ASTRAL: Produção e documentação de um jogo gênero plataforma em pixel art.

Pedro H. F. G. Oliveira, Lucas N. Ferreira

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Abstract. *This technical report presents ASTRAL, a 2D pixel-art game developed with a strong focus on dynamic movement and combat mechanics designed to deliver an immersive player experience. The development process integrated key concepts such as state machines and game design. The narrative follows Zoe, a young girl who awakens in a mysterious world to find her parents missing. Determined to reunite with them, she embarks on a cosmic journey filled with challenges, discoveries, and dangers. Playtests were executed and evaluated in order to collect feedback on the game.*

Resumo. *Este relatório técnico apresenta o projeto ASTRAL, um jogo 2D em pixel art desenvolvido com foco em mecânicas dinâmicas de movimentação e combate, proporcionando uma experiência imersiva ao jogador. O desenvolvimento do jogo explora conceitos fundamentais de máquinas de estado e game design. Na narrativa, acompanhamos Zoe, uma pré-adolescente que desperta em um mundo misterioso sem a presença de seus pais. Determinada a reencontrá-los, ela embarca em uma jornada cósmica repleta de desafios, descobertas e perigos. Playtests foram executados e avaliados para coletar feedback sobre o jogo.*

1. Introdução

1.1. Contexto e Problema

O mercado de jogos digitais apresenta crescimento contínuo e já conta com 3.6 bilhões de jogadores globais, movimentando cerca de 189 bilhões de dólares em 2025 [Newzoo 2025]. Este cenário evidencia a relevância dos jogos como mídia interativa e objeto de estudo técnico-científico.

No gênero de jogos de plataforma 2D, títulos independentes como *Celeste*, *Hollow Knight* e *ScourgeBringer* demonstram que mecânicas bem elaboradas e implementação técnica sólida podem resultar em experiências envolventes. Entretanto, o desenvolvimento desses jogos demanda conhecimento aprofundado em áreas como inteligência artificial para NPCs, sistemas de física, detecção de colisão, entre outros.

Este trabalho foi iniciado no segundo semestre de 2025, e foi finalizado no primeiro semestre de 2026, com o intuito de produzir um relatório do processo de criação de um jogo autoral completo, denominado *ASTRAL*, focando em movimentação e combate dinâmicos.

O problema central abordado neste trabalho é a produção de um jogo autoral com a aplicação de mecânicas dinâmicas de movimentação e combate. Especificamente,

investiga-se como implementar mecânicas envolventes para o jogador, utilizando de estruturas como máquinas de estados finitas (FSMs) e principios de Game Design, sem o auxílio de engines comerciais que abstraem essas complexidades.

Durante o primeiro semestre de execução deste trabalho, havia um grande foco no sistema de pathfinding utilizado para movimentação de NPCs. No segundo semestre, entretanto, o uso de algoritmos de caminhamento tornou-se secundário. Ele ainda existe no código do jogo, mas seu uso foi descontinuado para melhorar a experiência do jogador. Isso será detalhado mais à frente.

1.2. Motivação

O desenvolvimento de *ASTRAL* justifica-se por três aspectos principais. Primeiro, a implementação direta em C++ com a biblioteca SDL2 proporciona compreensão profunda dos fundamentos computacionais de jogos, incluindo renderização gráfica, gerenciamento de memória e otimização de desempenho.

Segundo, o projeto permite explorar aplicações concretas de conceitos algorítmicos estudados na graduação, como estruturas de dados espaciais (spatial hashing), algoritmos de busca (A*) e máquinas de estados. Esta abordagem conecta teoria e prática de forma tangível.

Terceiro, o contexto brasileiro apresenta desafios específicos para desenvolvimento de jogos, evidenciados pela ausência de cursos dedicados na UFMG. Embora o Marco Legal dos Games [Ministério da Cultura 2026] tenha avançado ao reconhecer jogos eletrônicos como obras audiovisuais, abrindo acesso a mecanismos de financiamento como a Lei Rouanet e o Fundo Setorial do Audiovisual, o veto ao abatimento de 70% do IR para remessas ao exterior em projetos independentes [Brasil 2024] ilustra que os incentivos ainda estão em maturação.

Nesse cenário, a escassez de formação técnica especializada é um gargalo tão relevante quanto o financeiro. Este trabalho contribui para fortalecer a formação de desenvolvedores nacionais através da documentação de um processo completo de produção de um jogo.

1.3. Objetivos

1.3.1. Objetivo Geral

Desenvolver um jogo 2D com foco em plataforma que explore a implementação de mecânicas adaptativas, utilizando C++ e SDL2, sem engine comercial.

1.3.2. Objetivos Específicos do primeiro semestre

- Elaborar documentação de game design (GDD) definindo mecânicas, sistemas e escopo do projeto;
- Implementar sistema de movimentação imersivo para a personagem principal;
- Desenvolver inteligência artificial para inimigos utilizando FSMs e algoritmos de pathfinding (A*);
- Implementar mecânicas de habilidades especiais com física customizada;

- Produzir versão jogável do protótipo;
- Realizar playtests com usuários reais para validação das mecânicas e identificação de melhorias.

1.3.3. Objetivos Específicos do segundo semestre

- Analisar os feedback coletados no primeiro semestre;
- Adaptar a experiência de jogo de acordo com os feedback coletados;
- Implementação de novas mecânicas;
- Executar um segundo ciclo completo de playtest com novos jogadores, expandindo a base de feedback;
- Coletar e analisar os dados de feedback dos playtests;
- Finalizar a versão completa.

1.4. Organização do Trabalho

Este documento está organizado em sete principais:

Seção 1 Corresponde a introdução, que contextualiza de maneira geral do que se trata o trabalho, e quais seus objetivos.
Seção 2 Apresenta o referencial teórico, abordando conceitos fundamentais de jogos digitais, game design, tecnologias utilizadas (SDL2, C++) e trabalhos correlatos no gênero de plataforma 2D.
Seção 3 Descreve a concepção do jogo, incluindo visão geral, público-alvo, mecânicas centrais, narrativa.
Seção 4 Documenta a arquitetura técnica, detalhando módulos, componentes, modelagem de dados e fluxos importantes.
Seção 5 Aborda aspectos de arte, áudio e interface, explicando direção visual, UX e integração sonora.
Seção 6 Detalha a implementação, ambiente de desenvolvimento, metodologia de playtests e resultados obtidos.
Seção 7 Discorre sobre as conclusões gerais sobre o trabalho.

2. Referencial Teórico e Trabalhos Relacionados

2.1. Conceitos de Jogos Digitais e Game Design

O desenvolvimento de jogos digitais fundamenta-se em conceitos consolidados de game design que orientam a construção de experiências interativas coesas. Esta seção apresenta os principais elementos teóricos aplicados em *ASTRAL*.

2.1.1. Gênero Plataformer 2D

Um jogo costuma ser classificado em um gênero específico, que define a experiência de jogo e as expectativas do jogador. *ASTRAL* pertence ao gênero *plataformer* 2D, que se caracteriza por desafios de movimentação e exploração em ambientes bidimensionais, com foco em saltos precisos, controle de personagem e navegação por plataformas.

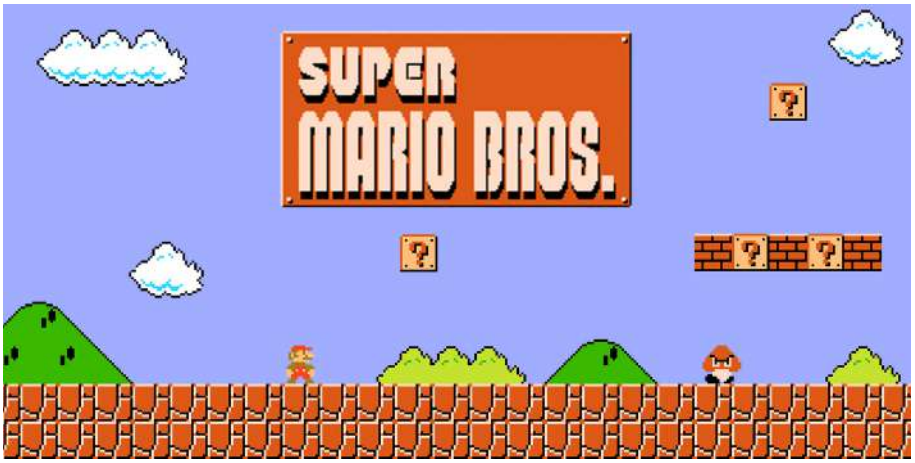

Figure 1. Screenshot de Super Mario Bros, exemplo de plataformer 2D. [Nintendo 1985]

2.1.2. Game Loop

O `game loop` constitui o núcleo de execução de um jogo, representando o ciclo contínuo de processamento de entrada, atualização de estado e renderização [Madhav 2013]. Sua estrutura típica consiste em três etapas principais: processamento de input do jogador, atualização da lógica do jogo, e renderização dos elementos visuais na tela. A taxa de atualização deste ciclo, medida em frames por segundo (FPS), impacta diretamente a responsividade e fluidez da experiência.

Para *ASTRAL*, a experiência está limitada a um máximo de 60 FPS (sessenta atualizações por segundo). Toda a experiência de jogo foi construída e testada sob esta taxa de atualização. Mas isso não significa que FPS menores (em caso de máquinas menos potentes) causam uma experiência de jogo ruim para os jogadores.

Tempo delta, ou ”delta time”, é uma variável ponto flutuante cujo intuito é balancear as taxas de atualização entre máquinas, que podem ter poder computacional diferente. Ela é uma variável computada a cada loop, que contabiliza a diferença de tempo entre a execução do último loop e do loop atual. Sua finalidade é ponderar qualquer tipo de mecanismo temporal do jogo.

Com isso, computadores mais lentos ponderam suas atualizações (que ocorrem menos vezes) com valores maiores, enquanto computadores mais rápidos ponderam suas atualizações (que ocorrem mais vezes) com valores de menores. Mas, em casos extremos de FPS baixo, a falta de atualizações suficientes prejudica a experiência do jogador, de forma que o balanceamento da *”dt”* não é suficiente. Por isso, é fundamental pensar em otimização no contexto de jogos digitais.

2.1.3. Cutscenes

Cutscenes são cenas pré determinadas que executam por determinados gatilhos no jogo. Elas controlam os atores do jogo (entidades, jogador, câmera, física...) de forma que o jogadores não necessariamente interagem. São usadas geralmente com fins de narrativa.

Algumas deltas apresentam certos níveis de interação com o jogador, por exemplo os *Quick Time Events*, usados em jogos *hack and slash*. Em que, durante uma cena pré configurada, o jogador é obrigado a apertar uma sequência de botões para responder aos eventos ocorrendo na tela, em tempo real, dando uma sensação de controle e interação sob um evento pré configurado. Em *ASTRAL*, esse conceito é usado para introdução de mecânicas.

2.1.4. Juiceness

"Juiceness" ou "Game feel" [Swink 2009] é um conceito subjetivo diretamente relacionado a experiência do jogador, que é o maior foco de qualquer design de jogos. Ele corresponde ao sentimento que o jogo causa naquele que joga através da interação das suas mecânicas e do resultado gerado.

Quando o *Game feel* falha, o jogador tende a se desconectar das mecânicas, perdendo a conexão com os elementos que a obra utiliza para mantê-lo imerso. Um exemplo de quebra, que ocorreu nesse trabalho, é o dicotomia entre responsividade e rigidez.

A fim de facilitar testes, garantir previsibilidade, e por falta de conhecimento do conceito acima, a implementação da movimentação de iminigos em combate foi projetada inicialmente com intermédio de um algoritmo de busca de caminhos, denominado A*. Essa técnica permite encontrar os melhores caminhos, em tempo real, entre um ponto e outro durante o jogo.

Graças a isso, as movimentações de inimigos se tornavam rígidas, determinísticas e muito diretas. Então, a experiência fica difícil de uma forma pouco atrativa, e frustante, de acordo com o feedback coletado.

Nesse caso, a decisão tomada foi descontinuar o uso de algoritmos de caminhamento, em detrimento de algoritmos mais simples de movimentação, que levam em conta a percepção dos inimigos em relação ao jogador, dados os obstáculos da cena. Isso gera um *Game feel* mais orgânico, real e responsivo.

Para compreender essa decisão, é necessário entender a relação entre dificuldade e frustação. De acordo com a intenção de um jogo, o foco das suas mecânicas pode ser evocar frustação, tentativa e erro para recompensar o progresso.

Isso é comum em outro gênero, muito famoso atualmente, chamado *soulslike*. Mas, frustação é um sentimento que deve ser abordado com cuidado, por que se ele não estiver associado à dificuldade balanceada, e sim a mecânicas pouco testadas ou com bugs recorrentes, a imersão que era objetivo inicial se transforma em aversão.

No caso de *ASTRAL*, a frustração evocada por inimigos com movimentação rígida e determinística não estava associada a uma dificuldade balanceada, mas sim a mecânicas pouco testadas. Com isso, a imersão que era objetivo inicial se transformava em aversão, de acordo com o feedback dos jogadores. Isso será discutido à frente.

2.1.5. Mecânica, Dinâmica e Estética

O framework MDA (Mechanics, Dynamics, Aesthetics) fornece estrutura conceitual para análise de jogos [Hunicke et al. 2004]. Mecânicas são as regras e sistemas do jogo (pulo, combate, colisão...). Dinâmicas emergem da interação do jogador com as mecânicas (esquivar de inimigos, combinar habilidades). Estética refere-se à resposta emocional evocada (desafio, descoberta, tensão).

Em jogos de plataforma 2D, mecânicas como o controle preciso de movimento, física de pulo ajustável e feedback imediato de colisão são fundamentais para a estética de desafio e maestria progressiva, que envolve o jogador entender as mecânicas e dinâmicas do jogo conforme joga, falhando e tentando novamente, ou refletindo e ponderando com base naquilo que já sabe para tomar a próxima ação.

2.1.6. Level Design

Level design envolve a disposição espacial de elementos de jogo para guiar o jogador, controlar dificuldade e comunicar informações. Princípios incluem: introdução gradual de mecânicas, uso de recursos visuais para indicar interatividade e balanceamento entre desafio e recompensa.

Em plataformas 2D, o design vertical (plataformas em diferentes alturas) e horizontal (progressão linear) define o ritmo da experiência. A colocação de inimigos e obstáculos deve considerar o tempo de reação do jogador e a curva de aprendizado.

Abaixo temos o exemplo de um técnica simples e efetiva. Ela já foi testada em títulos famosos como Super Mario Bros [Nintendo 1985] e Super Meat Boy [Meat 2010]. Consiste em posicionar o cenário com uma clara progressão à direta.

