updated_vertexes = PoolVector2Array()
14     for i in vertexes.size():
15        var vertex = to_local(vertexes[i].global_position)
16        updated_vertexes.append(vertex)
17     self.polygon = updated_vertexes
```

Capítulo 8

Gerenciadores e outras funcionalidades

8.1 Salas

Salas ou quartos, chamadas de *rooms* no código, são o cenário por onde os personagens do jogo se locomovem. Como já mencionado anteriormente, cada sala foi projetada de modo a balancear um típico modelo de nível de jogo de plataforma com um ambiente que permitisse testes dos comportamentos das criaturas (ver Figura 8.1). Elas são compostas por superfícies sólidas, superfícies semi-sólidas - isto é, plataformas que o jogador pode optar por atravessar ou não - e *ledges* que permitem que o jogador se movimente de forma vertical com mais facilidade. As entradas e saídas de uma sala são dadas por *conveyors*, como introduzido no Capítulo 4 Planejamento de projeto, e serão explicadas com mais detalhes a seguir.

Em relação à sua implementação, todas as salas herdam de uma classe mãe *Room.tscn*, que contém principalmente métodos relevantes para os algoritmos de navegação das criaturas. Uma cena de sala tem um *Node2D* como raiz, porém ela depende de sempre ter um *tilemap* e um nó *Navigation2D*. Além disso, cada sala possui indicações dos pontos máximos que a câmera do jogo pode atingir enquanto o jogador está nela.

É importante detalhar duas funções da classe base de sala: *get_random_point()* e *get_distant_point()*. A primeira escolhe um ponto aleatório dentre qualquer ponto que uma criatura possa se deslocar para e o retorna, usando as informações do *tilemap* para decidir as posições válidas. Já a segunda recebe uma posição e um valor e retorna um ponto que esteja pelo menos a esse valor de distância da posição dada. Para tanto, ela embaralha um vetor com todas as posições válidas e retorna a primeira que satisfizer a distância mínima.

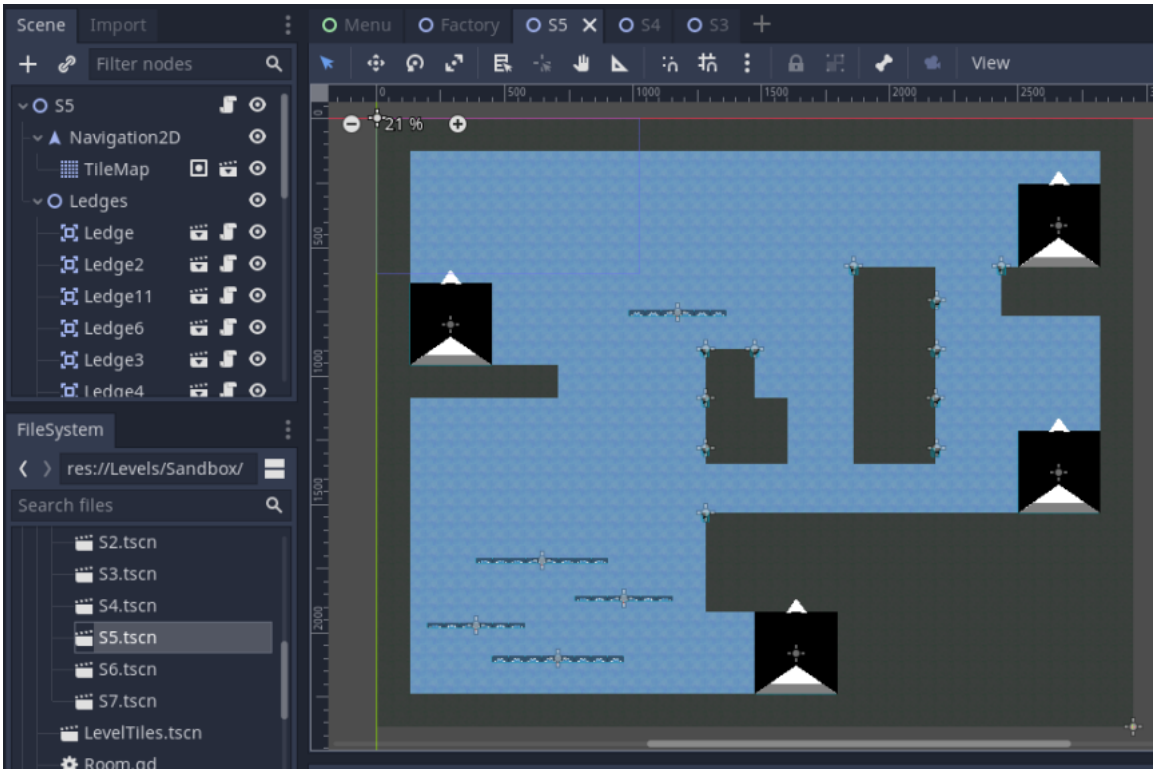

Figura 8.1: *Cena de uma sala no editor de cenas da Godot, mostrando várias plataformas e paredes, além de espaços abertos para serem ocupados por criaturas aleatoriamente.*

8.2 *Ledges* e Plataformas

Ledges foram pensadas para aumentar a mobilidade vertical do personagem do jogador, já que desde o começo foi decidido não implementar um pulo duplo - uma das mecânicas famosas de jogos de plataforma (Rodrigues, 2021). Sua implementação é simples, tratando-se apenas de uma *Area2D* com forma retangular que permite à cena *PlayerBody* saber se está na área em que pode se segurar, e então permitindo a ele um pulo vertical diferente do pulo mais horizontal que é possível ao se deslizar em uma parede.

As plataformas são outro elemento elaborado para ampliar as opções no *level design* e aprimorar a experiência de exploração do jogador. São cenas que herdam de *StaticBody2D* e têm uma forma de colisão retangular do tipo *CollisionShape2D*, que incluem a funcionalidade de colisão unidirecional e fazem com que o personagem do jogador só colida com elas ao vir de cima. Além disso, existe a lógica no *PlayerBody* para descer de tais plataformas caso o personagem esteja no topo delas e o jogador pressionar a seta para baixo.

8.3 *Conveyors*

As *conveyors* - ou esteiras transportadoras - são o único método de tanto o jogador como as criaturas se locomoverem entre salas diferentes. Elas passaram por duas implementações durante o desenvolvimento, sendo que a sua mudança foi consequência

direta dos *Feedbacks* do *playtest*. O seu funcionamento foi bastante debatido desde o início do projeto, pois o principal objetivo delas era uma transição entre salas que fosse intuitiva para o jogador e também comunicasse bem a passagem das criaturas. Por mais que a versão final não represente isso com fidelidade, ela produz uma experiência de jogo menos problemática.

Uma *conveyor* é, na verdade, um par de *conveyors*. Cada uma recebe um inteiro como identificador ao ser colocada em uma sala, sendo que sua outra metade recebe o inteiro oposto. Dessa forma, cada *conveyor* sabe para onde ela deve enviar as entidades que recebe e, caso uma esteira não encontre sua respectiva metade, isso significa que ela deve abstrair as entidades que receber para a tabela de criaturas[1], que gerencia as entidades fora das salas adjacentes ao jogador. A cada par de esteiras foi dado o nome de "irmãs".

