Desenvolvimento de um Jogo Metroidvania em C++ com SDL: Uma Abordagem sem o uso de *third-party engines*

Renato Augusto Platz Guimarães Neto
Campus Paranavaí – Instituto Federal do Paraná
Paranavaí, PR, Brasil
renatoplatz11@gmail.com

Eduardo Henrique Molina da Cruz
Campus Paranavaí – Instituto Federal do Paraná
Paranavaí, PR, Brasil
eduardo.cruz@ifpr.edu.br

***Abstract*—This article investigates the challenges and best practices in developing digital games without the use of third-party engines (TPEs), using the C++ language and the SDL library. Through a case study focused on the creation of a Metroidvania game, the work details the manual implementation of fundamental systems, such as the physics engine and animation. The results, validated by user testing, confirm the technical feasibility of the approach and highlight the balance between the flexibility to create unique mechanics and the complexity of refining interactions such as combat. The study highlights the formative and technical potential of low-level development, positioning it as a valuable alternative for deepening the fundamentals of game programming.**

***Keywords*—Game; C++; SDL.**

***Resumo*—Este artigo investiga os desafios e as boas práticas no desenvolvimento de jogos digitais sem o uso de third-party engines (TPEs), utilizando a linguagem C++ e a biblioteca SDL. Através de um estudo de caso focado na criação de um jogo do gênero Metroidvania, o trabalho detalha a implementação manual de sistemas fundamentais, como motor de física e animação. Os resultados, validados por testes com usuários, confirmam a viabilidade técnica da abordagem e destacam o balanço entre a flexibilidade para criar mecânicas únicas e a complexidade de se refinar interações como o combate. O estudo evidencia o potencial formativo e técnico do desenvolvimento em baixo nível, posicionando-o como uma alternativa valiosa para o aprofundamento dos fundamentos da programação de jogos.**

***Palavras-chave*—Jogos; C++; SDL.**

I. Introdução

O mercado global de jogos digitais apresenta um crescimento contínuo, com receitas que alcançaram 188 bilhões de dólares em 2024, superando as indústrias de cinema e música combinadas [1]. Este cenário representa um campo com vastas oportunidades para artistas, roteiristas, músicos e engenheiros de software. Motivados pelo entusiasmo, muitos iniciantes recorrem a motores de jogo de terceiros (do inglês *third-party engines* - TPEs).

As *TPEs* são essenciais na indústria de jogos digitais — principalmente para estúdios pequenos e desenvolvedores independentes — mas iniciar diretamente por elas pode dificultar, em alguns casos, o desenvolvimento de uma base mais sólida sobre os fundamentos da criação de jogos. Isso acontece porque elas já fornecem muitas funcionalidades prontas, como motor gráfico, motor de gravidade e sistema de animações, por exemplo. Embora não ter um domínio avançado sobre desenvolvimento de jogos digitais não impossibilite de criar um jogo usando uma *TPE*, ter uma base mais sólida permite extrair ao máximo o potencial dessas ferramentas.

Para alcançar essa base, este trabalho propõe explorar o desenvolvimento de jogos sem o uso de uma *TPE*, utilizando apenas a linguagem de alto nível e propósito geral C++, juntamente com a biblioteca multiplataforma e de código aberto *Simple DirectMedia Layer* (SDL). O objetivo é proporcionar um estudo mais aprofundado sobre os fundamentos da programação de jogos, buscando compreender tanto os desafios — como o gerenciamento de memória e a complexidade na organização do código — quanto as vantagens, como maior flexibilidade, desempenho e domínio sobre o código. Neste contexto, surge a seguinte questão de pesquisa: *Quais são os desafios técnicos e as boas práticas envolvidas no desenvolvimento de jogos digitais sem o uso de TPEs, com foco na linguagem C++ e na biblioteca SDL?* A resposta a essa pergunta pode fornecer qual o enriquecimento proporcionado pela abordagem.