Mesmo jogadores não experientes no gênero costumam entender com relativa facilidade a comunicação de que o caminho correto está para a direita. Já para jogadores experientes, a simples configuração de nível abaixo não costuma exigir sequer uma cutscene para entedimento.

Figure 2. Primeira cena do segundo nível do jogo

Antes desse nível ser jogável, uma cutscene mostra uma Estrela se movimentando e passando justamente pela brecha à direta. Além disso, essa mesma cena movimenta a protagonista até que ela fique a beira do buraco e pare, também comunicando a ideia de obstáculo à frente.

Comunicar implicitamente, sem dizer exatamente o que deve ser feito por um diálogo ou texto qualquer, é um princípio importante do game design. É essa prática que caracteriza imersão, em que um jogador não é explicitamente comunicado sobre o que fazer a cada passo, mas sim orientado de maneira fluída e envolvente pelos designers.

O Level Design introduz entidades não controláveis pelo jogador no jogo, que são chamadas na indústria de ”Non playable characters”, ou NPCs. Eles geralmente fazem parte ou da narrativa (representando personagens), ou dos desafios do jogo, representando adversários/inimigos.

Em *ASTRAL*, os NPCs são os seguintes:

- Inimigos:

Figure 3. Quasar

Figure 4. Zathura

Figure 5. Sith

Figure 6. Zod

- Não inimigos:

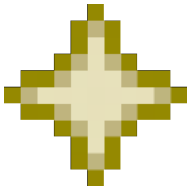

Figure 7. Estrela

Figure 8. Mãe da protagonista

Figure 9. Pai da protagonista

Aproveitando o espaço dedicado aos NPCs, em seguida consta a protagonista, chamada Zoe que é o unico personagem jogável (Playable Character):

Figure 10. Protagonista

2.1.7. Feedback ao Jogador

Sistemas de feedback comunicam o estado do jogo ao jogador através de canais visuais, sonoros e táteis. Feedback efetivo é imediato (resposta instantânea a ações), claro (comunica resultado sem ambiguidade) e proporcional (intensidade corresponde à importância da ação). Por exemplo, animações de dano em inimigos, sons de impacto, shake de câmera, indicadores de vida na HUD e efeitos de partículas para habilidades especiais.

DEMO's ou demonstrações são momentos em os desenvolvedores de jogos se organizam para publicar uma versão não finalizada do jogo, para capturar as impressões de jogadores teste. Elas comumente são definidas em *Alpha* e *Beta* no mercado de jogos, para indicar nível de proximidade ao produto final, respectivamente.

A primeira *DEMO* de *ASTRAL*, uma *Alpha*, teve reclamações constantes da falta de feedback efetivo para momentos de combate e movimentação. Esses feedbacks foram capturados e serão incluídos nas próximas versões do jogo. Ademais, as *DEMOs* seguintes de *ASTRAL* não foram perfeitas, pois o processo de manufatura de um jogo é iterativo. Isso significa dizer que são feitos sucessivos playtests, captura e interpretação dos dados, para a melhora contínua do jogo.

Esse processo não para mesmo após o lançamento oficial da obra. A partir de então, jogos continuam capturando feedbacks de varias formas, por exemplo: análise da produção de conteúdo dos jogadores em plataformas online, ou analytics inseridos dentro

dos jogos, que capturam quantas pessoas estão jogando por dia, quantas missões são completas por hora, entre outros.

2.2. Tecnologias e Ferramentas

2.2.1. SDL2 (Simple DirectMedia Layer)

SDL2 é uma biblioteca multiplataforma que fornece acesso de baixo nível a recursos de áudio, teclado, mouse, joystick e hardware gráfico [(SDL) 2025]. Diferentemente de engines completas, SDL2 oferece primitivas básicas, exigindo implementação manual de sistemas de alto nível.

A biblioteca organiza-se em subsistemas: SDL_Video para renderização 2D acelerada por hardware, SDL_Audio para reprodução de áudio, SDL_Events para gerenciamento de entrada, e SDL_GameController para suporte a controles, entre outros. Esta abordagem de baixo nível proporciona controle fino sobre desempenho e comportamento, adequada para aprendizado de fundamentos.

2.2.2. C++ como Linguagem Base

C++ foi escolhido por combinar performance próxima a C com recursos de orientação a objetos e programação genérica. O controle manual de memória e acesso direto a recursos de hardware tornam a linguagem apropriada para aplicações que demandam otimização, como jogos em tempo real.

Recursos utilizados incluem: herança e polimorfismo para hierarquia de atores e componentes, templates para containers genéricos, smart pointers (quando aplicável) para gerenciamento de recursos, e a Standard Template Library (STL) para estruturas de dados fundamentais.

2.2.3. Ferramentas Complementares

Além do SDL2, o projeto integra outros recursos auxiliares:

- **SDL_image**: carregamento de formatos de imagem (PNG, JPG);
- **SDL_mixer**: reprodução de áudio com suporte a múltiplos canais e formatos;
- **SDL_ttf**: renderização de texto com fontes TrueType;
- **nlohmann/json**: parsing de arquivos JSON para configuração de mapas e dados [nlohmann 2025];
- **Tiled Map Editor**: ferramenta externa para criação de níveis em formato tilemap [Lindeijer 2025].
- **Piskel e Aseprite**: ferramentas para criação/edição de pixelart [Piskel 2025] [Aseprite 2025].
- **Texture packer**: ferramenta online para codificar animações em JSONs interpretáveis em código [Norinchak 2025].

2.2.4. Ausência de Engine Comercial

A decisão de não utilizar engines como Unity ou Godot fundamenta-se no objetivo pedagógico do projeto. Engines abstraem implementações de física, renderização, IA e gerenciamento de cena, limitando compreensão profunda destes sistemas. A implementação direta em SDL2 expõe desafios pedagógicos: gerenciamento manual de memória, otimização de consultas espaciais, sincronização de `game loop` e integração de subsistemas.

2.3. Jogos e Trabalhos Correlatos

2.3.1. Celeste (2018)

Celeste exemplifica design refinado em plataforma 2D, com mecânicas centradas em movimentação precisa e desafio progressivo [Corporation 2018]. O jogo implementa dash direcional com cooldown, wall-jump e coyote time (tolerância para pulo após sair de plataforma), mecânicas que priorizam responsividade sobre realismo físico.

2.3.2. Hollow Knight (2017)

Hollow Knight demonstra integração profunda entre combate, exploração e progressão em mundo interconectado [Corporation 2017]. A IA de inimigos utiliza máquinas de estados finitas com comportamentos variados: patrulha, perseguição aérea, ataques telegrafados e recuo estratégico.

2.3.3. ScourgeBringer (2020)

ScourgeBringer [Corporation 2020] é um roguelike de movimentação ágil focado em combate. É uma inspiração que surgiu no segundo semestre de elaboração deste trabalho. Seu combate e movimentação fluida são inspirações para a implementação de *ASTRAL*.

2.3.4. Diferenciais de ASTRAL

ASTRAL diferencia-se dos trabalhos correlatos em aspectos específicos:

1. **Implementação técnica exposta**: enquanto jogos comerciais utilizam engines, *ASTRAL* implementa sistemas fundamentais manualmente, documentando decisões arquiteturais e algoritmos aplicados (spatial hashing, A*, FSMs);
2. **Habilidades com física customizada**: mecânicas como bola de fogo, nevasca e ventania são adaptações, reinvenções ou novas ideias perante as mecânicas das obras citadas;
3. **Foco pedagógico**: o projeto prioriza documentação de processo e exposição de fundamentos técnicos, servindo como material de referência para desenvolvimento sem engine.
4. **Equipe reduzida**: Esses jogos são comumente referidos como *indies*, por serem desenvolvidos por pequenas equipes. Para este trabalho, a equipe se resume a duas pessoas, que são os autores.

As limitações em relação aos trabalhos correlatos incluem escopo reduzido (30 minutos de gameplay contra 10+ horas), assets visuais de terceiros (em comparação arte original) e ausência de alguns sistemas complexos (save/load, inventário persistente, árvore de habilidades). Estas limitações são conscientes e alinhadas aos objetivos de um projeto acadêmico de um ano, com um desenvolvedor e um orientador.

3. Concepção do Jogo (Game Design)

3.1. Visão Geral do Jogo

ASTRAL é um jogo de plataforma 2D com elementos de ação e exploração, desenvolvido para PC, e distribuído em navegadores. Com isso, pode ser jogado em virtualmente qualquer plataforma com acesso a navegadores modernos, e é compatível somente com controles (joysticks, dualshocks...). O tempo médio de gameplay do jogo final é de aproximadamente 30 minutos, concentrando-se em uma experiência linear que introduz progressivamente as mecânicas centrais.

O jogo utiliza estética visual de pixel art, controles responsivos inspirados em jogos de plataforma clássicos e sistema de combate que combina ataques corpo a corpo com habilidades de longo alcance.

3.2. Público-Alvo e Experiência Desejada

O público-alvo principal consiste em jogadores casuais de jogos indie e entusiastas de plataformas 2D, com idade entre 12 e 35 anos. O jogo busca atrair jogadores que apreciam títulos como *Celeste*, *Scourgebringer*, *Hollow Knight* e *Super Meat Boy*, mas não requer experiência prévia no gênero devido à curva de aprendizado gradual.

A experiência emocional desejada combina familiaridade e descoberta. O controle básico (movimentação lateral e pulo) é imediatamente reconhecível para jogadores de plataforma, reduzindo barreira de entrada. As habilidades elementais introduzem complexidade progressiva, gerando momentos de experimentação e maestria.

Busca-se evocar três respostas emocionais principais:

- **Descoberta**: a ambientação espacial misteriosa e a perseguição da estrela-guia criam curiosidade narrativa;
- **Desafio controlado**: obstáculos e inimigos oferecem resistência calibrada, recompensando aprendizado. Para isso os níveis existentes foram testados iterativamente. A quantidade de inimigos e desafios foi ajustada de acordo com os feedbacks de dificuldade e frustação dos jogadores;
- **Expressividade**: variedade de habilidades permite ao jogador escolher abordagens (combate direto, controle de área, mobilidade aumentada).

3.3. Mecânicas Centrais e Loop de Jogo

3.3.1. Movimentação

A movimentação básica compreende deslocamento lateral analógico, suporte único a controle, e pulo. A física de pulo utiliza uma variável gravidade para simular física, e limita a velocidade máxima de um corpo para evitar comportamentos abruptos por cálculos eventualmente errôneos.

A mecânica de **Ventania** fornece impulso direcional no ar, aplicando força em cinco direções possíveis, de acordo com a entrada do jogador. Só pode ser usada uma vez a cada pouso, que previne uso contínuo, mas abre diversas portas para movimentação dinâmica. Pode ser usada estando agarrada à parede.

A mecânica de **Nevasca** dispara particulas de gelo que são capazes de criar neve acima de superfícies, ou congelar objetos. Neve diminui o coeficiente de atrito dos atores, permitindo movimentações mais velozes, se usada da forma correta. Alguns puzzles do jogo estão atrelados a mecânica de neve e congelamento. Para usá-la é necessário ter mana.

A mecânica de **Cling/Agarro** permite a protagonista pressionar contra paredes para grudar nelas, diminuindo a velocidade de queda, permitindo elaboração de estratégias, e parkour.

Figure 11. Ventania, a seta indica a partícula disparada pela mecânica.

Figure 12. Nevasca.

Figure 13. Agarro na parede.

3.3.2. Combate

O sistema de combate do jogador combina as abordagens:

1. **Ataque corpo a corpo**: golpe com hitbox ativa por tempo, causando knockback em inimigos. Disponível no chão e no ar;
2. **Ataque corpo a corpo carregado**: semelhante ao golpe padrão, mas só ocorre quando a protagonista está no chão, com um tempo de carregamento prévio. Esse golpe interage com determinados objetos e inimigos, arremessando-os para cima;
3. **Bola de Fogo**: projétil que explode ao colidir, causando dano. Para usá-la é necessário ter mana. Tem tempo de recarga (cooldown). Ela não causa dano ao jogador;
4. **Dodge**: esquiva com invencibilidade que ignora colisão com inimigos e projéteis inimigos durante animação, e por um pequeno tempo extra. Tem cooldown.
5. **Congelamento**: Nevasca é capaz de congelar todos os inimigos do jogo.

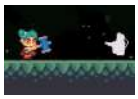

Figure 14. Ataque padrão.

Figure 15. Bola de fogo explodindo.

Figure 16. Zoe prestes a desviar de um projétil inimigo.

Figure 17. Ataque carregado.

Figure 18. Inimigo congelado por Nevasca.

3.4. Narrativa e Ambientação

3.4.1. Premissa

Zoe é uma pré-adolescente que, em um dia comum preparando-se para visitar a avó, descobre um portal dimensional no quarto dos pais. Ao ser sugada para o *Espaço Astral*, ela encontra uma estrela misteriosa que serve como guia, levando-a através de desafios complexos.

Mais tarde durante o jogo, Zoe descobre que a Estrela é, na verdade, seu pai disfarçado. O intuito é proteger Zoe, enquanto ela enfrenta desafios para se tornar mais forte. A narrativa revelará que a estrela prepara Zoe para enfrentar Zathura, entidade que mantém sua mãe em cativeiro.

Ao final do jogo, Zoe derrota Zathura com a ajuda de sua família e volta para casa. Para evitar que ela volte para o *Espaço Astral*, seus pais tratam a situação sobrenatural como um pesadelo da protagonista.

3.4.2. Progressão Narrativa

A história desenvolve-se através de cutscenes, utilizando um sistema de diálogos e movimentação scriptada. Existe um sistema de items com cutscenes associadas, para introduzir mecânicas e habilidades da personagem de forma gradativa.

3.4.3. Ambientação Visual

O *Espaço Astral* é representado por backgrounds de galáxias, plataformas flutuantes com estética de ruínas antigas e inimigos com design espacial alienígena (atiradores, criaturas voadoras, golems ancestrais e um Troll). Existem inspirações nos nomes de inimigos, ambientes e habilidades, mencionadas nos créditos do jogo, em obras da cultura pop de ficção, como Star Wars [Lucasfilm 2019] e Zathura [Favreau 2005].

4. Arquitetura Técnica do Jogo

4.1. Visão Geral da Arquitetura

A arquitetura de *ASTRAL* organiza-se em camadas modulares que separam responsabilidades de renderização, lógica de jogo, física e entrada. A Figura 19 apresenta diagrama dos módulos principais e suas interações.

A classe `Game` atua como controlador central, gerenciando `game loop`, transições de cena e referências a sistemas globais. `Actor` representa entidades do jogo (jogador, inimigos, tiles), agregando `Components` que implementam comportamentos específicos. `SpatialHashing` otimiza consultas espaciais para detecção de colisão.