A primeira implementação das *conveyors* era baseada apenas na detecção de uma entidade em sua área. Cada *conveyor* possuía dois nós do tipo *Area2D*: o primeiro determinava a área em que a entidade era considerada "dentro" da esteira e, enquanto houvesse entidades nela, não era possível que criaturas entrassem pela *conveyor* oposta. Uma vez que uma entidade estivesse dentro da primeira área, ela podia avançar para a segunda, que a transportaria para a segunda área de sua irmã. Por mais que esse funcionamento permitisse que o jogador soubesse quando era seguro ou não entrar em uma *conveyor*, ela frequentemente criava *deadlocks* devido a criaturas que decidiam se mover ao redor da *conveyor* por muito tempo e acabou sendo descartada.

A implementação definitiva das *conveyors* utiliza apenas uma *Area2D*. Assim como as suas predecessoras, ela mantém controle das entidades que estão dentro dela, mas não impede a passagem entre as esteiras caso esteja ocupada. Uma vez dentro dessa área, tanto o jogador como as criaturas tem a opção de interagir com a *conveyor* usando a função *ask_for_warp()*, presente no código da classe mãe das entidades - no caso, o jogador pode fazer isso usando um *input* e as criaturas possuem comportamentos programados para quando entram na área em questão. Com essa implementação, o único momento em que a *conveyor* não permite a passagem de uma entidade é imediatamente após uma chamada de *ask_for_warp()*, indicada pela mudança da cor do *sprite* da esteira. Por mais que essa implementação não seja ideal, ela agiliza o jogo e permite que o jogador explore com mais facilidade.

8.4 Objetos

Como desviou-se de um foco em combate para realçar o aspecto de exploração no jogo, foi decidido que o personagem do jogador não teria nenhum tipo de ataque próprio. Para que o jogador não se sentisse completamente desamparado ao se deparar com as criaturas hostis, foram introduzidos objetos que o ajudariam nesses momentos. Além de se tornarem um recurso finito que o jogador deve pensar usar com estratégia eles destacam

[1] Detalhes na Seção 8.5 Tabelas de gerenciamento.

a necessidade de exploração, já que para combater as criaturas o jogador deve primeiro achá-los.

A cena *PickableObject.tscn* providencia uma classe abstrata base para a implementação dos objetos, implementando as funcionalidades deles serem pegos, soltos e jogados, além de uma resposta de colisão com criaturas. Tal cena herda de *RigidBody2D* e se aproveita de sua funcionalidade de mudar de modo de simulação, agindo em dois modos diferentes dependendo da situação. Quando o personagem do jogador pega um objeto, sua colisão deixa de estar ativa e seu modo passa para *MODE_KINEMATIC*, permitindo-o seguir a posição do jogador sem ser afetado pela simulação física. Quando livre no ambiente seu modo é *MODE_RIGID* e portanto fica habilitado a reagir à simulação física e respostas de colisão. Ao serem jogados ou soltos, além de passarem para o modo *MODE_RIGID*, é aplicada uma velocidade numa direção que depende do personagem do jogador.

Dois objetos que herdam de *PickableObject* foram implementados. As bolas (em *Ball.tscn*) e lanças (em *Spear.tscn*) tem comportamentos similares, porém as lanças foram planejadas com um propósito mais belicoso e portanto dão mais dano a criaturas e podem ser arremessadas mais longe.

8.5 Tabelas de gerenciamento

As tabelas de gerenciamento são responsáveis por tudo que acontece fora do campo de visão do jogador. Elas controlam todas as criaturas que não estão sendo simuladas no momento e também mantêm informações relevantes sobre salas que estão e não estão instanciadas. Ambas as tabelas são inicializadas como *AutoLoads* e portanto sempre estão ativas durante a execução do jogo. As tabelas se baseiam em um modelo CRUD básico[2], porém possuem várias funcionalidades além dos esperados *Create*, *Read*, *Update* e *Delete*. Ambas as tabelas são compostas principalmente por um grande dicionário, que é carregado à partir de um arquivo *json*, e possuem funções para imprimir no console seus elementos atuais. Ademais, as tabelas geralmente dependem uma da outra e operam em sincronia, mas foram implementadas em arquivos separados para que o código fosse mais modular e facilitasse a depuração.

8.5.1 Tabela de Criaturas

A tabela de criaturas armazena cada criatura como um par chave-valor indexado pelo identificador da criatura, ou seja, um inteiro definido previamente pelo arquivo *json* lido pelo jogo. Associado ao identificador, está a sala atual da criatura e uma instância da cena de sua espécie, que por sua vez possui quaisquer informações adicionais necessárias. Algumas das operações da tabela são bastante simples, como por exemplo retornar todas as criaturas em uma certa sala ou todas as criaturas de uma certa espécie, mas a tabela também tem a capacidade de alterar variáveis de instância de uma criatura, como seu

[2] Mais informações sobre CRUD disponíveis em Wikipedia - Create, Read, Update and Delete.

valor de *health* e verificar se ela está viva.

Todas as outras funções da tabela de criaturas estão diretamente relacionadas com a simulação das criaturas que não estão atualmente na árvore de nós, ou seja, que não estão em nenhuma sala carregada no momento - e suas instâncias estão somente associadas à tabela em si. Cada criatura possui uma lista com a probabilidade de ela tomar um certo tipo de ação ou não agir, sendo que caso a criatura decida atacar, é necessário selecionar outra criatura válida de sua sala e atualizá-la de acordo. A ação mais comum para ambas as espécies de criaturas é se mover - e sua implementação é relativamente simples, apenas atualizando a entrada da criatura na tabela e emitindo um sinal para outros nós relevantes, como será explicado na seção seguinte.

8.5.2 Tabela de Salas

Diferente da tabela de criaturas, a tabela de salas não guarda instâncias de cada sala, apenas informações relevantes sobre elas que serão utilizadas por outros nós e cenas, além de uma referência para o *resource* da sala, que será instanciado por outras cenas quando necessário. Além de realizar operações CRUD em qualquer entrada da tabela de salas, também é possível requisitar algumas propriedades mais específicas, como por exemplo, dada uma *conveyor* e a sala atual, encontrar a sala vizinha de destino dessa *conveyor*.

As funções centrais da tabela são aquelas relacionadas com o deslocamento do jogador. A tabela de salas é responsável por gerenciar, conforme o jogador se move, que salas devem ou não estar carregadas - e também qual das salas carregadas atualmente contém o jogador e portanto deve simular a camada estética das criaturas nela. O algoritmo que gerencia essas questões é relativamente trivial, além de possuir um código simples que se limita a fazer poucas operações e enviar sinais para que as outras cenas e nós envolvidos no processo executem suas sub-rotinas de acordo.

Quando o jogador é inicializado, a sala inicial é carregada e sua camada estética é ligada, enquanto as salas vizinhas são apenas carregadas. Toda vez que o jogador troca de sala, os vizinhos na nova sala são carregados e os vizinhos da sala anterior, exceto a nova sala atual, são descarregados. Além disso, a câmera de jogo é centrada na nova posição do jogador e as camadas estéticas são ligadas e desligadas de acordo. Esse gerenciamento de quartos é eficiente e efetivo, pois garante que o jogador sempre conseguirá se locomover para uma sala vizinha sem esperar seu carregamento - e ele também não percebe todos os ajustes que ocorrem fora das câmeras do jogo, fazendo parecer com que todas as salas sempre estão prontas para recebê-lo.