O estudo é validado por meio da implementação de um jogo no estilo *Metroidvania*, que serve como estudo de caso para avaliar boas práticas de desenvolvimento de jogos digitais. A escolha desse gênero específico se justifica por sua complexidade, que exige a implementação de mecânicas variadas de movimentação, sistemas físicos aplicados e gerenciamento de estados. Essas características tornam o *Metroidvania* um

excelente campo de análise, permitindo investigar soluções robustas e práticas eficazes no desenvolvimento de jogos.

Este artigo está organizado da seguinte forma. Na Seção II, apresenta a fundamentação teórica sobre desenvolvimento de jogos. A Seção III apresenta as TPEs e trabalhos relacionados. A Seção IV descreve o processo de desenvolvimento do jogo utilizado como estudo de caso. A Seção V realiza a análise e avaliação dos resultados. Por fim, a Seção VI apresenta as conclusões e sugestões para trabalhos futuros.

II. Fundamentação Teórica sobre Desenvolvimento de jogos

Nesta seção, são apresentadas as principais técnicas e ferramentas utilizadas durante o desenvolvimento de jogos, como a Linguagem de Programação, o Paradigma de Programação, a Arquitetura de Software e Bibliotecas.

A. Linguagens e Bibliotecas para Desenvolvimento de Jogos

Existem diferentes linguagens e bibliotecas para desenvolvimento de jogos, para este trabalho foram selecionadas a Linguagem *C++* e a biblioteca *Simple DirectMedia Layer*.

1) A linguagem de programação multiparadigma C++: fornece suporte aos paradigmas imperativo, orientado a objetos, genérico e de propósito geral, sendo inicialmente uma extensão da linguagem de programação *C* [2]. Criada por Bjarne Stroustrup, em 1980, nos laboratórios *Bell*, em Nova Jersey. As principais vantagens da linguagem são sua eficiência, flexibilidade e a filosofia de que "o programador, e não a linguagem, é o encarregado". Também destaca-se o suporte à programação orientada a objetos.

As características orientadas a objetos do *C++*, nas palavras de Stroustrup, “permitem que programas sejam estruturados para clareza, extensibilidade e facilidade de manutenção, sem perda de eficiência”. A escolha do *C++* neste trabalho não se dá apenas por suas qualidades técnicas, mas também por seu amplo uso na indústria de jogos digitais. Como aponta Nystrom [3], *C++* é a língua franca da indústria", sendo a linguagem mais popular em jogos comerciais. Além disso, a sintaxe de *C*, que serve de base para *Java*, *C#*, *JavaScript*, entre outras, também contribui para sua relevância.

2) SDL – Simple DirectMedia Layer: De acordo com Mitchel [4], *Simple DirectMedia Layer (SDL)* é uma biblioteca multimídia multiplataforma criada por Sam Oscar Latinga. Ela fornece – em baixo nível – sistema de entrada (via *mouse*, teclado e *gamepads/joysticks*), *hardware* 3D, audio e *frame buffer* de vídeo em 2D. SDL é escrita na linguagem de programação C, mas também tem suporte nativo para C++.

SDL foi utilizada em vários jogos comerciais, incluindo *Half-Life*, *Neverwinter Nights* e *Second Life*. Muito usada em emuladores como *ZSNES* e *Mupen64*. SDL é uma biblioteca intuitiva e fácil de implementar, permitindo criar uma aplicação que rodará em múltiplas plataformas com poucos ajustes no código, além de ter uma boa documentação e ter uma sólida comunidade de usuários. Isso permite ao desenvolvedor se preocupar mais com o conteúdo do que com a necessidade de uma tela ou desenvolver um sistema de I/O (*Input e Output*).

B. Arquitetura de Software e Paradigmas

De acordo com Nystrom [3], *“arquitetura é sobre mudança”*. O código passa por mudanças, seja para adicionar novas mecânicas, ou realizar correções para melhor desempenho. Para desenvolvimento de jogos, existem duas arquiteturas mais comuns, a Baseada em Componentes (no caso a ECS) e a Orientada a Eventos, geralmente elas são utilizadas em conjunto com paradigmas como Programação Orientada a objetos (POO). Neste tópico, são abordados os conceitos básicos dessas arquiteturas e do paradigma POO.