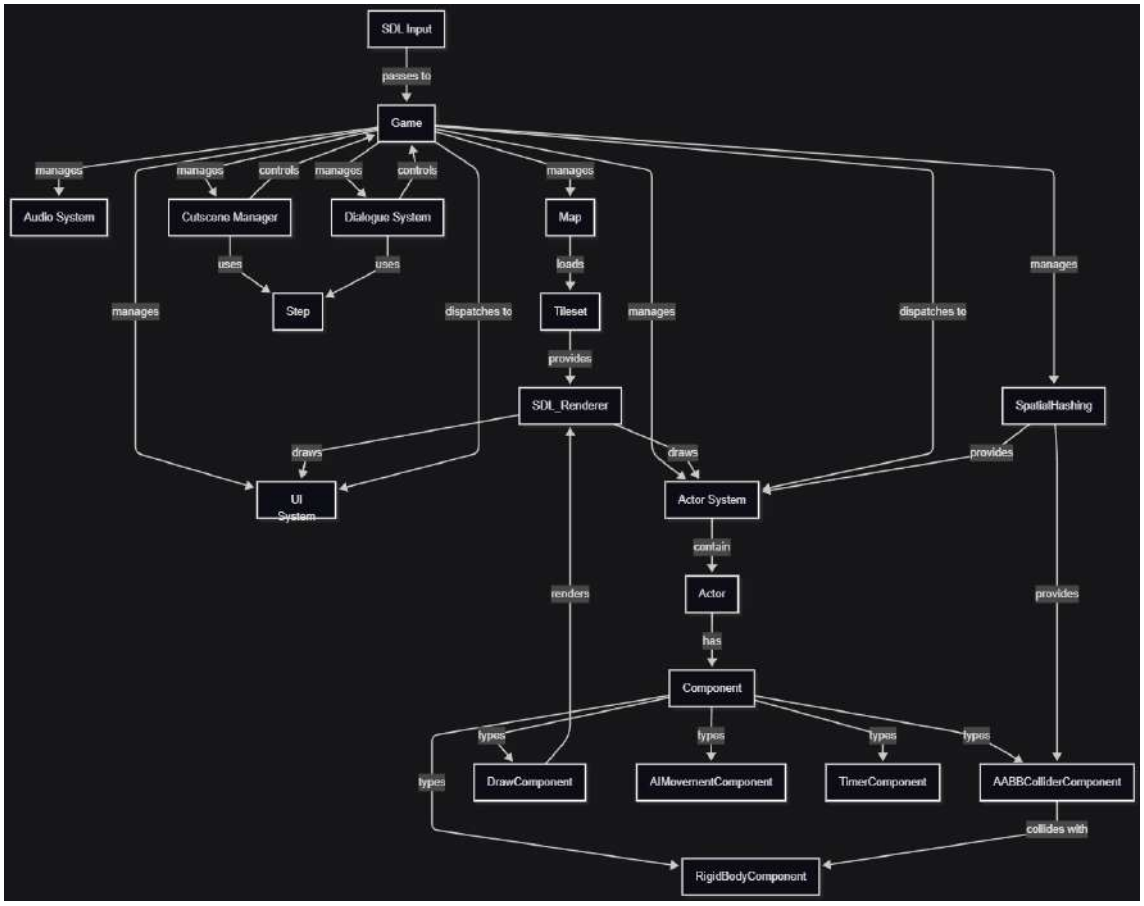

Figure 19. Diagrama de arquitetura geral

4.2. Componentes e Módulos Principais

4.2.1. Organização de Código

A estrutura de diretórios organiza-se por domínio funcional:

```
src/
 core/
 actors/
 components/
 ui/
 libs/
```

4.2.2. Padrão Entity-Component

O jogo implementa uma variação do padrão Entity-Component-System (ECS) [Bilas 2002]. `Actor` funciona como entidade base, agregando `Components` que encapsulam comportamentos modulares:

```
class Actor {
protected:
    std::vector<Component*> mComponents;
    Vector2 mPosition;
    ActorState mState;
    BehaviorState mBehaviorState;

public:
    void Update(float deltaTime);
    void ProcessInput(const Uint8* keyState);

    template<typename T>
    T* GetComponent() const;
};

class Component {
protected:
    Actor* mOwner;
    int mUpdateOrder;

public:
    virtual void Update(float deltaTime);
    virtual void ProcessInput(const Uint8* keyState);
};
```

Listing 1. Interface reduzida da classe Actor

Componentes principais incluem:

- **RigidBodyComponent**: gerencia física, através do controle de velocidade, aceleração, aplicação de forças e gravidade;
- **AABBColliderComponent**: define volume de colisão axis-aligned, detecta interseções e resolve colisões;

- **DrawAnimatedComponent**: renderiza spritesheets animados com controle de FPS e callbacks;
- **AIMovementComponent**: controla a lógica de movimentação de inimigos, tem uma máquina de estados interna. Já foi usado com intermédio de Spatial Hashing para implementar a execução de caminhamentos em inimigos, mas esse foco foi abandonado por motivos já mencionados;
- **TimerComponent**: gerencia temporizadores para cooldowns e eventos temporais.

4.3. Modelagem Comportamental via Máquinas de Estados Finitos

Para gerenciar a complexidade do comportamento dos NPCs e a lógica de animação dos atores, o projeto utiliza o conceito de Máquinas de Estados Finitos (*Finite State Machines* - FSM) [Nystrom 2014]. Uma FSM é um modelo matemático de computação concebido como uma máquina abstrata que pode estar em exatamente um de um número finito de estados em qualquer momento determinado.

Formalmente, a implementação no jogo segue a definição de um autômato finito determinístico, representado pela quíntupla $M = (Q, \Sigma, \delta, q_0, F)$, onde:

- Q é um conjunto finito de estados (ex: `Idle`, `Patrol`, `Attack`...);
- Σ é o conjunto finito de símbolos de entrada ou eventos (ex: *player* entrou no campo de visão, fim da animação);
- $\delta : Q \times \Sigma \to Q$ é a função de transição que determina o próximo estado com base no estado atual e na entrada recebida;
- $q_0 \in Q$ é o estado inicial (geralmente `Idle` ou `Asleep`);
- $F \subseteq Q$ é o conjunto de estados finais (no contexto do jogo, estados como `Dead`).

Cada entidade inteligente possui uma variável de controle (enumeração em C++) que dita qual lógica de atualização (`Update`) deve ser executada naquele quadro. Isso permite o desacoplamento de comportamentos: um inimigo no estado `Idle` ignora as lógicas de combate até que a função de transição δ seja satisfeita (detectar o jogador), momento em que o sistema altera atomicamente o estado para `Attacking`.

Listing 2. Exemplo de FSM em inimigo

```
enum class BehaviorState {
   enum class BehaviorState // For AI behaviors/animations
{
   Asleep = 0, Waking = 1, Idle = 2, Moving = 3,
   Charging = 4, Attacking = 5, AerialAttacking = 6,
   Dodging = 7, Stunned = 8, Jumping = 9, Falling = 10,
   Dying = 11, Provoking = 12, Fleeing = 13,
   Wandering = 14, TakingDamage = 15, Clinging = 16,
   Freezing = 17, Frozen = 18, Dashing = 19, Dead = 20,
   Barking = 21, Licking = 22, ChargingAttack = 23,
   Vanishing = 24, Appearing = 25
};

void Enemy::ManageState() {
   switch (mBehaviorState) {
      case BehaviorState::Moving:
         if (PlayerOnSight(400.f) && !mProjectileOnCooldown)
```

```
            mBehaviorState = BehaviorState::Charging;
         break;
      case BehaviorState::Charging:
         if (!PlayerOnSight(400.f))
            mBehaviorState = BehaviorState::Moving;
         break;
      // ...
   }
}
```

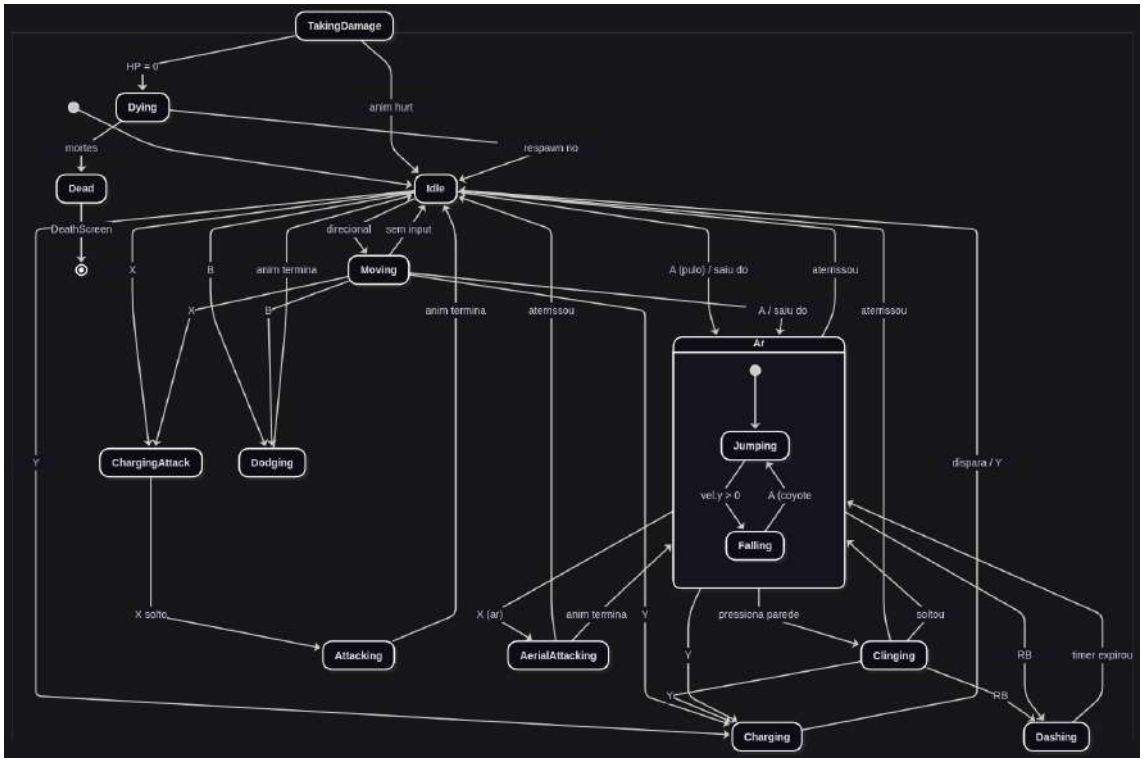

Figure 20. Diagrama de FSM simplificado para a protagonista.

`AIMovementComponent` implementa uma FSM separada para controle de movimentação (Wandering, Seeking, Patrolling), desacoplando locomoção de outros comportamentos em NPCs. Alémd isso, os sistemas `transição de cenas`, `cutscenes` e `Aúdio` também utilizam FSMs internas para controlar a progressão de eventos e passos.

4.3.1. Sistema de Eventos

O jogo utiliza callbacks para comunicação entre sistemas, evitando acoplamento direto. Exemplos incluem:

- Callbacks de fim de animação, tomada de dano, captura de item notificam mudanças de estado;

- Timers executam funções ao expirar (cooldowns, eventos de cutscene);
- Colisões invocam métodos virtuais `OnHorizontalCollision` e `OnVerticalCollision`, que invocam os identificadores de colisão dos AI movement components.

4.4. Modelagem de Dados

4.4.1. Representação de Níveis

Antes dessa seção, vale ressaltar algumas definições:

- **Tile**: unidade gráfica básica que compõe o cenário, geralmente em uma grade regular (ex: 32x32 pixels);
- **Tilemap**: mapa composto por uma grade de tiles, onde cada posição pode conter um tile específico ou estar vazia;
- **Tileset**: coleção de tiles gráficos, geralmente armazenados em um spritesheet (coleção de imagens), com informações de mapeamento para IDs de tile.

Níveis são criados em Tiled Map Editor e exportados como JSON. A estrutura contém layers de tiles, objetos e tilesets:

Listing 3. Estrutura de mapa (simplificada)

```
{
  "width": 60, "height": 60, "tilewidth": 32,
  "layers": [
    {
      "name": "ground",
      "data": [0, 0, 1, 2, 3, ...], // IDs de tiles
      "type": "tilelayer"
    },
    {
      "name": "objects",
      "objects": [
        {
          "id": 1, "x": 320, "y": 480,
          "properties": [
            {"name": "event", "value": "enter"},
            {"name": "function_name", "value": "spawn_entity"},
            {"name": "entity_code", "value": 1}
          ]
        }
      ]
    }
  ],
  "tilesets": [
    {"firstgid": 1, "source": "tileset.json"}
  ]
}
```

A classe `Map` parseia JSON, instancia `Tiles` com geometria de colisão e cria `MapObjects` que executam funções ao serem ativados (spawn de inimigos, início de cutscenes).

Existem tilesets reservados para tiles especiais, como as *Tochas*. Após o parseamento, durante o carregamento de mapas no jogo, o código identifica tiles desse tileset específicos no mapa, e ao invés de posicionar um tile comum, posiciona uma Tocha, ou qualquer outro item especial.

4.4.2. Configuração de Animações

Spritesheets utilizam descritores JSON gerados por ferramentas como [Norinchak 2025]: `DrawAnimatedComponent` carrega frames ordenados, permitindo definição de animações por intervalos, ou por frames específicos:

Listing 4. Descritor de spritesheet

```
{
  "frames": [
    {
      "filename": "idle_0.png",
      "frame": {"x": 0, "y": 0, "w": 64, "h": 64}
    },
    {"filename": "idle_1.png", ...},
    {"filename": "run_0.png", ...}
  ]
}
```

Listing 5. Exemplo de código de carregamento de animação.

```
mDrawComponent->AddAnimation("idle", 0, 8); // frames 0-8
mDrawComponent->AddAnimation("run", 9, 14); // frames 9-14
```

Listing 6. Exemplo de código de carregamento de animação.

```
mDrawComponent->AddAnimation("idle", {0, 1, 3});
```

4.4.3. Dados de Inimigos e Habilidades

Propriedades de inimigos (velocidade, cooldowns, força de knockback) são inseridas em um JSON (*config.json*) de configurações que deve estar na pasta do jogo em tempo de execução.