8.6 *Factory*

A cena *Factory.tscn* é a culminação de todas as etapas construídas até então e serve como cena principal a ser executada pelo jogo. A *Factory* (fábrica, em português) é um *Node2D* com cinco outros nós do mesmo tipo instanciados como filhos, chamados de *slots*

(leia-se “entradas”, em português), que são utilizados para instanciar cada uma das salas carregadas. Para as salas planejadas para o *sandbox* o máximo de salas carregadas ao mesmo tempo sempre seria cinco, mas a expansão do número de *slots* seria extremamente simples, mudando apenas um par de variáveis. Graças à inescapável natureza iterativa do desenvolvimento de software, a *Factory* também passou por mais de uma implementação, porém o uso de *slots*, o nó do jogador e o nó da câmera de jogo se mantiveram basicamente inalterados durante as iterações.

A primeira implementação da *Factory* era bastante mais complexa do que o necessário (ver Figura 8.2). Nela, quando uma criatura era adicionada na árvore por estar em alguma das salas carregadas, ela era inserida como filha da sala. Isso implicava que, a cada troca de sala que uma criatura realizava, era necessário modificar o pai de sua instância. Essas manipulações na árvore de cenas eram excessivamente trabalhosas devido ao fato de que a maior parte dos métodos das criaturas não dependia de uma relação de pai com uma sala, apenas de dados sobre sua sala atual.

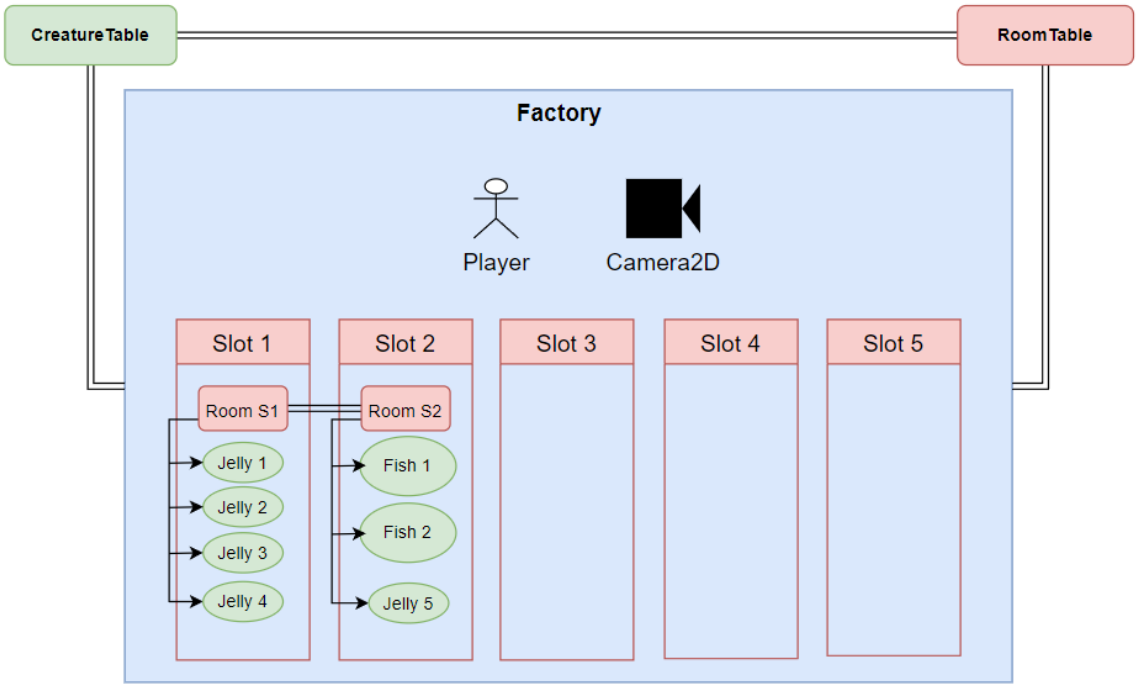

Figura 8.2: *Diagrama mostrando a organização original da Factory. Ao invés de facilitar a estruturação do código, as relações na árvore de cenas eram complicadas e precisavam ser mudadas com frequência, algo que gerava muitos bugs e problemas de sincronia de operações.*

As posições utilizadas nos algoritmos das criaturas eram globais e poderiam ser até calculadas com mais facilidade usando apenas o referencial do *Node2D* da *Factory*, além do fato de que ambas as tabelas de salas e de criaturas já mantinham todas as informações necessárias constantemente atualizadas, independente da árvore de cenas. Logo, a implementação final refatorou o código da *Factory* e parte do código das criaturas para que elas fossem apenas filhas do próprio nó da *Factory* (ver Figura 8.3), poupando o código

de complexidades desnecessárias.

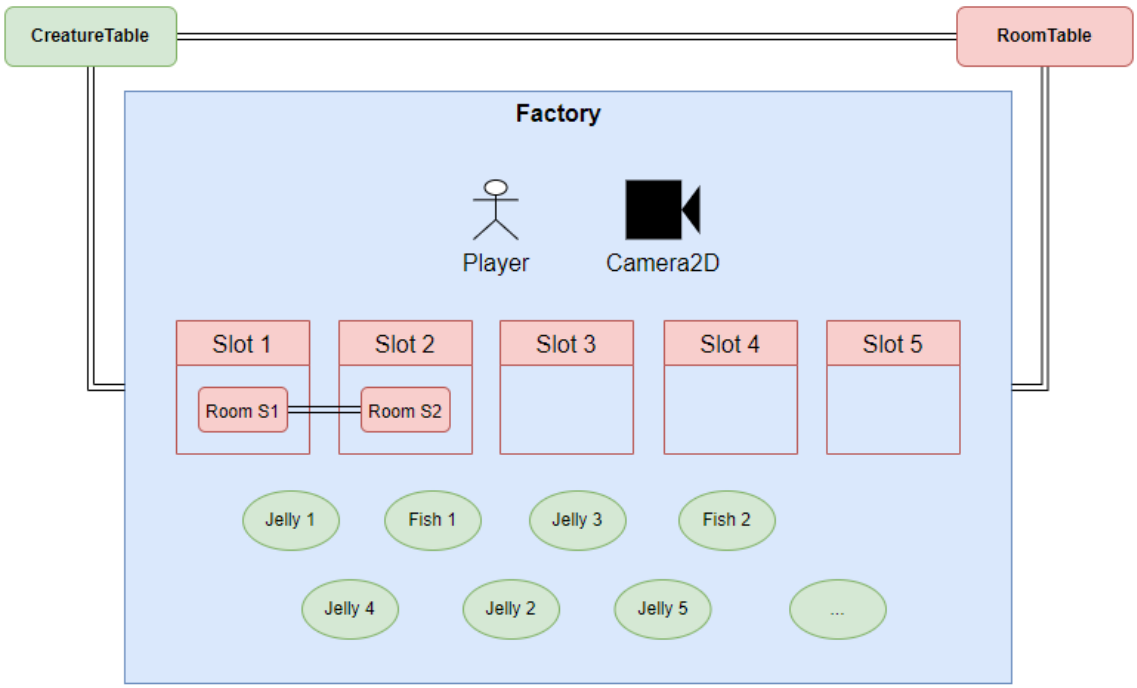

Figura 8.3: *Diagrama mostrando a organização atual da Factory. Essa implementação é mais prática e mais eficiente, pois depende de menos operações na árvore de cenas e é mais intuitiva de se trabalhar com. As conexões antes redundantes entre criaturas e salas agora são feitas apenas nas tabelas.*

Como dito antes, a *Factory* é responsável por reunir todas as funcionalidades das salas, gerenciadores e criaturas. Um exemplo bem simples disso é a função de inicializar o jogador da tabela de salas, que emite um sinal para a *Factory* instanciar o nó do jogador na sala correta. O algoritmo de alocação dos *slots* também é bem trivial, simplesmente alocando o primeiro *slot* disponível para a sala a ser carregada e desalocando o *slot* de uma sala quando ela deixa de estar carregada. Outra função essencial para o jogo contida na *Factory* é o *Timer* global que simula uma ação aleatória para cada criatura que não esteja em quartos carregados periodicamente.