1) Programação Orientada a Objetos: Para compreender o básico do Paradigma POO, deve-se entender o que é um *objeto*. Um *objeto* é uma **entidade que possui uma identidade**, podendo ser tanto algo *concreto* (como pessoas, carros e biscoitos) quanto algo *conceitual* (leis, emoções, entre outros), *objetos* são criados através de *Classes*. *Classes* seguem como um modelo para a criação de um *objeto*, **um *objeto* é *a instância de uma classe***, os *objetos* ficam então organizados em grupos, sendo cada grupo representado por sua respectiva *classe*, contendo seus atributos, métodos e comportamentos únicos.

Seguem as características da POO. **Abstração**, focalizar nos aspectos essenciais a uma entidade. Preserva a liberdade de tomada de decisões de desenvolvimento ou implementação. **Encapsulação**, separar os aspectos externos dos detalhes internos de um objeto. Evita que um programa torne-se interdependente e mudanças o afetem por completo. **Herança**, permite que estruturas comuns sejam compartilhadas entre classes similares sem redundância. **Ligações/Associações**, estabelecem relacionamentos entre objetos e classes. Ligação é uma conexão entre duas instâncias. Associação é um conjunto de ligações.

2) Arquitetura Baseada em Componentes: De acordo com Gillin [5], a arquitetura baseada em componentes (*Entity Component System – ECS*) é uma estrutura para a construção de software com base em partes reutilizáveis. Cada componente contém funcionalidades bem definidas em uma unidade binária que é armazenada em uma biblioteca e, em seguida, inserida em um aplicativo sem modificar outros componentes. Os benefícios incluem tempo reduzido de desenvolvimento e teste, confiabilidade aprimorada e flexibilidade para alterar aplicativos, adicionando ou substituindo componentes sem interrupção.

As características mais comuns dos componentes são: **Reutilização**, os componentes se conectam a várias aplicações sem a necessidade de adaptação; **Extensibilidade**, componentes

se combinam com outros para criar novos comportamentos; **Substituibilidade**, componentes com funcionalidade similar podem ser trocados; **Encapsulamento**, Os componentes são autocontidos e expõem a funcionalidade por meio de interfaces, ao mesmo tempo que ocultam os detalhes dos processos internos; **Independência**, os componentes têm mínima dependência e podem operar em diferentes ambientes e contextos.

Quando se trata de desenvolvimento de jogos, trabalha-se com uma versão mais específica, no caso a ECS (Entidade-Componente-Sistema). Ela separa os dados dos comportamentos para melhorar desempenho, flexibilidade e organização do código. ela consiste nos seguintes elementos: **Entidades**, um identificador único (geralmente um número), que representa qualquer "coisa" no jogo; **Componente**, dados puros que descrevem uma característica de uma entidade; e **Sistema**, lógica que processa entidades com determinados componentes.

3) Arquitetura Orientada a Eventos: De acordo com AWS [6], arquitetura orientada a eventos (*Event Driven Architecture* – EDA) é um padrão de arquitetura moderno criado com base em serviços pequenos sem acoplamento que publicam, consomem ou encaminham eventos. Um *evento* representa uma mudança de estado ou uma atualização. Por exemplo: um item colocado em um carrinho de compras, um arquivo carregado em um sistema de armazenamento ou um pedido que está pronto para ser enviado. Os eventos podem conter o estado ou *identificadores* para pesquisa de informações relacionadas.

Seguem as características da EDA. **Diversas funções rápidas**, cria-se diversas funções rápidas, especializadas e concisas, elas geralmente reduzem o tempo de processamento. **Processamento sob demanda**, onde o processamento personalizado pode responder a todos os eventos. Permitindo que o serviço aumente a escala verticalmente, tornando-o quase em tempo real. **Recuperação de interrupções**, Um serviço pode ser chamado de forma automática para tentar processar um evento novamente. Um serviço pode receber o mesmo evento outra vez, as funções projetadas devem ser idempotentes.