Listing 7. Exemplo de parte do arquivo de configuração do jogo.

```
{
    "METAL_CRATE_PUSH_FORCE": 1000,
    "ENEMY": {
        "PLAYER_KNOCKBACK_FORCE": 400,
        "FOV_ANGLE": 1.3963
    },
    "FREEZING_RATE": 1.0,
    "ZATHURA": {
        "HEALTH": 15,
        "ROCKS_ATTACK_COOLDOWN": 30.0,
        ...
    }
}
```

4.5. Fluxos Importantes

4.5.1. Game Loop

O loop principal segue estrutura clássica com delta time fixado:

Listing 8. Game loop simplificado

```
void Game::RunLoop() {
   while (mIsRunning) {
      // Limite de sessenta frames.
      while (!SDL_TICKS_PASSED(SDL_GetTicks(), mTicksCount + 16));

      float deltaTime = (SDL_GetTicks() - mTicksCount) / 1000.0f;
      deltaTime = std::min(deltaTime, 0.05f); // Max 50ms

      mTicksCount = SDL_GetTicks();

      ProcessInput();
      UpdateGame(deltaTime);
      GenerateOutput();
   }
}
```

`UpdateGame` atualiza atores visíveis na câmera (via `SpatialHashing::QueryOnCamera`), sistemas de áudio e UI. Atores marcados com `mustAlwaysUpdate` são atualizados independentemente de visibilidade (jogador, objetos de cutscene). Mas, possibilidade de atualizar atores independentemente de estarem presentes na câmera foi utilizada com cautela, para evitar um número grande de atores a atualizar por frame, o que pode diminuir a performance do jogo.

4.5.2. Detecção de Colisão

O sistema de colisão opera em duas fases:

Fase um: `SpatialHashing` divide mundo em grid de células 32x32 pixels. Atores são inseridos em células baseadas em posição, permitindo consultas diretas, em complexidade constante:

Listing 9. Função que busca por colisores em uma região.

```
std::vector<AABBColliderComponent*>
SpatialHashing::QueryColliders(const Vector2& pos,
                               int range) const {
   int col = pos.x / mCellSize;
   int row = pos.y / mCellSize;

   std::vector<AABBColliderComponent*> results;
   for (int r = row - range; r <= row + range; ++r) {
      for (int c = col - range; c <= col + range; ++c) {
         // Adiciona colliders da celula [r][c]
      }
   }
```

```
    return results;
}
```

Fase dois: `RigidBodyComponent` movimenta ator em eixos separados (X depois Y), invocando `AABBColliderComponent::DetectCollision` após cada movimento:

Listing 10. Verificação de colisão por eixo.

```
void RigidBodyComponent::Update(float deltaTime) {
    // Aplica forcas, atualiza velocidade
    mVelocity += mAcceleration * deltaTime;

    // Move eixo X
    mOwner->SetPosition(
        Vector2(mOwner->GetPosition().x + mVelocity.x * deltaTime,
              mOwner->GetPosition().y));
    collider->DetectHorizontalCollision(this);

    // Move eixo Y
    mOwner->SetPosition(
        Vector2(mOwner->GetPosition().x,
              mOwner->GetPosition().y + mVelocity.y * deltaTime));
    collider->DetectVerticalCollision(this);
}
```

Ao detectar interseção entre colisores retangulares, o sistema calcula overlap mínimo, resolve posição e zera componente de velocidade:

Listing 11. Resolução de colisão

```
void AABBColliderComponent::ResolveHorizontalCollisions(
    constexpr float epsilon = 0.001f; // Small separation buffer
    float adjustment = minXOverlap + (minXOverlap > 0 ? epsilon :-
        epsilon);
    mOwner->SetPosition(mOwner->GetPosition() - Vector2(adjustment, 0.0f
        ));
    rigidBody->ResetVelocityX();
}
```

Colisores podem ignorar outros colisores específicos (inimigos ignoram projéteis de outros inimigos). Essa configuração pode ignorar tanto a resolução de colisão, quanto os efeitos da mesma (dano, knockback).

Resolução da colisão significa reposicionar colisores para que não haja mais interseção, e zerar a velocidade do ator na direção da colisão. Ignorar colisão implica que o sistema de detecção de colisão não irá considerar colisores específicos, ou seja, eles poderão se sobrepor sem serem reposicionados ou terem suas velocidades zeradas.

Os efeitos de colisão são aplicados pelos *callbacks* de colisão, que são invocados quando a colisão é detectada. Por exemplo, quando um inimigo colide com o jogador, o callback de colisão do inimigo é invocado, e ele aplica dano ao jogador e aplica knockback.

Além disso, colisores podem ter tangibilidade desabilitada (zonas de gatilho ou colisores de ataques). Com tangibilidade ligada, colisores são intransponíveis. Desligar tangilibilidade remove toda e qualquer resolução de colisão, mas ainda permite detecção de colisão e invocação de callbacks.

4.5.3. Física Customizada

`RigidBodyComponent` implementa integração de Euler com gravidade, fricção e forças externas:

```
void RigidBodyComponent::Update(float deltaTime) {
   if (mApplyGravity)
      ApplyForce(Vector2(0.f, 980.0f)); // Gravity

   if (mApplyFriction && mIsOnGround)
      ApplyForce(Vector2(-mFrictionCoefficient * mVelocity));

   mVelocity += mAcceleration * deltaTime;

   // Clamp velocity
   mVelocity.x = Clamp(mVelocity.x, -750.f, 750.f);
   mVelocity.y = Clamp(mVelocity.y, -750.f, 750.f);

   // Move and detect collision (codigo anterior)
}
```

Determinadas mecânicas aplicam forças iterativas, aplicadas de maneira sucessiva a cada frame, através de `ApplyForce`. Outras precisam aplicar movimentos em um único frame, por exemplo o *knockback* (impulso para uma direção dado um colisão). Essas habilidades alteram a velocidade do ator diretamente, por meio de ApplyImpulse. Ao final de todo frame, a aceleração é resetada para zero, para que forças sejam reaplicadas no próximo frame.

```
void RigidBodyComponent::ApplyForce(const Vector2 &force) {
   mAcceleration += force * (1.f/mMass);
}

void RigidBodyComponent::ApplyImpulse(const Vector2 &impulse) {
   mVelocity += impulse * (1.f/mMass);
}
```

4.5.4. Animações

Animações são sequências de frames renderizados em ordem, controlados por um timer que avança com base no FPS (frames por segundo) definido para a animação. `DrawAnimatedComponent` avança timer baseado no FPS, e indice (sprite corrente) da animação. Além disso, gatilhos de fim de animação permitem sincronização com lógica (transição de estado após animação de ataque, destruição após animação de morte).

```
void DrawAnimatedComponent::Update(float deltaTime) {
```

```
    if (mIsPaused) return;

    mAnimTimer += mAnimFPS * deltaTime;

    if (mAnimTimer >= mAnimations[mAnimName].frames.size()) {
        if (mAnimationEndCallback)
            mAnimationEndCallback(mAnimName);

        if (mAnimations[mAnimName].isLoop)
            mAnimTimer = 0.0f;
        else
            mAnimTimer = frames.size() - 1; // Hold last frame
    }
}
```

4.6. Spatial Hashing

Spatial Hashing é a estrutura de dados central do jogo para consultas espaciais eficientes. Ela divide o mundo em uma grade regular de células quadradas de 32×32 pixels, o mesmo tamanho de um tile, e mantém dois planos sobrepostos sobre essa grade:

- **`mGrid`**: vetor tridimensional de atores que armazena os atores presentes em cada célula;
- **`mCellTypes`**: vetor bidimensional de `CellType` que classifica a topologia de cada célula para fins de navegação de IA.

Dois índices inversos complementam a estrutura: `mPositions` e `mCellIndices` mapeiam cada ator à sua posição e célula atuais, tornando as operações `Remove` e `Reinsert` constantes sem varredura.

4.6.1. Classificação de células

O enumerador `CellType` categoriza cada célula em quatro tipos:

```
enum class CellType {Empty, Tile, Platform, Corner };
```

Quando um ator do tipo tile é inserido, sua célula é marcada como `Tile` e os vizinhos imediatos são reclassificados automaticamente: a célula diretamente acima passa a `Platform` (superfície de pouso acessível para a IA), e as células diagonalmente acima dos extremos passam a `Corner` (bordas de plataforma). A remoção de um tile reverte as reclassificações dependentes.

Além de `Tiles` propriamente ditos, são tratados como ocupantes do tipo tile para fins de classificação: `Spikes`, `Spear`, `Shuriken` e colisores `EnemyBlocker`. Isso garante que armadilhas e barreiras de IA façam parte do mapa de navegação sem configuração adicional.

4.6.2. Consultas espaciais

O sistema expõe três formas de consulta:

- **`Query(pos, range)`**: retorna todos os atores nas células dentro de `range` células ao redor de `pos`,
- **`QueryColliders(pos, range)`**: variante filtrada para atores com `AABBColliderComponent`, usada pelo sistema de detecção de colisão na fase broadphase;
- **`QueryOnCamera(camPos, w, h)`**: retorna todos os atores contidos na janela da câmera, com alcance configurável. Esta é a consulta usada pelo game loop para determinar quais atores atualizar por frame, atores fora da câmera e não marcados com um flag especial não são processados, o que constitui a principal otimização de desempenho do jogo.

4.6.3. Duplo uso: colisão e navegação

A mesma estrutura serve dois subsistemas distintos. Para a detecção de colisão, `QueryColliders` fornece o conjunto candidato de colisores próximos antes da segunda fase, eliminando comparações desnecessárias entre atores distantes.

Para a IA, `mCellTypes` é o grafo de navegação consumido pelo algoritmo A*: células `Tile` são nós inválidos, `Platform` e `Corner` têm custos diferenciados por tipo de inimigo, e `Empty` tem custo alto para que inimigos terrestres permaneçam naturalmente sobre superfícies. Essa dualidade significa que a inserção de qualquer tile mantém ambos os planos sincronizados simultaneamente, sem etapas intermediárias ou estruturas separadas para cada subsistema.

4.7. Algoritmo A*

A navegação de entidades autônomas em *ASTRAL* era fundamentada no algoritmo A* (A-Star). Atualmente, ele não é mais utilizado para movimentação de inimigos, por questões de responsividade e rigidez. Essa técnica consiste na busca heurística que determina o caminho de menor custo entre dois pontos em um grafo. Formalmente, o ambiente de jogo é modelado como um grafo $G = (V, E)$, onde V representa o conjunto de posições acessíveis e E as conexões válidas entre elas.

No contexto desta implementação, o sistema de *Spatial Hashing* atua não apenas como otimizador de colisão, mas como a estrutura de dados que discretiza o espaço contínuo em uma representação matricial navegável. O grid de células gerado pelo hashing pode ser interpretado como um grafo regular onde cada célula $c_{i,j}$ (nó) possui arestas implícitas conectando-a aos seus vizinhos ortogonais (cima, baixo, esquerda, direita), desde que estes não sejam obstáculos.

O algoritmo seleciona o caminho explorando nós com base em uma função de avaliação $f(n)$, que estima o custo total de um caminho passando pelo nó n. Esta função é definida pela equação:

$$f(n) = g(n) + h(n) \tag{1}$$

Onde:

- $g(n)$ é o custo real do caminho desde o nó inicial até o nó n. Em um grid uniforme, este custo é proporcional à distância percorrida;

- $h(n)$ é a heurística, uma estimativa do custo mais barato de n até o objetivo.

Para garantir a eficiência e a admissibilidade da busca em um ambiente de plataforma 2D, utiliza-se a Distância de Manhattan como função heurística, dada por:

$$h(n) = |n_x - \text{goal}_x| + |n_y - \text{goal}_y| \tag{2}$$

Essa abordagem permitia que os inimigos (como o *Sith* ou *Zod*) calculassem trajetórias dinâmicas em tempo real, adaptando-se à geometria do nível mapeada pelo *Spatial Hashing*, contornando obstáculos estáticos e plataformas para alcançar o jogador.

As alternativas para cálculo de caminho do A* são computacionalmente pouco viáveis para jogos digitais, que precisam atualizar dezenas de vezes por segundos. Ou seja, toda a computação realizada dentro de um loop deve se passar em menos de uma fração de segundo (comumente 1/60, como citado anteriormente).

O A* usa uma fila de prioridades, como será mostrado abaixo, para expandir primordialmente nos caminhos que considera mais próximos do objetivo pela heurística. Ao contrário de algoritmos como Dijkstra, que expande pelo grafo de caminhos completamente antes de terminar de executar.

4.7.1. Pathfinding com A*

`SpatialHashing` mantém grid de `CellType` (Empty, Tile, Platform, Corner) atualizado dinamicamente conforme tiles são adicionados/removidos. `AIMovementComponent` solicita caminho fornecendo posição inicial e alvo:

```
std::vector<Cell> SpatialHashing::findPath(
   const std::vector<std::vector<CellType>>& grid,
   Cell start, Cell end,
   std::function<float(int,int)> getCellCost,
   std::function<bool(int,int)> isValid,
   std::function<int(Cell,Cell)> heuristic,
   int maxJumpHeight) const {

   std::priority_queue<Node> open; // Min-heap por f-score
   std::unordered_map<Cell, Cell> cameFrom;
   std::unordered_map<Cell, float> gScore;
   std::unordered_map<Cell, int> upwardCount;

   open.push({start, heuristic(start, end), 0});
   gScore[start] = 0;
   upwardCount[start] = 0;

   while (!open.empty()) {
      Node curr = open.top(); open.pop();

      if (curr.pos == end) {
         // Reconstruir caminho
      }

      // Explorar vizinhos (4-direcoes)
      for (auto dir :directions) {
```

```
            Cell neighbor = {curr.pos.row + dir.x,
                             curr.pos.col + dir.y};

            int cellUpwards = upwardCount[curr.pos];
            if (dir.y == -1) {// Moving up
                cellUpwards++;
                if (cellUpwards > maxJumpHeight) continue;
            }else if (grid[neighbor] == Platform) {
                cellUpwards = 0; // Reset on landing
            }

            float newG = curr.g + getCellCost(neighbor);

            if (!gScore.count(neighbor) || newG < gScore[neighbor]) {
                gScore[neighbor] = newG;
                upwardCount[neighbor] = cellUpwards;
                cameFrom[neighbor] = curr.pos;
                open.push({neighbor, newG + heuristic(neighbor, end),
                        newG});
            }
        }
    }
}
```

Diferentes tipos de inimigos têm diferentes funções de custo. Elas diferenciam os custos dos tipos de célula (Platform, Corner, Empty) nós cálculos de caminho, permitindo que inimigos terrestres sempre tenham que se guiar nas plataformas, enquanto voadores se movimentam em qualquer direção sem limitações.

4.8. IA de inimigos

A inteligência artificial dos inimigos em *ASTRAL* é organizada em duas camadas independentes: a **percepção**, que define o que o inimigo consegue enxergar do mundo, e a **movimentação**, que traduz essa percepção em ação física. Essa separação facilita a criação de inimigos com personalidades distintas sem duplicação de lógica.

4.8.1. Percepção

Cada inimigo possui dois mecanismos de detecção do jogador, inspirados no conceito humano de visão periférica versus foco direto.