Além disso, é a *Factory* que deve inserir as instâncias de criaturas nas salas apropriadas, usando a função *spawn_creature()*. Existem duas possibilidades para inserir uma criatura em uma sala: ou a sala foi carregada e as criaturas que estavam nela segundo a tabela devem - logicamente - aparecer na sala, ou uma criatura entrou na sala por alguma *conveyor*. No primeiro caso, basta simplesmente escolher aleatoriamente uma posição em que a criatura poderia estar e inserir sua instância nela. Já no segundo caso, é necessário usar as funcionalidades de ambas as tabelas gerenciadoras para localizar a *conveyor* correta e então verificar se ela não está sendo utilizada, sequencialmente inserindo a instância de criatura na posição adequada. Caso a *conveyor* esteja ocupada, a criatura se mantém na sala de origem e a tabela é atualizada de acordo.

Finalmente, a cena da *Factory* também trata alguns *inputs* do jogador - mais especificamente, aqueles usados para controlar funções que auxiliam os testes do jogo. Funcionali-

dades de controlar a câmera e separá-la do jogador (ver Figura 8.4), imprimir dados dos gerenciadores no console e pausar a simulação de criaturas não carregadas, por exemplo, são ações que facilitam e agilizam o processo de testes e depuração.

Figura 8.4: *Vista da cena de Factory com a câmera de testes, permitindo acompanhar várias salas ao mesmo tempo. O jogador se encontra na segunda sala e ela seria a única contemplada por seu campo de visão.*

Capítulo 9

Playtesting e últimos ajustes

9.1 Lançamento do *playtest*

Desde o início do trabalho, havia o objetivo de lançar uma versão de testes do jogo para que houvesse *feedback* externo, permitindo melhorias e opiniões novas no desenvolvimento. Devido a complicações internas, a *build* de *playtest* foi lançada depois do estabelecido pelo cronograma e com alguns *bugs* conhecidos, mas mesmo assim a opinião de jogadores externos ajudou a melhorar o projeto e determinar as prioridades da etapa final de desenvolvimento.

Junto com a primeira *build*, foi lançado um questionário utilizando o *Google Forms*, informando as intenções do projeto, informações sobre a versão de *playtest* e algumas perguntas elaboradas para avaliar os resultados do projeto até então, sendo que três delas pediam uma resposta escrita e quatro delas pediam uma avaliação de um aspecto do jogo usando valores de um a cinco, sendo um "discordo completamente" e cinco "concordo completamente". As respostas obtidas estão disponíveis no Apêndice B (Dados de *playtest*).

9.2 *Feedbacks* do *playtest*

Infelizmente, o questionário não teve muito alcance, porém mesmo assim as repostas obtidas foram úteis e esclarecedoras. Analisando-as, ficaram claras as seguintes demandas:

- A falta de animação das *Jellies* as fazia ser exatamente o oposto do desejado: apenas um pedaço do cenário. Somente seu comportamento aleatório obviamente não era suficiente para gerar uma interação positiva.
- O funcionamento das *conveyors* era problemático e gerava *deadlocks*. Por mais que fosse um problema conhecido, ele impactou bastante a experiência dos jogadores.
- O *pathfinding* da barracuda possuía problemas maiores do que os esperados. Por mais que já se soubesse que alguns caminhos eram produzidos de forma equivocada

devido às colisões da criatura, isso acontecia com frequência altíssima e não permitia avaliar o comportamento das criaturas como da forma desejada. A maioria das respostas sobre este comportamento concordava que ele não fazia sentido.

Além disso, convém detalhar um pouco dos aspectos positivos observados nos testes:

- Tanto a movimentação quanto as animações do jogador foram bastante satisfatórias. Os controles do personagem eram responsivos e suas ações abriam uma gama interessante de possibilidades para serem expandidas além do playtest. Não somente isso, as suas animações geraram agrado e eram representativas em relação às suas ações.
- Mesmo com ações limitadas, interagir com as criaturas deixou algumas impressões positivas, como por exemplo o fato de que a Barracuda se movia com mais velocidade ao avistar o jogador.

Com base nos dados recolhidos e debates com o supervisor do projeto, foram determinadas as prioridades para a etapa final de desenvolvimento. Ao invés de tentar desenvolver novas criaturas e expandir o *sandbox* construído até então, o foco seria em polir as criaturas já existentes e consertar os problemas encontrados no *playtest*.

9.3 Ajustes finais

Nesta seção, serão detalhadas as últimas modificações feitas no projeto. Entretanto, vale notar que várias delas já foram explicadas com detalhe nas seções anteriores, em função de uma leitura mais fluida do texto. São elas: a implementação das animações da *Jelly*, a mudança de comportamento das *conveyors*, a refatoração da estrutura interna da *Factory* e a criação da classe base *Entity*.

Um dos problemas centrais era o *pathfinding* da Barracuda, que frequentemente prendia a criatura em paredes e impedia sua movimentação. Isso se devia a dois fatores principais, que foram descobertos posteriormente por meio de testes internos: a otimização de caminhos do nó de *pathfinding* da *Godot* e um erro de conversão de coordenadas locais para globais.

Para resolver a primeira questão, foi adicionada uma nova *tile* no *TileSet* utilizado para a construção das salas, que foi utilizada para contornar a área ao redor de todas as paredes. Ela permite a passagem de qualquer entidade, mas não é reconhecida no algoritmo de *pathfinding*. Dessa forma, quando uma *Barracuda* está buscando um caminho ótimo, ela não corre o risco de colidir com uma parede, mesmo quando realizando curvas fechadas, por exemplo. Já a segunda questão foi resolvida com testes simples e refatorações, encontrando o valor equivocado.

Ademais, foram feitas novas adições simples ao jogo, como por exemplo mostrar a vida do jogador e um menu principal, permitindo que o jogador reinicie a tabela de criaturas,

para testar simulações diferentes.