III. Third-Party Engines, Custom Engines e Trabalhos Relacionados

No início, as *engines* não eram separadas dos jogos. Cada estúdio desenvolvia seus próprios sistemas, feitos sob medida para cada título. Contudo, devido ao avanço dos computadores e à crescente complexidade dos jogos, os estúdios começaram a perceber as vantagens de reaproveitar suas ferramentas. Um grande marco foi a *id Software*, que, após o desenvolvimento de *Wolfenstein 3D* (1992) — pioneiro dos jogos de tiro em primeira pessoa (FPS, *First Person Shooter*) — e de *Doom* (1993) [7], percebeu que seu motor gráfico poderia ser reaproveitado. Assim, passou a licenciá-lo.

A *Unreal Engine*, desenvolvida inicialmente por Tim Sweeney, que mais tarde fundaria a Epic Games, a *engine* foi lançada junto com o jogo Unreal em 1998 [8]. Na época, o jogo foi revolucionário graças à sua renderização rápida em 3D, estabelecendo um novo padrão para a indústria. Escrita em C++, a *engine* permitia que a maioria dos desenvolvedores aprendesse a utilizá-la com facilidade. A evolução do suporte de 8 bits para 16 bits trouxe mudanças visuais impressionantes, que poucas outras *engines* da época conseguiam oferecer. Outro recurso fundamental, criado por Sweeney, foi um editor de níveis que permitia aos desenvolvedores fazer alterações no *layout* dos mapas em tempo real.

Em 1999, era lançada a *GameMaker*, criada por Mark Overmars, inicialmente com o nome de *Animo*, destinada a ser um programa de Animação 2D [9]. Durante a primeira década do século 21, ocorreu uma democratização na indústria, com a *engine Unity* surgindo em 2005, inicialmente para Mac, mas que depois se expandiu para outras plataformas [10]. Através de seu modelo de preços acessível, a Unity impulsionou o mercado *indie*. Foi também nessa época que A *Unreal engine* mudou seu modelo de négocio para gratuito com *royalties*, tornando-se acessível a qualquer desenvolvedor.

Mesmo com a popularidade e recorrente adoção das *TPEs*, as *engines* customizáveis não são incomuns. Tanto no mercado *AAA*[1] quanto *Indie*[2], ainda são lançados vários jogos com *engines* customizadas.[3] Existem várias vantagens ainda em se criar a própria *engine*, não só na parte técnica, que possibilita ao desenvolvedor maior controle de seu código, mas também no financeiro, pois não se paga *royalties* ou licenças.

A. Comparativo entre TPEs

As *TPEs* possuem inúmeras vantagens e métodos de abordagem, como a curva de aprendizado, interface interativa, documentação estruturada e uma comunidade presente, muitas tem sua própria IDE, como Godot, outras são frameworks, como Love2d. Como mostrado no tópico anterior, existem diferentes *TPEs* no mercado, cada uma com foco e propostas diferentes, além de licenças diferentes, observe a tabela I e II onde são comparadas as *Engines* Godot, Love2d, Unity e Unreal Engine, que estão entre as mais conhecidas do mercado.

1) TPE para jogos 2D ou projetos 3D com estilização "indie": A **Godot Engine** se destaca. Seus recursos 2D são

[1]Classificação para jogos de maior orçamento, geralmente são jogos de estúdios grandes.

[2]Expressão que remete a um jogo "independente", geralmente feito por uma única pessoa ou equipe pequena.

[3]Nesse repositório montado pelo usuário raysan5, ele catalogou diferentes *engines* customizadas, desde jogos de estúdios grandes, até jogos feitos por uma única pessoa https://gist.github.com/raysan5/909dc6cf33ed40223eb0dfe625c0de74

TABELA I
COMPARATIVO DE TPES (PARTE 1: FOCO 2D E INDIE).

Característica	Godot	Love2d
Tipo	Engine completa	Framework
Licença	MIT	zlib/libpng
Custo	Gratuito	Gratuito
Melhor para	2D/3D indie	Prototipagem 2D
Linguagens	GDScript, C#, C++	Lua
Gráficos	2D: Excelente 3D: Bom (Vulkan)	2D: Foco principal 3D: Não
Dificuldade	Baixa	Média
Ecossistema	Comunitário	Mínimo

TABELA II
COMPARATIVO DE TPES (PARTE 2: FOCO 3D E COMERCIAL).