O **campo de visão** (*Field of View*) determina se o jogador está dentro de um cone angular à frente do inimigo. Para isso, o sistema calcula o ângulo entre a direção de frente do inimigo e o vetor até o jogador usando o produto escalar. Além do cone, existe uma distância mínima em que o inimigo detecta o jogador independentemente de direção, como um espaço pessoal:

Listing 12. Verificação de campo de visão

```
bool Enemy::PlayerOnFov() {
    Vector2 toZoe = zoe->GetCenter() - GetCenter();
    float distSq = GetDistanceToPlayerSquared();

```

```
    if (distSq > mMaxSeeDistance * mMaxSeeDistance) return false;

    float dot = Vector2::Dot(GetForward(), toZoe.Normalized());
    float angle = Math::Acos(dot);

    bool isInFov = angle < mGame->GetConfig()->Get<float>("ENEMY.
        FOV_ANGLE");
    bool withinMinDistance = distSq < mMinSeeDistance * mMinSeeDistance;

    // Valida que nao ha obstaculo entre o inimigo e o jogador
    return (isInFov || withinMinDistance) &&
          mColliderComponent->IsSegmentIntersectingPlayerLayer(
             GetCenter(), zoe->GetCenter());
}
```

A segunda verificação, `IsSegmentIntersectingPlayerLayer`, lança um segmento de reta entre o inimigo e o jogador e verifica se algum colisor o intercepta. Isso garante que paredes bloqueiam a visão: um inimigo atrás de um muro não sabe que o jogador está do outro lado.

4.8.2. Movimentação: ciclo Perceber-Planejar-Agir

O `AIMovementComponent` organiza sua atualização em três etapas executadas a cada frame, um padrão clássico de IA para agentes autônomos [Gregory 2018]:

- **Perceber** (*Sense*): coleta obstáculos perigosos próximos (armadilhas, espinhos) via `SpatialHashing`, armazenando seus centros para uso posterior;
- **Planejar** (*Plan*): decide em qual estado de movimentação permanecer ou transitar, com base no tempo e em um fator de aleatoriedade (`mCraziness`);
- **Agir** (*Act*): aplica a força ou velocidade resultante ao `RigidBodyComponent` do inimigo.

O componente mantém três estados de movimentação internos:

Listing 13. Estados de movimentação da IA

```
enum class MovementState {
    Wandering, // Vaga pelo nivel sem objetivo
    Seeking, // Persegue a ultima posicao conhecida do jogador
    Patrolling // Retorna ao ponto de origem
};
```

Quando o inimigo detecta o jogador via FOV, a camada de comportamento chama `SeekPlayer()`, que altera o estado para `Seeking`. Ao perder o jogador de vista, transita para `Patrolling`, retornando ao ponto de surgimento. O inimigo lembra a última posição vista do jogador (`mLastSeenPlayerCenter`) e se move até ela antes de perder o jogador de vista completamente, um comportamento de memória curta que torna a perseguição mais crível.

Listing 14. Movimento no estado de perseguição

```
case MovementState::Seeking: {
    if (CanBePressingPlayerAgainstWall()) {
```

```
        dir = Vector2(0.f, 0.f); // Para de empurrar jogador contra
            parede
        break;
    }
    Vector2 toPlayer = mOwnerEnemy->GetLastSeenPlayerCenter()
                    - mOwner->GetCenter();

    if (mOwnerEnemy->GetLastSeenPlayerDistanceSquared() < 400.f) break;

    toPlayer.Normalize();
    dir = toPlayer;
    break;
}
```

Aqui é importante notar a função `CanBePressingPlayerAgainstWall`, que verifica se o inimigo está empurrando o jogador contra uma parede, o que pode resultar em uma situação de *softlock* ou *hardlock*, onde o jogador não consegue se mover. Se essa condição for detectada, o inimigo para de aplicar força nessa direção, permitindo que o jogador se liberte.

Além disso, esse é o sistema que substituiu o A* para movimentação de inimigos. Ele é mais simples, mas produz um comportamento mais fluido e adaptativo, do qual o jogador pode se adaptar, e os inimigos parecem menos mecânicos e mais naturais.

4.8.3. Desvio de obstáculos

Quando armadilhas são detectadas na etapa de percepção, o componente aplica um vetor de repulsão acumulado, cada obstáculo desloca o inimigo para longe proporcionalmente à proximidade. Esse vetor é somado à direção principal antes de ser aplicado, resultando em um desvio suave sem lógica de caminhamento:

Listing 15. Acumulação de forças de repulsão

```
for (const auto& obstacleCenter :mObstaclesAroundCenters) {
    Vector2 away = mOwner->GetCenter() - obstacleCenter;
    float norm = away.Length() + 0.001f;
    float modifier = (limit - norm);
    if (modifier < 0) modifier = 0.f;

    avoidDir += (away * (1.f/norm)) * modifier;
}
dir += avoidDir;
dir.Normalize();
```

4.8.4. Tipos de movimento

Inimigos terrestres (`Walker`) recebem apenas força horizontal, deixando a gravidade gerenciar o eixo vertical. Inimigos voadores (`Flier`) controlam velocidade diretamente nos dois eixos, ignorando gravidade e se movendo em linha reta em direção ao jogador. Essa distinção, configurada na instanciação de cada inimigo, produz perfis de comportamento radicalmente diferentes com a mesma lógica de decisão.

4.9. Câmera

Em *ASTRAL*, a câmera não é uma entidade independente: ela é representada por um único vetor `mCameraPos`, que indica o canto superior esquerdo da área visível no mundo. Todo elemento renderizado subtrai esse valor para converter sua posição de mundo em posição de tela. O sistema oferece quatro modos de funcionamento, trocáveis em tempo de execução:

- **Zoe**: a câmera centraliza continuamente na protagonista. É o modo padrão durante o gameplay;
- **Point**: a câmera aponta para uma posição fixa configurável (`mCameraCenterPos`), não utilizado no escopo do jogo. Mas poderia ser, especificamente em cutscenes para direcionar o olhar do jogador a pontos específicos do mapa;
- **LogicalWindowSizeCenter**: divide o mundo em blocos do tamanho da janela (640×352 pixels) e centraliza na seção onde o jogador se encontra, criando um comportamento de sala a sala em vez de seguimento contínuo;
- **Shaking**: idêntico ao modo anterior, mas sobrepõe um deslocamento aleatório por frame dentro de um raio de intensidade configurável, produzindo o efeito de tremor de tela (*screen shake*).

Em todos os modos, a posição final é travada nos limites do mapa (via `std::clamp`), impedindo que a câmera exiba áreas vazias além das bordas. O deslocamento do *shake* é aplicado após esse travamento, garantindo que o efeito seja visível mesmo nas extremidades do nível.

4.9.1. Screen Shake como feedback

O tremor de tela é acionado pela função `SetCameraCenterToShake(duration, intensity)`, chamada em resposta a eventos de impacto:

Listing 16. Ativação do shake em resposta a dano e morte

```
// Zoe toma dano
void Zoe::OnDamageCallback() {
   mGame->SetCameraCenterToShake(0.125f, 1.5f);
}

// Zoe morre e retorna ao checkpoint
void Zoe::Kill() {
   mGame->SetCameraCenterToShake(0.4f, 5.f);
}
```

A intensidade escala com a gravidade do evento (feedback efetivo): dano leve gera tremor sutil (1.5), morte gera tremor forte (5.0). Habilidades como Ventania e Bola de Fogo também acionam o efeito, reforçando o *game feel* de peso e impacto. O sistema de cutscenes pode acionar o shake de forma scriptada via `ShakeCameraStep`, desacoplando o efeito da lógica da protagonista.

4.10. Transição de Cenas

O gerenciamento de transição de cenas implementa outra FSM com estados Entering (fade-out), Active (transição) e None (cena estável):

```
void Game::SetGameScene(GameScene scene, float delay) {
   mNextScene = scene;
   mSceneManagerState = SceneManagerState::Entering;
   mSceneManagerTimer = delay;
}

void Game::UpdateSceneManager(float deltaTime) {
   if (mSceneManagerState == SceneManagerState::Entering) {
      mSceneManagerTimer -= deltaTime;
      if (mSceneManagerTimer <= 0.0f) {
         mSceneManagerState = SceneManagerState::Active;
         mSceneManagerTimer = 0.f;
      }
   }else if (mSceneManagerState == SceneManagerState::Active) {
      mSceneManagerTimer += deltaTime;
      if (mSceneManagerTimer >= TRANSITION_TIME) {
         ChangeScene(); // Unload current, load next
         mSceneManagerState = SceneManagerState::None;
      }
   }
}
```

Durante transição, um retângulo preto é renderizado com opacidade crescente, criando a sensação de *fading* entre cenas. A função `ChangeScene` deleta todos atores, limpa spatial hash, redefine câmera e invoca função de load específica (LoadFirstLevel, LoadMainMenu).

Isso é necessário por questões de otimização mediante liberação de memória. Novas cenas vão operar com novas entidades, atores e componentes, e os anteriores podem ser liberados, para evitar sobrecarga.

4.11. Sistema de Áudio

O sistema de áudio é gerenciado pela classe `AudioSystem`, que encapsula a biblioteca `SDL_mixer`. Sons são divididos em duas categorias funcionais: música de fundo (`.ogg`, reproduzida em loop) e efeitos sonoros — SFX (`.wav`, para descompressão rápida sem custo de decodificação por frame).

4.11.1. Gerenciamento de Canais

`SDL_mixer` opera com um número fixo de canais de mixagem, inicializado com 8 canais:

Listing 17. Inicialização do sistema de áudio

```
Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 2048);
Mix_AllocateChannels(numChannels);
```

Cada som em reprodução ocupa um canal. Ao solicitar `PlaySound()`, o sistema busca um canal com a seguinte ordem de prioridade:

1. Canal completamente livre;
2. Canal tocando o mesmo som solicitado (evita sobreposição do mesmo SFX);
3. Canal com som em loop (música de fundo é interrompida por SFX se necessário);
4. Canal mais antigo em uso (último recurso).

Essa política garante que novos SFX nunca falhem silenciosamente por falta de canal, ao custo de eventualmente interromper sons menos prioritários.

4.11.2. Carregamento e Cache

Sons são carregados sob demanda na primeira chamada a `PlaySound()` e armazenados em um `unordered_map` interno. Chamadas subsequentes ao mesmo arquivo retornam o `Mix_Chunk` já em memória, sem I/O adicional. Na inicialização de cada cena, `CacheAllSounds()` pré-carrega todos os arquivos da pasta `Assets/Sounds/`, eliminando pausas de carregamento durante o gameplay.

4.11.3. SoundHandle e Controle de Reprodução

`PlaySound()` retorna um `SoundHandle` — um identificador inteiro único — que permite controle posterior sobre o som específico:

```
SoundHandle handle = mGame->GetAudio()->PlaySound("portalAmbient.wav",
    true);
// ...
mGame->GetAudio()->StopSound(handle);
```

Sons sem handle (a maioria dos SFX pontuais) são disparados e esquecidos. O volume global é controlado via `SetMasterVolume(0.0f--1.0f)`, mapeado internamente para a faixa 0–128 do `SDL_mixer` e aplicado a todos os canais ativos.

4.12. Cutscenes

Conforme introduzido na Seção 2.1.2, cutscenes são sequências pré-configuradas que controlam atores, câmera e sistemas do jogo. Em *ASTRAL*, elas são implementadas como um **sequenciador de passos**: a classe `Cutscene` mantém um `vector<unique_ptr<Step>>` e um índice do passo atual. Ao ser iniciada, muda o `GamePlayState` para `PlayingCutscene`, bloqueando o input normal do jogador. Ao término do último passo, restaura o estado anterior e dispara um callback opcional.

O loop de execução é simples: a cada frame, `Update()` é chamado no passo atual. Quando o passo sinaliza conclusão (`IsComplete()`), o sequenciador avança para o próximo e chama `PreUpdate()` nele antes do primeiro frame de execução.

4.12.1. Anatomia de um Step

A classe base `Step` inclui um `maxTime` opcional: passos com tempo limite se completam automaticamente ao expirar, sem lógica adicional nas subclasses. Cada `Step` expõe três momentos de ciclo de vida:

- **`PreUpdate()`**: executado uma única vez quando o passo começa, usado para setup (buscar ator alvo, desabilitar gravidade, iniciar animação);
- **`Update(dt)`**: chamado a cada frame enquanto o passo está ativo;
- **`SetComplete()`**: sinaliza fim e realiza cleanup (reabilitar gravidade, liberar recursos).

4.12.2. Tipos de Steps

Os steps disponíveis cobrem quatro categorias funcionais:

- **Movimentação**: `MoveStep` desloca qualquer ator até uma posição alvo via `ApplyImpulse`, com suporte a desligamento temporário de gravidade. Steps como `JumpStep`, `VentaniaStep` e `DodgeStep` simulam inputs da protagonista programaticamente, chamando os mesmos métodos `OnPressed`/`OnReleased` que o jogador acionaria;
- **Controle do mundo**: `SpawnStep` instancia atores em posições específicas; `BreakTileStep` localiza um tile via `SpatialHashing`, tremula-o por N frames e o destrói; `SetBehaviorStateStep` e `ApplyKnockbackStep` manipulam estado e física de qualquer ator;
- **Controle de sistemas**: `FreezePhysicsStep` e `UnfreezePhysicsStep` congelam e descongelam a física do jogo inteiro; `ShakeStep` aciona o tremor de câmera; `SoundStep` toca um som e completa imediatamente;
- **Interação com o jogador**: `DialogueStep` inicia o sistema de diálogos (detalhado a seguir); `SpawnJoystickButtonStep` exibe uma animação de botão no HUD e aguarda o jogador pressioná-lo, implementando prompts de tutorial interativos dentro de cutscenes.

O exemplo abaixo mostra a cutscene disparada pelo Zod ao detectar o primeiro projétil em direção ao jogador, combinando congelamento de física, diálogo, prompt de botão e mecânica de esquiva:

Listing 18. Montagem de cutscene para tutorial de esquiva

```
steps.push_back(std::make_unique<DialogueStep>(
    mGame, "Zoe",
    std::vector<std::string>{"Esse tiro esta vindo na minha direcao."}))
        ;

steps.push_back(std::make_unique<FreezePhysicsStep>(mGame));
steps.push_back(std::make_unique<SpawnJoystickButtonStep>(mGame, Button
    ::B));
steps.push_back(std::make_unique<DodgeStep>(mGame));
steps.push_back(std::make_unique<UnfreezePhysicsStep>(mGame));
```

4.12.3. Diálogos

O `DialogueSystem` é o componente responsável por exibir caixas de texto durante cutscenes e momentos narrativos. Ele recebe um `vector<string>` de linhas e um callback de conclusão, e gerencia a progressão entre elas.