Capítulo 10

Conclusão

10.1 Resultados

O principal resultado do trabalho é o **Axolotl**, nome dado ao *sandbox* construído. O jogo é composto por sete salas conectadas e um total de 23 criaturas, que se deslocam livremente e interagem aleatoriamente. Além disso, o personagem do jogador possui diversas ações, permitindo bastante liberdade de movimentos como em jogos de plataforma tradicionais. Tanto as criaturas como o personagem são proceduralmente animados, utilizando algoritmos derivados do estudo de cinemática frontal e inversa, dando destaque ao uso de *Forwards and Backwards Reaching Inverse Kinematics* (Aristidou e Lasenby, 2011). Não somente isso, foi lançada uma versão de *playtest* que trouxe *feedbacks* importantes sobre o projeto, como por exemplo o fato de que os controles fluidos somados à animação do personagem proporcionaram uma experiência positiva, apesar dos problemas e inconvenientes já mencionados na Seção 9.1 Lançamento do *playtest*.

É também essencial mencionar as falhas e aspectos que poderiam ser melhorados no projeto. Infelizmente, o escopo inicial dele foi bastante reduzido e, consequentemente, a capacidade de avaliar a influência de animações procedurais num jogo de plataforma foi um tanto comprometida. **Axolotl** passou por diversos obstáculos, diversas reuniões com mudanças de planos e ajustes de escopo, gerando um produto final que deixa a desejar em alguns aspectos, como por exemplo os comportamentos das criaturas, que poderiam ser mais diversos e suas animações mais expressivas.