Característica	Unity	Unreal
Tipo	Engine completa	Engine completa
Licença	Assinatura	Royalties (5% após $1M)
Custo	Gratuito até $200K	Gratuito + royalties
Melhor para	Multiplataforma	AAA fotorrealista
Linguagens	C#	C++, Blueprints
Gráficos	2D: Bom 3D: Muito bom	2D: Limitado 3D: Excepcional
Dificuldade	Baixa-Média	Alta
Ecossistema	Asset Store	Marketplace

considerados "Excelente", com um workflow integrado e intuitivo para esse tipo de jogo. Seu desempenho em 3D, com o suporte a *Vulkan*, tem-se destacado cada vez mais, tornando-a viável para projetos tridimensionais. Dois exemplos de destaque são *Buckshot Roulette* e *Primal Light*.

2) Thirdy-Party Engine para prototipagem rápida e jogos 2D minimalistas: O **LÖVE** (ou Love2D) é uma escolha poderosa. Sendo um *framework*, ele oferece liberdade e exige que o desenvolvedor construa muitas das estruturas do zero. Ideal para quem quer ter alto controle do código, aprender os fundamentos da criação de jogos ou simplesmente criar um protótipo funcional em pouco tempo, com foco maior na lógica 2D. Dois jogos de destaque são *Balatro* e *Gravity Circuit*.

3) Para projetos multiplataforma (Mobile, PC, Consoles) e jogos 3D versáteis: A **Unity** é, a líder de mercado. Sua força reside no equilíbrio: é "Muito boa" em 3D e "Boa" em 2D, com o ecossistema mais robusto para exportar um mesmo projeto para diferentes plataformas. Se o objetivo é atingir o maior número de dispositivos possível, a *Unity* é uma aposta segura. Dois jogos de destaque são *Hollow Knight* e *Subnautica*.

4) Jogos AAA com foco em fotorrealismo e gráficos de ponta: A **Unreal Engine** é a escolha indiscutível. Sua capacidade gráfica em 3D é "Excepcional", sendo o padrão da indústria para jogos que buscam o limite do que é visualmente possível. O custo dessa potência é uma maior complexidade e recursos 2D mais "Limitados". Dois jogos de destaque são *Lies of P* e *Mortal Kombat 1*.

Fig. 1. *Stardew Valley.*

As *TPEs* são ferramentas que permitiram o desenvolvimento de títulos de peso como os apresentados, mostrando que são ferramentas essenciais de dominar para aqueles que buscam seguir carreira no mercado de jogos.

B. Custom Engines e Trabalhos Relacionados

Desenvolver um jogo sem o uso de *TPEs* não é algo incomum. há diversas vantagens em criar um jogo sem utilizar um motor pronto, entre as quais se destaca a possibilidade de desenvolver mecânicas e sistemas mais avançados e personalizados. Neste tópico, são abordados alguns exemplos de jogos desenvolvidos sem o uso de *TPEs*.

Um famoso jogo que não utiliza uma *TPE* é **Stardew Valley**, mostrado na Figura 1. Eric Barone decidiu desenvolver um jogo simples para utilizar como portfólio, tomando como inspiração um de seus jogos favoritos da infância. O que inicialmente seria um projeto de dois meses acabou se estendendo por quatro anos. Atualmente, o jogo ultrapassa 41 milhões de cópias vendidas [11], possui nota 89 no *Metacritic* e mais de 753.746 avaliações extremamente positivas na plataforma Steam [12]. *Stardew Valley* é um jogo de RPG (*Role-Playing Game*) e simulação de fazenda, apresentando diferentes mecânicas, que vão desde cuidar de animais e cultivar milho e se casar.

Para o desenvolvimento de seu jogo, Barone escolheu a linguagem de programação multiparadigma C#, utilizando o framework XNA — uma sigla recursiva para *XNA's Not Acronymed*. Esse framework era usado no desenvolvimento de jogos para Windows, Xbox 360 e Windows Phone 7. Ele optou por essa abordagem, em vez de utilizar uma *TPE* pois, segundo Schreier [13], acreditava que aprenderia muito mais e poderia construir cada parte do projeto à sua maneira[4]. Posteriormente,

[4]Além da programação, Barone também foi responsável pela criação das artes e trilhas sonoras do jogo.