A integração com o sistema de cutscenes ocorre via `DialogueStep`: ao iniciar, o step chama `StartDialogue()` ou `StartDialogueWithSpeaker()`, que muda o `GamePlayState` para `Dialogue`. Ao fim de todas as linhas, `GoBackToPreviousGameState()` restaura `PlayingCutscene` e o callback marca o step como completo, retomando o sequenciador normalmente.

A progressão entre linhas acontece de duas formas: o jogador pressiona o botão B (ou Enter), ou um timer automático de 15 segundos avança a linha sozinho. Uma barra de progresso acima da caixa de texto comunica visualmente o tempo restante para o avanço automático. A cada linha avançada, o som `dialogueStep.wav` é tocado.

Figure 21. Caixa de diálogo com nome do falante e barra de progresso.

O diálogo é renderizado na resolução real da janela, não na resolução lógica de 640×352 pixels. Isso evita que o upscaling de pixel art aplique interpolação sobre o texto, que exige nitidez independente do tamanho de tela.

4.13. Objetos de Mapa

Objetos de mapa (`MapObject`) são gatilhos invisíveis definidos diretamente no Tiled. Qualquer retângulo no layer `objects` de um mapa vira um `MapObject` com colisão intangível, configurado por três propriedades:

- **`event`**: quando disparar — `enter`/`exit` (ao cruzar a borda), `in`/`out` (todo frame dentro/fora), `contains` (jogador inteiramente dentro), `closeToCenter` (distância parametrizável) ou `atStart` (na construção, independente do jogador);
- **`function_name`**: o que executar;
- **parâmetros extras**: dependem da função (ex: nome da cutscene, código da entidade, distância).

As funções disponíveis são `play_cutscene` (inicia uma cutscene pelo nome e se auto-destrói), `spawn_entity` (instancia qualquer entidade do jogo pelo código e se auto-destrói), `teleport_to_checkpoint` e `teleport_to_checkpoint_if_damaged`.

Esse mecanismo é a ponte entre o Tiled e o código: level design é gerenciado quase completamente no tiled, permitindo geração de inimigos, início de cutscenes e zonas de recuperação sem tocar em C++, apenas posicionando retângulos com propriedades no editor.

4.13.1. Colisores de Fronteira

Além dos objetos de mapa, o sistema de carregamento reconhece dois layers especiais do Tiled que criam colisores invisíveis sem comportamento associado:

- **`en_collider_objects`**: colisores na layer `EnemyBlocker`, que bloqueiam inimigos mas ignoram o jogador. Usados para confinar NPCs a regiões específicas do nível sem criar barreiras visíveis;
- **`player_collider_objects`**: colisores na layer `Blocks`, que bloqueiam o jogador. Usados para criar limites invisíveis em bordas de nível, ou em cenários com obstáculos de geometria complexa.

4.14. Items

Items são objetos interativos distribuídos pelos níveis que ensinam mecânicas ao jogador e avançam a narrativa. Cada tipo é instanciado por uma função estática (`Item::CreateFireballItem`, `CreateVentaniaItem`, etc.), que encapsula sprite, colisão e um callback de coleta (`PickHandler`).

Quando Zoe se aproxima, o item exibe um prompt de botão acima de si (via `DrawAnimatedComponent`) e bloqueia esse botão na protagonista, impedindo que outra ação conflite com a coleta. Ao pressionar o botão correto, `OnPick()` é chamado. Por sua vez, e le toca o som de coleta, executa o `PickHandler`, destrói o item (quando configurado para isso) e dispara uma cutscene, por exemplo:

- **Narrativos** (livro, foto, geladeira): um único `DialogueStep` com texto contextual. Sem mecânica nova, apenas worldbuilding;
- **Tutorial de habilidade** (bola de fogo, ventania): sequência de steps que demonstra a habilidade na prática, simulando mecânica via step dedicado e diálogo narrativo;
- **Encadeamento** (nevasca): a cutscene de coleta dispara um callback que monta e inicia uma segunda cutscene, permitindo narrativas em dois atos dentro do mesmo fluxo de aquisição.

4.15. Checkpoints

Um checkpoint é um ponto de retorno que o jogador pode alcançar para salvar seu progresso. O sistema de checkpoints baseia-se em um vetor (`mCurrentCheckpoint`), que marca o ponto de retorno atual, e um contador de mortes (`mDeaths`). Eles são ativados de duas formas:

A primeira é por colisão com uma `Torch`, o ator de tocha presente nos níveis, que ao detectar contato com o jogador chama `SetCheckpoint()` silenciosamente, sem feedback adicional.

A segunda é via zonas de gatilho do Tiled (`MapObject`), que podem acionar `teleport_to_checkpoint` ou `teleport_to_checkpoint_if_damaged` em resposta a eventos de área (`enter`, `contains`, etc.), usadas principalmente para recuperar o jogador após quedas em buracos.

Ao morrer, `Zoe::Kill()` verifica o contador de mortes contra `ZOE.MAX_DEATHS` (lido do `config.json`). Se o limite for atingido ou não

houver checkpoint, o estado muda para `Dead` e a tela de game over é exibida. Caso contrário, Zoe é reposicionada no checkpoint com vida completa e o jogo continua.

4.16. Armadilhas

O jogo conta com três tipos de armadilhas estáticas distribuídas pelos níveis: **Spikes** (espinhos), **Spear** (lança) e **Shuriken** (estrela giratória). Todas ignoram ataques do jogador na camada de colisão, não podem ser destruídas, apenas evitadas. Também causam dano em inimigos, criando oportunidades de combate ambiental.

Spikes e Spear seguem o mesmo padrão: um `TimerComponent` agenda periodicamente a transição para `BehaviorState::Attacking`, que ativa a animação de ataque e habilita o colisor de dano. Ao fim da animação, um callback desabilita o colisor e reagenda o timer. O colisor de base permanece sempre ativo, servindo como superfície sólida para o jogador se apoiar. O Shuriken é mais simples: gira continuamente sem estado de ataque, funcionando como obstáculo permanente.

5. Arte, Áudio e Interface com o Jogador

A identidade visual e sonora de *ASTRAL* foi concebida para suportar a narrativa de exploração espacial e descoberta. Dado o foco deste trabalho na implementação de algoritmos e mecânicas, a produção artística baseou-se na curadoria, edição e criação de assets pontuais e integração de ativos de comunidade (*third-party assets*), selecionados para compor uma estética coerente e funcional.

5.1. Direção de Arte

O jogo adota o estilo *Pixel Art*, uma escolha que remete aos clássicos jogos de plataforma da era 16-bits, alinhando-se à expectativa do público-alvo de jogos *indie*. A direção visual busca estabelecer uma atmosfera cósmica e etérea.

A paleta de cores prioriza o contraste entre o plano de fundo e os elementos interativos. Os cenários utilizam tons frios e escuros (azuis profundos, roxos e preto) para representar a vastidão do espaço e o mistério do *Espaço Astral*.

Vários dos recursos gráficos, incluindo *sprites* de personagens, *tilesets* de ambiente e efeitos visuais, foram obtidos em repositórios de ativos de jogos (como [Corcoran 2025]), sob licenças permissivas (Creative Commons 0, ou licenças livres para uso não comercial ou licenças de uso livre). A lista completa de créditos e referências aos autores originais encontra-se detalhada no Documento de Design de Jogo (GDD) [Oliveira 2026a].

5.2. Interface e UX (User Experience)

A Interface de Usuário (UI) foi projetada seguindo princípios de minimalismo, visando não obstruir a área de jogo nem quebrar a imersão.

5.2.1. HUD (Heads-Up Display)

O HUD, implementado pela classe `HUD`, fornece feedback constante sobre o estado vital da personagem. Ele é composto por:

- **Barra de vida**: Uma representação visual da saúde de Zoe, atualizada dinamicamente conforme a personagem sofre dano ou recupera vida;
- **Barra de mana**: Uma representação visual da mana de Zoe, atualizada dinamicamente conforme a personagem utiliza habilidades ou recupera mana;
- **Indicadores de carregamento**: Elementos visuais que indicam o carregamento de habilidades;
- **Barra de vida de Zathura**: Zathura é o chefe final do jogo, e sua barra de vida é exibida no topo da tela durante o confronto, fornecendo feedback visual sobre o progresso do combate.

Para a tipografia, majoritariamente diálogos, utilizaram-se as fontes `VT323` e `SMB`, que reforçam a estética pixelada e garantem legibilidade em baixas resoluções, essenciais para a manutenção da proporção visual do jogo.

5.2.2. Menus

Menus são comumente utilizados para expor sistemas configuráveis do jogo de forma simples, ou para telas de transição entre fases. No caso deste trabalho, o menu é propositalmente simples, pois o jogo tem escopo reduzido. *ASTRAL* conta somente com o menu principal, que é acessado na tela de título. Ele é composto somente pelo botão de "Jogar", e a interação é feita via controle.

5.2.3. Feedback Visual e Sonoro

Para compensar a abstração do pixel art, o jogo emprega feedbacks visuais intensos para ações do jogador:

- **Dano**: Quando o jogador ou inimigos sofrem dano, eles tem um animação própria, um som é tocado no jogo (no caso de dano ao jogador), knockback é aplicado, e um período de invencibilidade temporária se inicia para aquela entidade;
- **Transições**: O uso de *fades* (escurecimento gradual) suaviza a troca entre menus e níveis, evitando cortes abruptos.
- **Tremor de tela**: O efeito de *screen shake* é aplicado em eventos de impacto, como dano ou morte, reforçando a sensação de peso e intensidade da ação.
- **Hit Blocks**: Quando um ataque do jogador atinge um inimigo em um estado não atingível, um *hit block* (um pequeno efeito visual de impacto com som) é exibido para comunicar que o ataque foi registrado, mesmo sem causar dano, reforçando o feedback de combate.
- **Item Pickup**: Ao coletar um item, um som específico e, em certos casos, animações são acionadas, reforçando o efeito da ação.
- **Checkpoints**: Ao retornar para o último checkpoint após morte, um leve tremor de tela é acionado, e um som específico é tocado.

5.3. Áudio e Trilha Sonora

A identidade sonora de *ASTRAL* foi pensada para reforçar a atmosfera de exploração espacial e amplificar o *game feel* das mecânicas.

5.3.1. Ambientação e Música

A trilha sonora é composta por faixas em formato `.ogg` que tocam em *loop*, definindo o tom emocional de cada cena:

- **Exploração**: Temas calmos e misteriosos para os níveis de plataforma;
- **Tensão**: Temas mais acelerados ou graves para telas de *Game Over* ou encontros perigosos.

5.3.2. Efeitos Sonoros (SFX)

Os efeitos sonoros, armazenados em formato `.wav` para descompressão rápida, desempenham papel crucial no *game feel* e no feedback de mecânicas. Estão organizados nas seguintes categorias:

- **Combate**: Sons distintos para o disparo da Bola de Fogo (`fireball.wav`), impacto de ataques corpo a corpo e *hit block*, efeito sonoro tocado quando um ataque acerta um inimigo em estado não atingível, comunicando o registro do golpe sem dano;
- **Movimentação**: Sons de passos durante deslocamento e ativação da habilidade Ventania (`ventania.wav`), confirmando a execução dos comandos de entrada;
- **Interação com itens**: Ao coletar qualquer item, `PickItem.wav` é tocado. Itens de aquisição de habilidade disparam sons adicionais associados à cutscene de tutorial;
- **Dano e respawn**: Ao sofrer dano, um som de impacto é tocado para o jogador. Ao morrer e retornar ao checkpoint, `respawn.wav` confirma o retorno. O mesmo som é usado na teleportação direta ao checkpoint via zonas de recuperação;
- **Diálogos**: A cada linha avançada no sistema de diálogos, `dialogueStep.wav` é tocado, cadenciando a leitura;
- **Ambiente**: Sons contínuos como `portalAmbient.wav` fornecem pistas espaciais sobre a localização de objetivos no nível.

6. Implementação, Testes e Avaliação

Esta seção detalha o processo de materialização do projeto *ASTRAL*, descrevendo o ambiente de desenvolvimento, os desafios técnicos enfrentados na evolução do código base e a metodologia utilizada para a validação preliminar do protótipo, bem como os resultados, conclusões e ações realizadas.

6.1. Detalhes de Implementação e Evolução do Código

A base arquitetural do projeto originou-se de atividades práticas desenvolvidas na disciplina de Desenvolvimento de Jogos da UFMG. No entanto, para atender aos requisitos específicos de *ASTRAL*, o código sofreu mudanças substanciais.

A estrutura original, puramente didática, foi expandida para suportar sistemas mais complexos, como o gerenciamento de estados de IA hierárquicos, items, cutscenes, habilidades especificas, *Spatial Hashing* dinâmico, cálculo de caminhamentos, e todos os outros sistemas descritos nas seções anteriores.

6.2. Ambiente de Desenvolvimento e Ferramentas

O ciclo de desenvolvimento utilizou um conjunto de ferramentas de código aberto e proprietárias, integradas para garantir produtividade e compatibilidade:

- **IDE**: Visual Studio Code (VS Code) com extensões para C/C++ e CMake;
- **Compilação**: O sistema de build foi gerenciado via CMake, permitindo a configuração modular das dependências (SDL2, SDL_image, SDL_mixer, SDL_ttf);
- **Controle de Versão**: Git e GitHub para gerenciamento de código fonte e versionamento de *assets*. O repositorio oficial está disponível em [Oliveira 2026b];
- **Sistema Operacional**: O desenvolvimento iniciou-se em ambiente Linux (Ubuntu via WSL), depois migrou para Windows, e terminou com Linux nativo. Isso garantiu compatibilidade cross-platform de desenvolvimento. Mas, a versão final publicada foi compilada para navegadores. Ou seja, qualquer sistema com acesso a um navegador moderno pode rodar o jogo, sem necessidade de instalação;
- **Emscripten e publicações contínuas**: a compilação para WebAssembly é feita via Emscripten. Um pipeline de CI/CD no GitHub Actions automatiza o build web a cada push na branch principal, publicando a versão mais recente do jogo diretamente no GitHub Pages sem intervenção manual. Isso é necessário para facilitar o processo de publicação de novas versões para o público.

6.3. Playtests e Metodologia de Avaliação

A validação do protótipo seguiu uma abordagem mista, predominantemente qualitativa. A primeira sessão de testes ocorreu durante a Feira de Apresentação de Trabalhos de Conclusão de Curso (TCC) do DCC/UFMG, ao final do primeiro semestre do projeto. Ao todo, cerca de 15 pessoas diferentes testaram os protótipos e a versão final. Algumas testaram mais de uma vez.

6.3.1. Primeiro playtest

Perfil e Amostragem: O teste contou com uma amostragem por conveniência de aproximadamente 6 participantes presenciais, compostos majoritariamente por estudantes de Computação e entusiastas de jogos. Posteriormente, a *build* do jogo foi distribuída digitalmente para coleta de feedback remoto.