Por mais que o resultado do desenvolvimento não tenha sido o ideal, os ganhos positivos prevaleceram. O *sandbox* é funcional, com poucos problemas e com uma arquitetura facilmente expansível para um jogo completo. Além disso, as práticas de desenvolvimento aplicadas foram produtivas e o estudo das técnicas de animação gerou resultados tangíveis.

10.2 Próximos passos

Uma das continuidades mais importantes que poderia ser dada é o desenvolvimento de um sistema de testes mais robusto para entidades de jogo que operam fora da percepção do jogador. Mesmo com dois gerenciadores cujas informações podiam ser impressas no console, uma câmera que se movesse para qualquer ponto do mapa carregado e auxílio do *debugger* da *Godot*, testar o projeto se provou bastante difícil. Qualquer jogo que busque realizar um grande número de operações "escondidas" deveria ter métodos rápidos e práticos para testá-las. Seria interessante e produtivo construir essa infraestrutura de testes, pois simplesmente testar partes do jogo em isolamento e combiná-las depois não é tão simples quando há tamanha aleatoriedade envolvida.

Outro próximo passo bastante lógico é a implementação de elementos restantes que haviam sido delimitados no GDD. Mais criaturas e salas diferentes, objetos que permitissem mais interações e o desenvolvimento da narrativa do jogo proposta originalmente. Esse tipo de continuação provavelmente precisaria de um time maior e mais diverso por trás do projeto, ou de um intervalo de tempo maior para o desenvolvimento.

Finalmente, seria bastante interessante o uso das técnicas, algoritmos e sistemas sugeridos - e implementados - em outros gêneros de jogos. Este projeto foi fortemente inspirado por *Rain World* e, consequentemente, teve foco na experiência de jogo de plataforma e exploração. Entretanto, acredita-se que animação procedural e entidades com comportamentos baseados em aleatoriedade podem contribuir com a experiência de jogo independente de seu tipo - e a experimentação resultante é encorajada.