Fig. 2. *Dwarf Fortress.*

o jogo foi migrado para a *engine* MonoGame, permitindo sua expansão para outras plataformas.

Outro jogo que não usa *TPE* é **Dwarf Fortress**, mostrado na Figura 2. Desenvolvido pelos irmãos Tarn e Zach Adams, *Dwarf Fortress* é um jogo *roguelike* de simulação, construção e gerenciamento em tempo real, disponível nas plataformas *Steam* e *itch.io*. O foco aqui será na versão clássica, um *freeware* que pode ser baixado diretamente no site oficial[5]. *Dwarf Fortress Classic* começou a ser desenvolvido em 2002 e lançado em 2006, sendo programado nas linguagens *C* e *C++*.

Segundo *Spliced Online* [14], Tarn Adams escolheu essas linguagens devido ao seu alto desempenho, elevado nível de controle e flexibilidade, além de serem as linguagens com as quais possuía maior familiaridade. Com isso, Tarn desenvolveu um motor de jogo customizável, que possibilitou a implementação das principais características de *Dwarf Fortress*, como a geração procedural de mundos, bem como mecânicas complexas de psicologia, economia, ecologia e geologia.

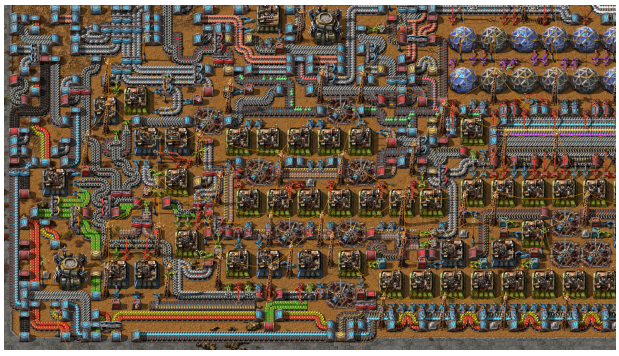

Fig. 3. *Factorio.*

O jogo *Factorio*, mostrado na Figura 3, foi originalmente feito por Michal Kovařík, Tomáš Kozelek e Albert Bertolín Soler, originários de Praga, República Tcheca. *Factorio* já vendeu mais de 3,500,000 de cópias, nas plataformas *Steam*, *GOG* e na própria loja do jogo. É um jogo onde o jogador constrói e mantém fábricas. Minerará recursos, pesquisará tecnologias, construirá infraestruturas, automatizando a produção e lutando contra inimigos [15]. A princípio pode não parecer uma proposta distante de outros jogos, contudo, deve-se ter destaque na principal mecânica de *Factorio*: a **automação**. O objetivo é deixar o mais automatizado possível, criando verdadeiras fábricas automatizadas que reaproveitam os próprios recursos coletados. O jogo foi desenvolvido em C++ e a *framework* Allegro para os gráficos, futuramente substituída por SDL [16].

Factorio é um jogo otimizado, o que permite a criação de várias fábricas ativas e automatizadas em larga escala em um mapa de geração procedural – o tamanho do mapa é limitado a 2000 x 2000 quilômetros (um quadrante com 2.000.000 blocos de lado, uma área de 4.000.000.000.000 de peças no quadrante). Isso está entre o tamanho da Índia e da Austrália. O trem – veículo presente no jogo – levaria cerca de 240 minutos de jogo para alcançar a fronteira saindo do centro [17].

C. Análise dos trabalhos relacionados

Ao analisar os três jogos mencionados, é evidente o impacto de se desenvolver um jogo utilizando uma *engine* própria. *Stardew Valley* foi desenvolvido com ampla liberdade criativa e técnica, permitindo que seu criador ajustasse as mecânicas de acordo com sua visão. *Dwarf Fortress*, ao empregar tecnologias de mais baixo nível, conseguiu se tornar um dos simuladores mais complexos já criados. Já *Factorio* se destaca por manter um nível de alta complexidade, aliado a uma otimização forte.