Metodologia: Os participantes jogaram a versão *Alpha* do jogo, que continha o nível de tutorial e o primeiro nível completo. A coleta de dados foi realizada através de observação direta (durante a feira) e formulários de feedback espontâneo. Essa versão foi disponibilizada em uma máquina física, com controle, no dia da Feira de TCCs.

O formulário de feedback foi disponibilizado digitalmente aos participantes via Google Forms. Ao todo, até o dia 13/12/2025, o formulário recebeu 6 respostas. Para o primeiro playtest, considerei os feedbacks vindos primordialmente desses dados. Segue a estrutura do formulário, com as perguntas e tipos de resposta:

1. **E-mail** (opcional): coletado apenas com consentimento explícito do participante para contato em playtests futuros;
2. **Compreensão da proposta narrativa** (múltipla escolha):

    - Entendi perfeitamente!
    - Não tinha entendido completamente antes de ler a descrição dessa pergunta.
    - Mesmo com a descrição, ainda não entendo completamente.
    - Não entendi durante o jogo, e não acho que tenha relação com a descrição.
3. **Mecânicas de movimentação** (resposta aberta): solicitou descrição livre da experiência de controle da personagem, com exemplos orientadores para estimular respostas específicas;
4. **Desafios encontrados** (resposta aberta): abrangeu armadilhas, inimigos e demais obstáculos, pedindo ao participante que detalhasse como se sentiu diante de cada situação;
5. **Sugestões de melhoria** (resposta aberta): restrita explicitamente a propostas não relacionadas a bugs, separando feedback de design de relatórios técnicos;
6. **Impacto dos bugs** (múltipla escolha):
    - Simplesmente me impediram de jogar!
    - Prejudicaram bastante a experiência.
    - Afetaram mais do que deveriam.
    - Não observei impacto real.
    - Não vi nenhum bug!
7. **Descrição dos bugs encontrados** (resposta aberta, opcional): complementar à questão anterior, para registro detalhado de falhas técnicas.

Resultados: a análise dos feedbacks permitiu identificar discrepâncias entre a intenção do design e a experiência do usuário. Os pontos críticos levantados foram:

1. **Ausência de controle aéreo**: o problema mais recorrente, apontado por 4 dos 6 participantes. Jogadores habituados a plataformers modernos esperavam poder redirecionar o pulo após o salto:
   "não poder mudar a direção do pulo depois que começou a pular acaba sendo difícil de se adaptar pelo costume em outros jogos"
2. **Feedback de combate e hitbox**: múltiplos jogadores relataram não saber se seus ataques estavam conectando com inimigos, reduzindo a satisfação do combate:
   "A responsividade de alguns inimigos é baixa, não sei se estou dando dano neles ou não"
   "Hitbox dos inimigos e do martelo muito estranhas/pequenas"
3. **Ausência de invencibilidade após dano**: jogadores relataram situações em que receber dano levava à morte em sequência sem possibilidade de reação:
   "Muitas vezes, tomar dano parece ser gameover na hora [...] aplicar uma janela de invencibilidade após tomar dano poderia ajudar"
4. **Impossibilidade de pular cutscenes ao morrer**: a obrigatoriedade de assistir ao tutorial a cada morte foi apontada como prejudicial à rejogabilidade:
   "ter que pular os primeiros diálogos com o narrador toda vez que morria prejudicou um pouco a experiência"
5. **Bug crítico na função de pausar**: dois participantes independentes reportaram que pressionar Enter durante cutscenes travava ou encerrava o jogo:
   "quando eu apertava enter pra pular as cutscenes, o jogo pausava e não voltava mais"

Estes resultados definiram o roteiro da fase seguinte de desenvolvimento, priorizando controle aéreo, feedback visual de combate, frames de invencibilidade e a possibilidade de pular cutscenes.

Ações tomadas a partir dos feedbacks:

1. **Controle aéreo**: a função `Move()` passou a ser chamada nos estados `Jumping` e `Falling` com modificador `1.3f`, 30% acima do valor padrão de solo, permitindo redirecionar a trajetória no ar. Para evitar aceleração ilimitada, um teto de velocidade horizontal é aplicado enquanto a personagem está no ar:

```
float MAX_MOVE_X_SPEED_ON_AIR = 120.f;
if (!mRigidBodyComponent->GetOnGround() &&
   Math::Abs(currentVelX) >= MAX_MOVE_X_SPEED_ON_AIR &&
   Math::Sign(currentVelX) == Math::Sign(movementDir.x))
{
   return; // nao aplica forca na mesma direcao
}
mRigidBodyComponent->ApplyForce(movementDir * mForwardSpeed *
   modifier);
```

   Mudanças de direção no ar permanecem sempre livres; apenas a aceleração contínua na mesma direção é limitada;
2. **Feedback de combate**: os colisores de ataque foram remanejados e knockbacks calibrados empiricamente para comunicar impacto de forma mais clara. A remoção do A* para movimentação de inimigos também contribuiu: trajetórias exatas demais tornavam os encontros previsíveis e frustrantes, comprometendo a percepção de combate;
3. **Invencibilidade após dano**: adicionados frames de invencibilidade pós-dano. O dodge também foi reforçado como ferramenta de esquiva, com sua mecânica introduzida explicitamente via cutscene de tutorial;
4. **Cutscenes repetidas**: cutscenes não transicionais passaram a ser exibidas apenas uma vez. Cutscenes transicionais, que gerenciam eventos fundamentais para a progressão, não podem ser puladas e foram mantidas como estavam;
5. **Bug do Enter**: correção pontual; nas versões atuais, a tecla pausa o jogo e o áudio corretamente.

6.3.2. Playtest subsequentes

A partir do primeiro playtest, o jogo passou por ciclos iterativos de desenvolvimento e teste, em que foram disponibilizadas versões mais recorrentes para um grupo de teste mais amplo, incluindo amigos e familiares, para coleta de feedbacks adicionais.

Primeiramente, o orientador deste trabalho forneceu alguns feedbacks precisos e ações correspondentes, que foram:

1. **Movimentação estática, pouco responsiva e controlável**: mesmas correções aplicadas a partir do primeiro playtest, controle aéreo via `Move(1.3f)` nos estados `Jumping` e `Falling`, com teto de velocidade horizontal de 120 unidades por segundo para evitar aceleração ilimitada na mesma direção;

2. **Cutscenes e diálogos excessivos**: todas as cutscenes e diálogos foram revisitados e reduzidos. Comunicações que podiam ser substituídas por *game design* implícito foram removidas, priorizando *juiciness* em detrimento de instruções explícitas;
3. **Diálogos com tempo limite prejudicam a leitura**: optou-se por manter o avanço automático, aumentando a duração da janela para garantir tempo hábil de leitura. A barra de progresso acima da caixa de diálogo comunica visualmente o tempo restante;
4. **Ventania omni-direcional era confusa**: a direção do impulso passou a ser quantizada pela função `SnapVentaniaDir()`, que mapeia qualquer input do analógico para a direção mais próxima dentro de um conjunto fixo de cinco opções (direita, diagonal direita-cima, cima, diagonal esquerda-cima e esquerda). Nenhuma direção descendente é permitida, tornando a mecânica mais legível e previsível;
5. **Foco em polimento de mecânicas existentes**: a habilidade de aura elétrica, prevista no GDD e no relatório do primeiro semestre, foi abandonada. O esforço foi redirecionado para finalizar a *Nevasca*, introduzir sistemas mais diretos e polir o que já existia;
6. **Remover cooldown de Ventania e manter uma ativação por pouso**: o controle de uso da Ventania passou a ser exclusivamente pela flag `mLandedAfterVentania`, sem nenhum timer de cooldown. Ela é definida como `false` ao usar a habilidade e restaurada para `true` ao pousar, limitando a um uso por vez no ar sem penalizar o jogador com tempo de espera;
7. **Entregar mecânicas gradualmente**: introdução do sistema de `Item`, que vincula a aquisição de cada habilidade a uma cutscene de tutorial contextual. Antes dessa mudança, todas as habilidades estavam disponíveis desde o início do jogo;
8. **Ventania deve ignorar gravidade**: durante o estado `Dashing`, `SetApplyGravity(false)` é chamado a cada frame. O timer `mDashGravityDisableTimer` (0.1s) controla a duração: ao expirar, a velocidade é zerada, a gravidade é restaurada e o estado transita para `Falling`, produzindo um impulso limpo sem interferência da física;
9. **Interação com o cenário**: adição de items não funcionais no primeiro nível (quarto da protagonista), livro, geladeira e foto, que acionam cutscenes exclusivamente narrativas, contextualizando o mundo e tornando o cenário mais profundo.

A partir daqui, o feedback dos jogadores ficou mais qualitativo e menos quantitativo, e pouco linear. Ao mesmo tempo em que correções eram implementadas, novos feedbacks de versões distintas emergiam. O questionário de avaliação foi mantido, e 4 novas respostas foram coletadas, totalizando 10 respostas até o dia 23/06/2026. Além disso, outros feedbacks foram coletados informalmente, via conversas com os testadores.

Nesse momento, o jogo passou a ser compilado para Web via Emscripten, e disponibilizado para os testadores via link no navegador, para que pudessem jogar em seus próprios computadores, com seus próprios controles. Essa abordagem trouxe novos bugs, associados a ferramenta emscripten, que não existiam antes. Mas, também permitiu uma coleta de feedback mais fluida, com testadores jogando em seus próprios ambientes, sem necessidade de configuração prévia.

O conjunto consolidado de defeitos reportados, excluídas as repetições, é apresentado a seguir com as respectivas correções:

1. **Fullscreen em tela preta**: ao ativar o modo de tela cheia no navegador (relatado em Chrome e Firefox), o jogo exibia apenas fundo preto e exigia recarregamento da página. Defeito pontual na inicialização do contexto SDL em ambiente Emscripten; Foi corrigido em versão subsequente;
2. **Mapeamento de controle incorreto na versão web**: os botões exibidos nas dicas e utilizados nas cutscenes de tutorial não correspondiam ao *layout* físico do *gamepad* conectado. A versão Emscripten não estava consultando o arquivo `gamecontrollerdb.txt` para remapear os controles; o carregamento do banco de dados foi habilitado no *build* web;
3. **Loop infinito entre salas e portal que não suga**: ao percorrer ambas as salas do primeiro nível, e voltar a primeira sala, retroceder para a segunda sala não carregava a cutscene do Portal do segundo nível. A correção foi simples, foi necessário definir a flag de transicional dessa cutscene para `true` para garantir que ela fosse construída a cada entrada na sala, e não apenas na primeira vez;
4. **Espera longa passar por MoveSteps próximo de paredes**: o `MoveStep` responsável por movimentar Zoe recebia como destino uma posição inalcançável e aguardava o *timeout* máximo, sem se mover. A correção envolveu reduzir o *timeout* dos `MoveSteps` que podem ser executados próximos à paredes;
5. **Zoe sem pulo após reinicialização por morte no primeiro nível**: ao morrer e reinicializar, o pulo permanecia bloqueado. A cutscene inicial do primeiro nível é quem desbloqueia o pulo ao final; como ela era registrada com `overwrite=false` (não transicional), a tentativa de re-registrá-la no segundo carregamento era silenciosamente ignorada e ela nunca disparava novamente. Corrigido marcando-a com `overwrite=true`. A mesma causa explica o defeito seguinte;
6. **Nevasca não re-obtida ao morrer no nível 2**: morrer definitivamente no segundo nível e retornar ao *checkpoint* inicial daquele nível tornava o progresso impossível, o *puzzle* seguinte exige a Nevasca, mas o item que a concede já havia sido consumido e a cutscene de aquisição não era retrigada. Mesma causa do item anterior: a cutscene passou a ser registrada com `overwrite=true` para permitir reinserção a cada morte;
7. **Tela de game over não aparecia ao morrer no Quasar**: em situações pontuais, a morte definitiva por contato com o Quasar não chamava a transição para a `DeathScreen`. Defeito pontual no caminho de execução de `Kill()`;
8. **Projétil do inimigo vira Zoe para o lado do knockback**: ao receber o *knockback* de um projétil, o vetor de velocidade resultante alterava a rotação do sprite, pois ambos estavam acoplados. A correção desvinculou as duas: fora de cutscenes, a rotação de Zoe responde apenas a `mInputMovementDir`; dentro de cutscenes, a velocidade dita o sentido (necessário para que o `MoveStep` oriente o sprite corretamente durante animações automáticas):

```
// durante cutscene: velocidade orienta o sprite
if (PlayingCutscene && velocity.x > 0.f)
   SetRotation(0.0f);
// fora de cutscene: apenas input orienta
if (!PlayingCutscene && inputDir.x != 0.f)
   SetRotation(inputDir.x > 0.f ? 0.0f :Math::Pi);
```

9. **Ventania infinita ao pressionar Zoe contra parede**: pressionar o personagem contra uma parede enquanto se tentava disparar a Ventania repetidamente permitia ativações consecutivas sem pousar. Corrigido pela mesma flag `mLandedAfterVentania` descrita anteriormente.
10. **Coletar item em cima dele trava os *move steps***: ao estar posicionado exatamente sobre o item no momento da coleta, o `MoveStep` de aproximação recebia um destino inalcançável e aguardava o *timeout*. Reduzir o *timeout* máximo dos `MoveSteps` de aquisição de item para 0.5 s (o mesmo ajuste do item 4) eliminou a pausa.
11. **Golpe carregado lança o Quasar levemente abaixo das Shurikens**: o impulso vertical aplicado ao Quasar ao receber o golpe carregado era insuficiente para garantir o acerto consistente nas Shurikens posicionadas acima. O vetor foi ajustado de `(0, -500)` para `(0, -520)` em `Quasar::TakeKnockback`.
12. **Mecânica do golpe carregado não era clara na cena do Quasar**: o jogador não recebia indicação de que devia usar o golpe carregado para arremessar o Quasar nas Shurikens. Foi introduzido um contador (`mQuasarEncounterTimeCounter`) que incrementa enquanto o jogador está na área e o Quasar ainda possui vida máxima. Decorridos `MAX_TIME_QUASAR_ENCOUNTER_TIP = 20 s` sem nenhum dano aplicado ao Quasar, o jogo comunica uma dica para o jogador. O mesmo mecanismo foi reutilizado na cena de congelamento da caixa metálica no nível 2.
13. **Softlock do Quasar pressionando Zoe contra a parede**: o ataque de *dash* do Quasar podia empurrar Zoe horizontalmente contra uma parede, deixando-a sem espaço para pular ou reagir. A limitação de proximidade de parede já existente na `AIMovementComponent` não era suficiente para impedir o contato. A correção foi adicionada em `Zoe::OnHorizontalCollision`: ao detectar colisão com `ColliderLayer::Quasar`, verifica-se se Zoe está encostada em uma parede lateral; nesse caso o *knockback* é invertido horizontalmente e mantém componente vertical para cima, arremessando-a para longe:

```
int closeToWall = mColliderComponent
   ->IsCloseToTileWallHorizontally(1.f);

if (closeToWall == -1 && minOverlap > 0)
   TakeKnockback(Vector2(1, -1) * knockbackForce); // parede esq.
else if (closeToWall == 1 && minOverlap < 0)
   TakeKnockback(Vector2(-1, -1) * knockbackForce); // parede dir
      .
else if (quasar->GetBehaviorState() == BehaviorState::Attacking)
   TakeKnockback(Vector2(Math::Sign(-minOverlap), -1) *
      knockbackForce);
```

14. **Mecânica de agarro não satisfatória**: a mecânica de agarro da personagem principal *Cling* foi ajustada de acordo com a feedbacks coletados, e com estudo de gameplay de outros plataformers (*ScourgeBringer*). O ajuste consistiu em não obrigar que o jogador mantivesse a ação de pressionar contra a parede para manter o agarro, e sim permitir que o jogador solte o botão de direção, mantendo o agarro ativo. Facilitando o controle de outras mecânicas associadas, como a Ventania.