Apêndice A

Rascunhos e artes conceituais

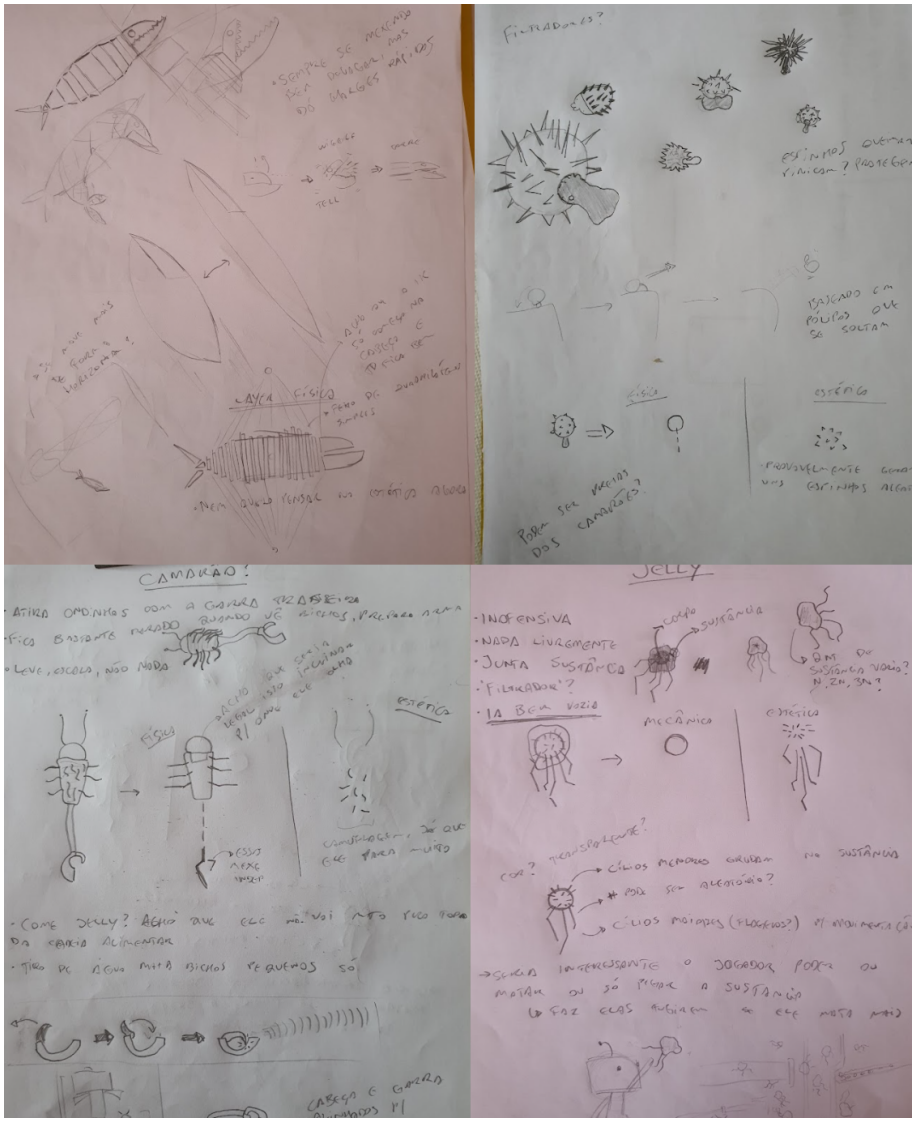

Figura A.1: *Rascunho das quatro criaturas originais: Barracuda, Urchin, Shrimp e Jelly.*

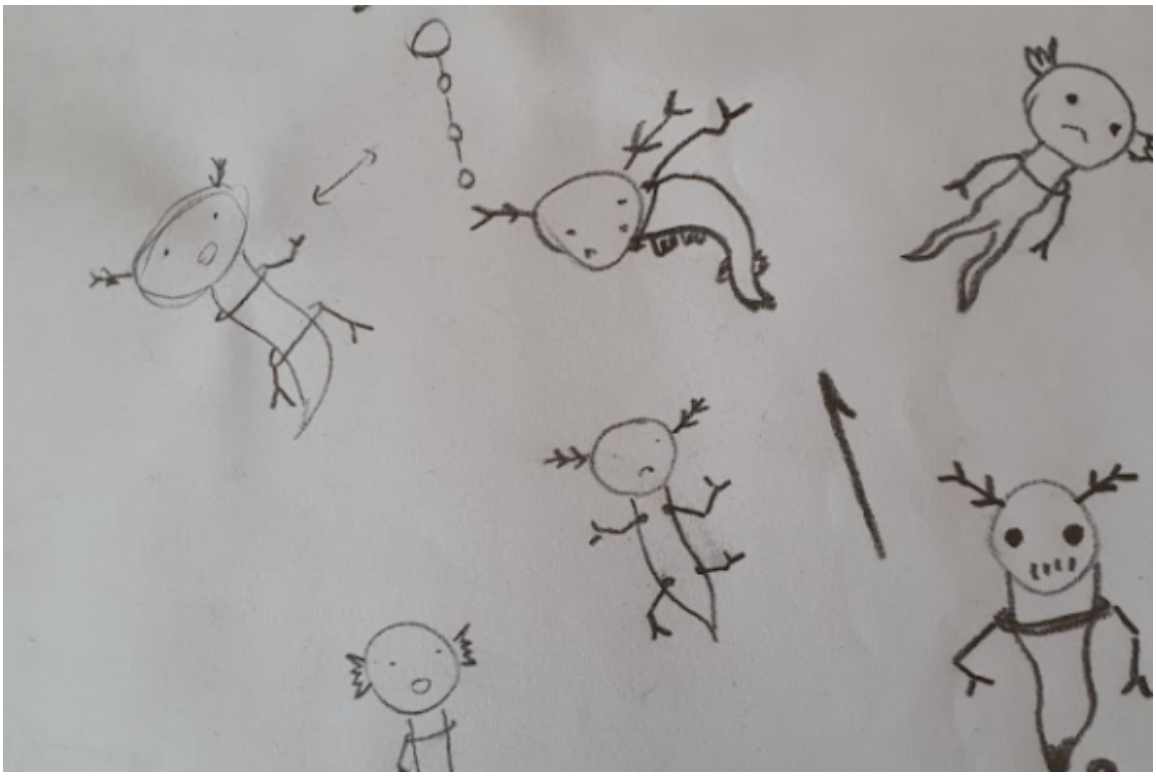

Figura A.2: *Rascunho da aparência original do jogador, baseada em um axolote. É possível também ver primeiros conceitos de sua camada física, que seria apenas um corpo como o da Barracuda, porém vertical.*

Apêndice B

Dados de *playtest*

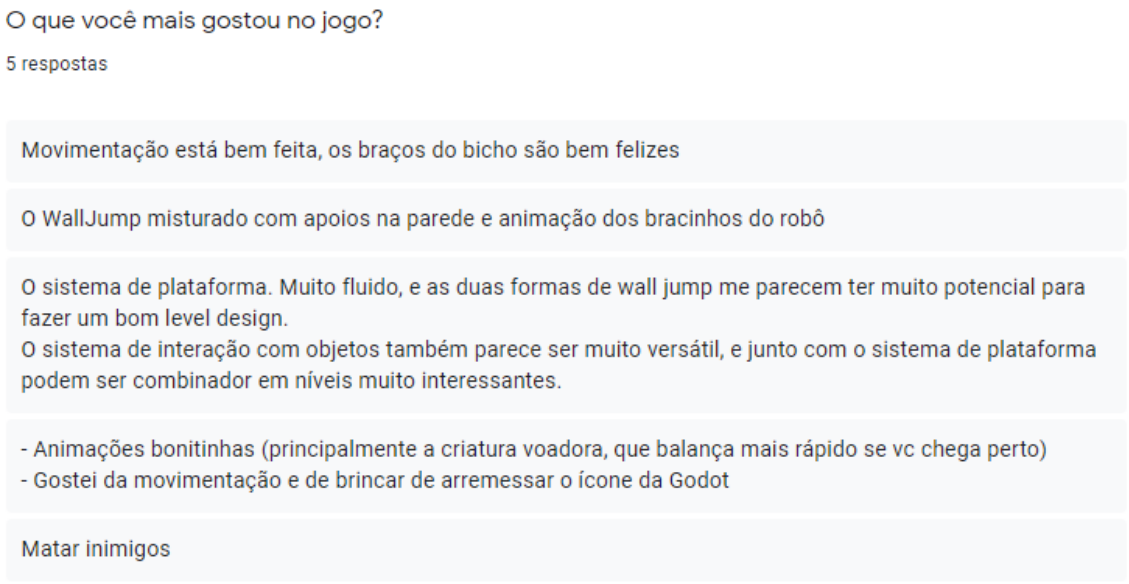

Figura B.1: *Respostas da primeira pergunta aberta.*

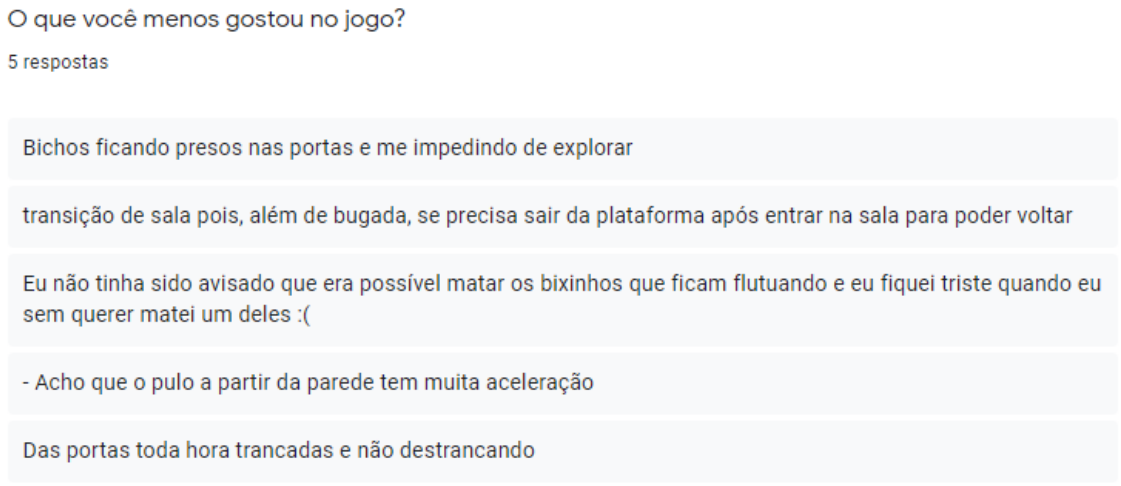

Figura B.2: *Respostas da segunda pergunta aberta.*

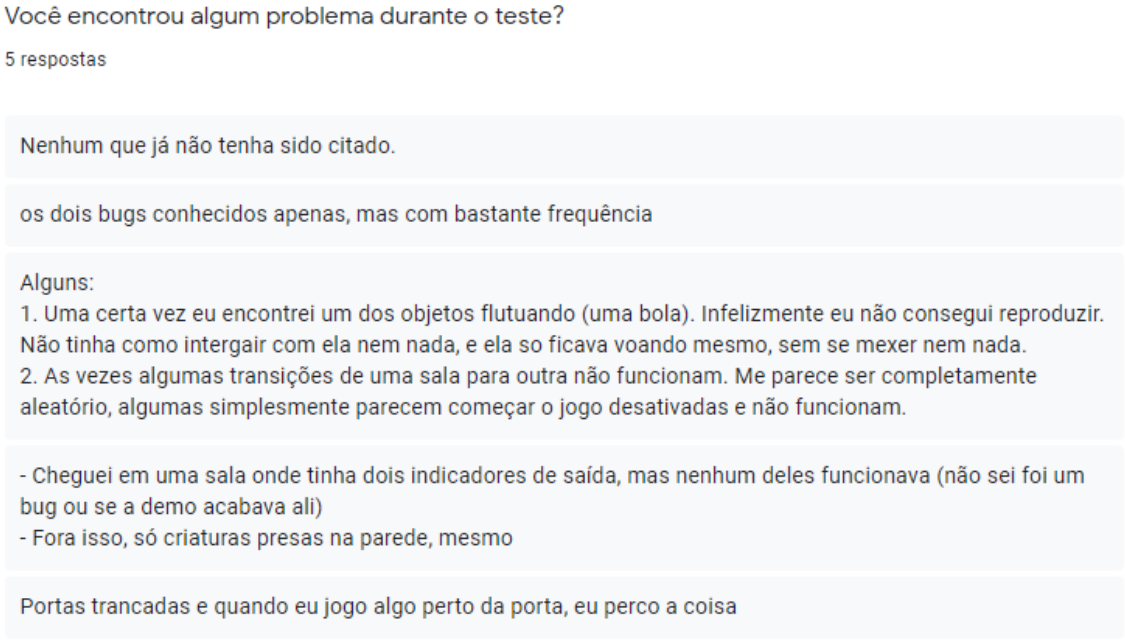

Figura B.3: *Respostas da terceira pergunta aberta.*

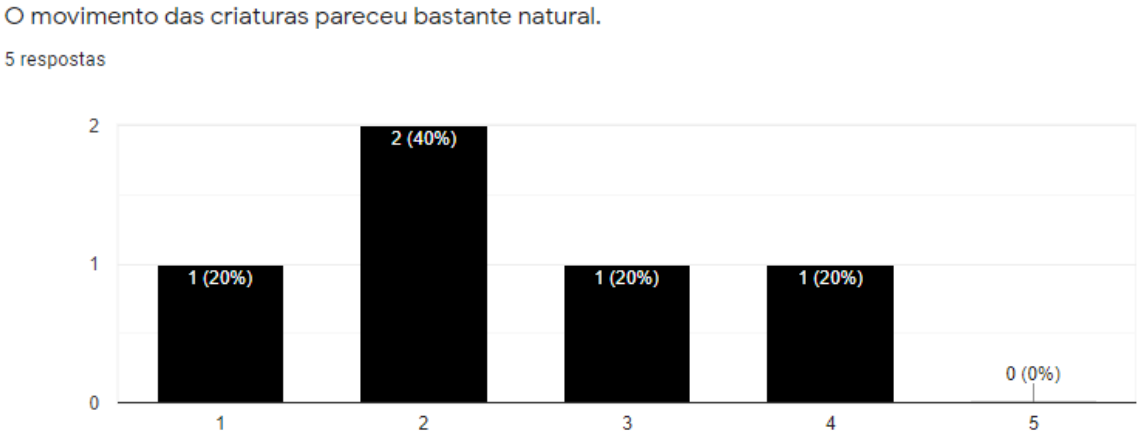

Figura B.4: *Respostas da primeira pergunta qualitativa.*

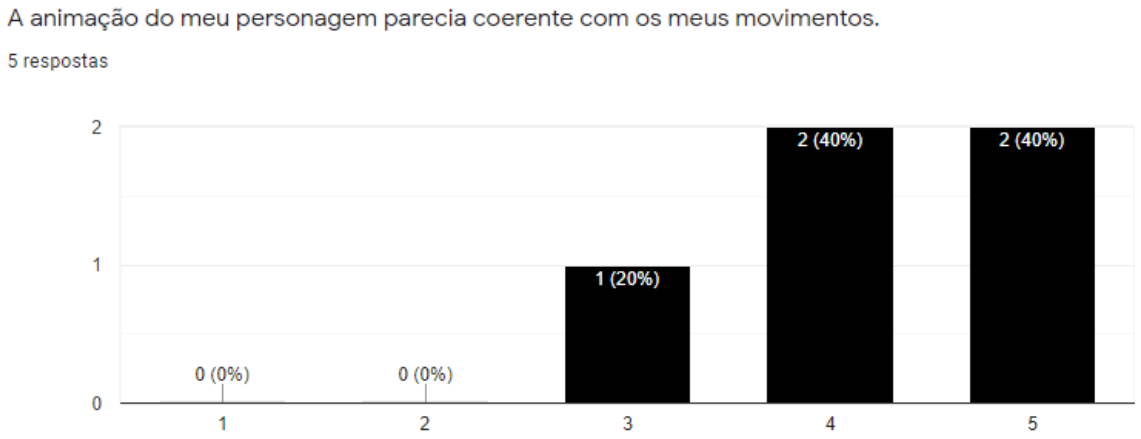

Figura B.5: *Respostas da segunda pergunta qualitativa.*

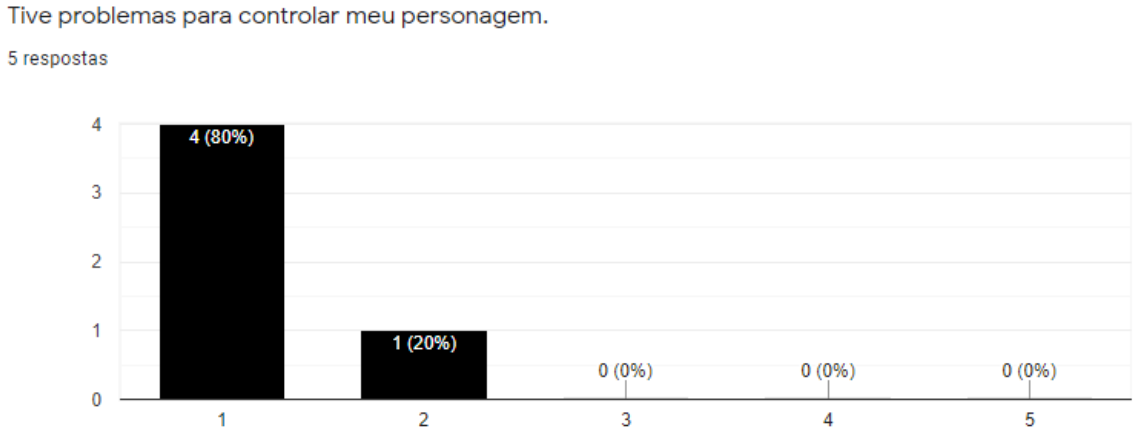

Figura B.6: *Respostas da terceira pergunta qualitativa.*

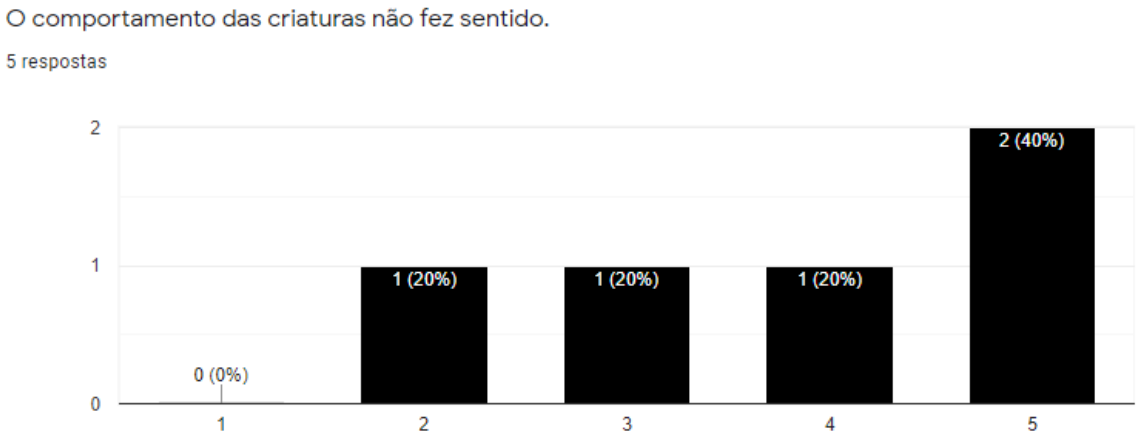

Figura B.7: *Respostas da quarta pergunta qualitativa.*

Anexo A

Game Design Document

O Game Design Document elaborado no início do projeto está disponível em GDD. Ele foi utilizado para direcionar o projeto e determinar que tipo de implementações seriam feitas no *sandbox*. A versão em PDF também está disponível na *homepage* do trabalho.