15. **Mecânica de agarro ativando em baixas alturas**: a mecânica de agarro era ativada mesmo quando a personagem estava muito próxima do chão, dificultando pulos com obstáculos próximos. A correção consistiu em adicionar uma verificação de altura mínima para permitir o agarro, evitando que a mecânica atrapalhasse a movimentação próxima do chão.

7. Uso de Inteligência Artificial

Esta seção será uma descrição do uso de inteligência artificial neste trabalho, discorrendo sobre a utilização de Grandes Modelos de Linguagem (LLMs) e suas ferramentas. Essas tecnologias acarretam em ganhos de produtividade que são de grande relevância para desenvolvimento de jogos, principalmente em equipes pequenas, e tempo curto, como é o caso deste trabalho.

1. **Geração de Assets**: alguns assets de background para menus foram gerados com auxílio de LLMs, como o Gemini 3 Pro Image [DeepMind 2026], da Google.
   Entre eles estão: o fundo do menu principal, o fundo do menu de morte de jogador e o fundo de menu de recomeço de gameplay, ao final do jogo. Todos os outros assets foram obtidos de repositórios de assets livres, como mencionado anteriormente, ou são fruto de criações e edições do desenvolvedor do jogo.
2. **Geração de Código**: o código do jogo é inteiramente escrito pelo desenvolvedor. Mas a função de publicação para web, via Emscripten [Contributors 2024a], foi gerada com auxílio de LLMs. A configuração foi revisada e editada até funcionar plenamente para permitir a publicação sem bugs tanto para ambientes de testes, quanto para o público geral.
3. **Debugação**: alguns bugs encontrados durante o desenvolvimento foram resolvidos com auxílio de LLMs, que ajudaram a identificar a causa raiz do problema, e sugerir soluções. Mesmo com auxílio de LLMs, a implementação das soluções foi feita pelo desenvolvedor, que revisou e adaptou as sugestões para o contexto do jogo.
4. **Geração de Documentação**: os diagramas gerados para este relatório foram produzidos com auxílio de LLMs, que ajudaram a gerar o código em *Mermaid* [Contributors 2024b] para os diagramas, que foram posteriormente revisados e editados pelo desenvolvedor.
5. **Revisão**: a revisão tanto desta documentação, do código do jogo e dos feedbacks de playtests foram auxiliadas por LLMs, que ajudaram a identificar erros de digitação e inconsistências, melhorar coesão textual, sugerir melhorias e encontrar erros de lógica.
6. **Pesquisa**: a pesquisa de literatura e estudos relacionados ao desenvolvimento de jogos foi auxiliada por LLMs, que ajudaram a encontrar e resumir informações relevantes.

8. Conclusão

Este trabalho documentou o ciclo completo de produção de *ASTRAL*, um jogo de plataforma 2D desenvolvido em C++ com SDL2, sem engine comercial, ao longo de dois semestres. O resultado é uma experiência jogável de aproximadamente 20 a 30 minutos, com dois níveis completos, quatro inimigos com comportamentos distintos, cinco

mecânicas de movimentação e combate, sistema de cutscenes e diálogos, e compilação para navegador via Emscripten.

A escolha de não utilizar uma engine foi deliberada e mantida até o fim. Ela exigiu a implementação manual de sistemas que costumam ser fornecidos prontos, física com integração de Euler, colisão em duas fases (*spatial hashing* + AABB por eixo separado), câmera com quatro modos, sequenciador de cutscenes, sistema de áudio com pool de oito canais e cache de sons, transições de cena com *fade*.

Essa decisão tornou o desenvolvimento mais lento, mas também mais coerente com seus objetivos pedagógicos: cada problema encontrado, desde aliasing de rotação até gestão fina de colisões, revelou fundamentos que engines abstraem parcialmente.

O maior aprendizado do projeto não foi técnico, mas de design: o algoritmo A*, implementado e funcional, foi descontinuado. Seu problema não estava na correção do algoritmo, mas no que ele comunicava ao jogador, trajetórias determinísticas e exatas que tornavam o combate previsível de forma frustrante, de acordo com os feedbacks.

Substituí-lo por um sistema de percepção com estados de busca, patrulha e errância, acrescido de vetores de repulsão para desvio de obstáculos, produziu comportamentos mais orgânicos. Esse episódio ilustra com clareza a distinção entre complexidade algorítmica e qualidade de experiência: a solução mais sofisticada pode ser a menos adequada ao jogador.

O processo iterativo de playtests foi determinante para a evolução do jogo. O primeiro playtest formal revelou ausências que não eram visíveis durante o desenvolvimento isolado: falta de controle aéreo, feedback de combate insuficiente, ausência de invencibilidade pós-dano e bugs críticos de progressão.

Isso é padrão de um primeiro playtest, onde o jogo não tinha sido testado por pessoas externas, e vários dos seus sistemas precisavam ser postos à prova, e alguns completamente reescritos.

Os playtests subsequentes, mais qualitativos, identificaram uma camada diferente de bugs pontuais a serem corrigidos, como softlocks, colisões incorretas e glitches, para garantir maior polimento.

A convergência entre o feedback do orientador e dos jogadores em pontos como a movimentação aérea confirmou que problemas recorrentes em playtests distintos raramente são coincidência.

As limitações do trabalho são conscientes e documentadas. A equipe reduzida impôs restrições de escopo: a habilidade de aura elétrica, prevista no GDD, foi abandonada para priorizar o polimento do que estava implementado.

Além disso, os assets visuais são de terceiros, o que preservou tempo de desenvolvimento ao custo da identidade visual autoral. Não há sistema de save persistente, e o conteúdo total não ultrapassa 30 minutos. Essas escolhas refletem um julgamento deliberado sobre o que era alcançável dentro de um ano com um desenvolvedor/designer, um orientador, e um grupo pequeno de testers.

Um dos focos deste trabalho, presente na problemática original, é documentar a experiência de desenvolvimento, e não entregar um produto comercial final. Para que,

outros futuros desenvolvedores possam aprender com os desafios e soluções encontrados, e não apenas com o resultado final.

Como trabalhos futuros, destacam-se:

1. Implementação da aura elétrica para completar o conjunto de habilidades previsto;
2. Expansão para mais níveis;
3. Introdução de um sistema de save para permitir sessões não contínuas;
4. A realização de playtests com público mais amplo e diversificado, incluindo pessoas sem experiência prévia no gênero;
5. Documentação do processo completo em vídeos em plataformas digitais, para alcançar um público mais amplo e não apenas leitores de artigos acadêmicos.

ASTRAL entrega o que se propôs: um jogo autoral com mecânicas dinâmicas, construído do zero, com processo documentado. O código completo está disponível em [Oliveira 2026b].

References

[Aseprite 2025] Aseprite (2025). Aseprite: Animated sprite editor & pixel art tool. https://www.aseprite.org/. Accessed: 2025-12-06.

[Bilas 2002] Bilas, S. (2002). A data-driven game object system. GDC 2002. https://www.youtube.com/watch?v=Eb4-0M2a9xE. Accessed: 2026-06-17.

[BorisTheBrave 2019] BorisTheBrave (2019). Dungeon generation in enter the gungeon. https://www.boristhebrave.com/2019/07/28/dungeon-generation-in-enter-the-gungeon. Accessed: 2025-12-03.

[Contributors 2024a] Contributors, E. (2024a). Emscripten: An llvm to webassembly compiler. https://emscripten.org. Accessed: 2026-06-28.

[Contributors 2024b] Contributors, M. (2024b). Mermaid: Diagramming and charting tool. https://mermaid.js.org. Accessed: 2026-06-28.

[Corcoran 2025] Corcoran, L. (2025). itch.io: Independent game marketplace and community. https://itch.io/. Accessed: 2025-12-06.

[Corporation 2017] Corporation, V. (2017). Hollow knight: Steam store page. https://store.steampowered.com/app/367520/Hollow_Knight/. Accessed: 2025-09-06.

[Corporation 2018] Corporation, V. (2018). Celeste: Steam store page. https://store.steampowered.com/app/504230/Celeste/. Accessed: 2025-09-06.

[Corporation 2020] Corporation, V. (2020). Scourgebringer: Steam store page. https://store.steampowered.com/app/1037020/ScourgeBringer/. Accessed: 2026-06-16.

[Corporation 2025] Corporation, V. (2025). Indie game: The movie: Steam store page. https://store.steampowered.com/app/207080/Indie_Game_The_Movie. Accessed: 2025-12-03.

[DeepMind 2026] DeepMind, G. (2026). Gemini 3 pro image. https://deepmind.google/models/gemini/. Accessed: 2026-06-28.

[Developer 2025a] Developer, G. (2025a). Formal abstract design tools. https://www.gamedeveloper.com/design/formal-abstract-design-tools. Accessed: 2025-12-03.

[Developer 2025b] Developer, G. (2025b). Game design deep dive: One-hit kills in titan souls. https://www.gamedeveloper.com/design/game-design-deep-dive-one-hit-kills-in-i-titan-souls-i-. Accessed: 2025-12-03.

[Favreau 2005] Favreau, J. (2005). Zathura: A space adventure. https://www.imdb.com/title/tt0406375/. Accessed: 2026-06-16.

[Foster 2015] Foster, M. (2015). Interview: Mark foster on titan souls. https://the3headedmonkey.blogspot.com/2015/10/interviews-mark-foster-on-titan-souls.html. Accessed: 2025-12-03.

[Games 1995] Games, R. B. (1995). Red blob games: algorithm and game-dev tutorials. https://www.redblobgames.com/. Accessed: 2025-11-25.

[Gregory 2018] Gregory, J. (2018). *Game Engine Architecture*. A K Peters / CRC Press, 1rd edition.

[Guillen 2025] Guillen, F. (2025). Papers, please devlog scrap. https://fguillen.github.io/PapersPleaseDevlogScrap. Accessed: 2025-12-03.

[Hunicke et al. 2004] Hunicke, R., LeBlanc, M., and Zubek, R. (2004). Mda: A formal approach to game design and game research. https://users.cs.northwestern.edu/~hunicke/MDA.pdf. Accessed: 2025-12-03.

[Lindeijer 2025] Lindeijer, T. (2025). Tiled: Flexible level editor (mapeditor.org). https://www.mapeditor.org/. Accessed: 2025-12-06.

[Lucasfilm 2019] Lucasfilm (1977–2019). Star wars (franquia cinematográfica). https://www.starwars.com/films. Accessed: 2026-06-16.

[MaddyMakesGames 2025] MaddyMakesGames (2025). Celeste and towerfall physics. https://maddymakesgames.com/articles/celeste_and_towerfall_physics/index.html. Accessed: 2025-12-03.

[Madhav 2013] Madhav, S. (2013). *Game Programming Algorithms and Techniques: A Platform-Agnostic Approach*. Addison-Wesley Professional, 1st edition.

[Meat 2010] Meat, T. (2010). Super meat boy: Steam store page. https://store.steampowered.com/app/40800/Super_Meat_Boy/. Accessed: 2026-06-16.

[Ministério da Cultura 2026] Ministério da Cultura (2026). Marco legal dos games abre novas oportunidades para o desenvolvimento da indústria no brasil. Disponível aqui. Publicado em 16/03/2026. Accessed: 2026-06-16.

[Nacional 2024] Nacional, C. (2024). Veto nº 10/2024 – detalhamento do dispositivo vetado (art. 3º-b da lei 8.685/1993, incluído pelo art. 19 do pl 2.796/2021). online legislative record. Received 06 May 2024; under suspension since 05 June 2024.

[Newzoo 2025] Newzoo (2025). Global games market report, january 2024 (free version). https://newzoo.com/resources/trend-reports/newzoo-global-games-market-report-2025. Accessed: 2026-06-1filt6.

[Nintendo 1985] Nintendo (1985). Super mario bros: Official game page. https://www.nintendo.com/en-gb/Games/NES/Super-Mario-Bros-803853.html. Accessed: 2026-06-16.

[nlohmann 2025] nlohmann (2025). Json for modern c++ (nlohmann/json): Github repository. https://github.com/nlohmann/json. Accessed: 2025-12-06.

[Norinchak 2025] Norinchak, A. O. (2025). Free texture packer: sprite sheet / atlas generator (web app). https://free-tex-packer.com/app/. Accessed: 2025-12-05.

[Nystrom 2014] Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning.

[Oliveira 2026a] Oliveira, P. (2026a). Astral gdd. https://docs.google.com/document/d/1gOtH9MNsNgO--11MScywYJZnbogwulrNnyu3uo4BiaY/edit?tab=t.xlqg7mpd9mrh#heading=h.lh134xahjzic. Accessed: 2026-06-17.

[Oliveira 2026b] Oliveira, P. H. F. G. (2026b). Astral: Repositório do jogo. https://github.com/pedrofreitaas/astral. Accessed: 2026-06-17.

[Piskel 2025] Piskel (2025). Piskel: Free online sprite editor and pixel-art tool. https://www.piskelapp.com/. Accessed: 2025-12-05.

[(SDL) 2025] (SDL), S. D. L. (2025). Sdl: Simple directmedia layer (official site). https://www.libsdl.org/. Accessed: 2025-09-06.

[Shiffman 2025] Shiffman, D. (2025). The nature of code. https://natureofcode.com/. Accessed: 2026-05-11.

[Swink 2009] Swink, S. (2009). *Game Feel: A Game Designer's Guide to Virtual Sensation*. Morgan Kaufmann.

[Yu 2025] Yu, D. (2025). Spelunk / makegames: Devlogs and notes. https://www.derekyu.com/makegames. Accessed: 2025-12-03.