Anexo B

Repositório no *Git*

O repositório do GitLab que foi utilizado durante o projeto encontra-se na URL https://gitlab.com/lordanb/axolotl e todo o código fonte está disponível sob a licença GNU GPL 3.0. Uma cópia local e o executável do jogo estão ambos disponíveis na *homepage* do trabalho.

Anexo C

Versões animadas

Para ilustrar melhor os conceitos debatidos nesse trabalho, são necessárias versões animadas das imagens. Elas foram capturadas nos respectivos jogos indicados na legenda de cada uma e estão nomeadas de acordo com a ordem das imagens nesse documento, disponíveis em Versões Animadas. Essas imagens também estão disponíveis na *homepage* do trabalho.

Referências

[Aristidou e Lasenby 2011] Andreas Aristidou e Joan Lasenby. "Fabrik: a fast, iterative solver for the inverse kinematics problem". Em: *Graphical Models* 73 (set. de 2011), pgs. 243–260. doi: 10.1016/j.gmod.2011.05.003 (citado nas pgs. 33, 49).

[Berry 2018] Noel Berry. 2018. url: https://www.reddit.com/r/gamedev/comments/9a0cfr/comment/e4rvrg2/?utm_source=share&utm_medium=web2x&context=3 (acesso em 18/12/2021) (citado na pg. 8).

[Botea *et al.* 2013] Adi Botea, Bruno Bouzy, Michael Buro, Christian Bauckhage e Dana Nau. "Pathfinding in Games". Em: Dagstuhl Follow-Ups 6 (2013). Ed. por Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck e Julian Togelius, pgs. 21–31. issn: 1868-8977. doi: 10.4230/DFU.Vol6.12191.21 (citado na pg. 19).

[Brathwaite e Schreiber 2009] Brenda Brathwaite e Ian Schreiber. Course Technology, 2009 (citado na pg. 11).

[Cunha 2020] Matheus Lima Cunha. "Desenvolvimento de um jogo do gênero metroidvania com geração procedural de mapas". Em: (dez. de 2020). url: https://bcc.ime.usp.br/tccs/2020/matheusl/Monografia_TCC.pdf (citado na pg. 5).

[Dotsenko 2017] Andrew Dotsenko. *Designing Game Controls*. 2017. url: https://www.gamedeveloper.com/disciplines/designing-game-controls (acesso em 20/12/2021) (citado na pg. 28).

[Elias *et al.* 2012] George Skaff Elias, Richard Garfield e K. Robert Gutschera. *Characteristics of Games*. The MIT Press, 2012 (citado na pg. 3).

[Gamma *et al.* 1995] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. "Singleton". Em: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Reading, Mass. : Addison-Wesley, 1995, pgs. 127–129. isbn: 9780201633610 (citado na pg. 20).

[Gifford 2010] Kevin Gifford. *Super Mario Bros.' 25th: Miyamoto Reveals All*. 2010. url: https://web.archive.org/web/20160813205247/http://www.1up.com/news/super-mario-bros-25th-miyamoto (acesso em 17/12/2021) (citado na pg. 1).

[Gregory 2019] Jason Gregory. *Game Engine Architecture*. 3ª ed. AK Peters, 2019 (citado na pg. 15).

[Johnson 1997] Ralph E. Johnson. "Frameworks = (components + patterns)". Em: *Commun. ACM* 40.10 (out. de 1997), pgs. 39–42. ISSN: 0001-0782. DOI: 10.1145/262793.262799 (citado na pg. 16).

[Kohler 2016] Chirs Kohler. *Power-Up: How Japanese Video Games Gave the World an Extra Life, 2016 Edition*. Brady Games, 2016 (citado na pg. 3).

[Millington e Funge 2009] Ian Millington e John Funge. *Artificial Intelligence for Games*. 2ª ed. Elsevier, 2009 (citado na pg. 3).

[Nystrom 2014] Robert Nystrom. *Game Programming Patterns*. 2ª ed. Genever Benning, 2014 (citado na pg. 15).

[Pérez-Quiñones e Sibert 1996] Manuel Pérez-Quiñones e John Sibert. "A collaborative model of feedback in human-computer interaction". Em: *Conference on Human Factors in Computing Systems* (jun. de 1996). DOI: 10.1145/238386.238535 (citado na pg. 1).

[Richards 2010] Tony Richards. *Frameworks vs. Engines*. 2010. URL: https://web.archive.org/web/20100908180609/http://indiegamesguild.com/sgtflame/2010/02/23/frameworks-vs-engines/ (acesso em 18/12/2021) (citado na pg. 16).

[Rodrigues 2021] Eduardo Yukio Rodrigues. "Jogo digital 2d focado na expressão do jogador por meio da seleção de mecânicas de jogo". Em: (mar. de 2021). URL: https://bcc.ime.usp.br/tccs/2020/eduyukio/Monografia(Revisada)_EduardoYukio.pdf (citado nas pgs. 28, 38).

[Saeed e Fadhil 2020] Noor Saeed e Tahseen Fadhil. *Forward and Inverse Kinematic Analysis of Robots: Industrial Robotics*. LAP LAMBERT Academic Publishing, 2020 (citado na pg. 9).

[Schell 2020] Jesse Schell. *The Art of Game Design: A Book of Lenses*. 3ª ed. AK Peters/CRC Press, 2020 (citado na pg. 1).