Da mesma forma, no desenvolvimento do estudo de caso do jogo desenvolvido neste trabalho, observa-se benefícios semelhantes, ao também utilizar uma *engine* própria: maior liberdade e controle na criação das mecânicas, além da modularidade, manutenibilidade e otimização.

IV. Desenvolvimento do jogo *A Rusty Screw*

Nesta seção é abordado o processo do desenvolvimento do jogo de estudo de caso, a adoção do gênero *Metroidvania*, apresenta a arquitetura adotada e também explica a relação entre as classes e entidades do jogo.

A. Adesão ao gênero metroidvania

O jogo *A Rusty Screw*, observado na Figura 4 – o estudo de caso deste trabalho – é do gênero *Metroidvania*. Para aderir-se ao gênero, necessitou-se a compreensão dos elementos essenciais que caracterizam esse gênero, que combinam exploração, progressão baseada em desbloqueio de áreas através de novas mecânicas adquiridas.

Durante a fase de concepção, foram definidas mecânicas fundamentais para tornar o jogo aderente ao gênero, como,

[5]https://www.bay12games.com/dwarves/index.html

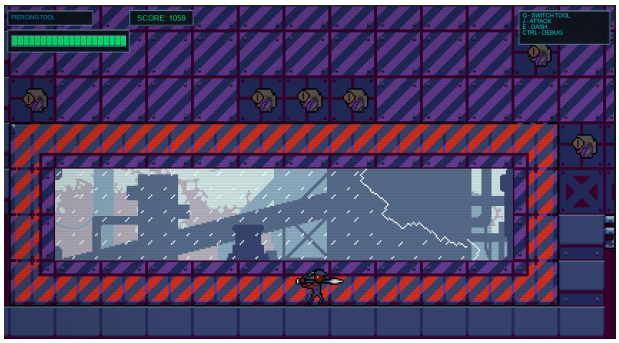

Fig. 4. Jogo A Rusty Screw.

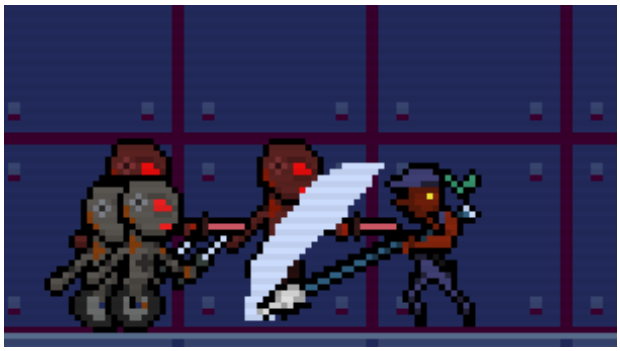

Fig. 5. A ponta inicial do jogador é a de "fenda", com um ataque cortante.

por exemplo, a mecânica de “desparafusar”. Nessa mecânica, o jogador utiliza pontas de chave em sua arma, que não servem apenas para o combate — fornecendo movimentações e atributos diferentes, como pode ser notado nas Figuras 5 e 6 — mas também para resolver quebra-cabeças ou remover obstáculos no cenário, como pode ser visto na Figura 7.

A mecânica de “desparafusar” comporta-se da seguinte forma: a classe parafuso diferencia seus objetos em tipos, no caso como “fenda” e “phillips”, no qual cada tipo de parafuso apenas vai interagir com a ponta correspondente que o jogador consegue trocar dentro do jogo. Quando o jogador acerta com sua arma o parafuso, se o tipo da arma e o tipo do parafuso forem compatíveis, o parafuso desaparece da tela, ou seja, foi “desparafusado”. Também é possível associar um parafuso a um objeto da classe `gate`, que, ao acertar o parafuso que o `gate` esteja apontando, o mesmo irá se abrir. Outro ponto forte da mecânica está na capacidade do jogador de se impulsionar para cima ao acertar o parafuso enquanto não estiver tocando no chão, trazendo para o jogo uma jogabilidade única e divertida.

Assim como os jogos desse gênero, como *Super Metroid* [18] e Castlevania – Symphony of The Night [19], é preciso ter inimigos, por conta do tempo de desenvolvimento, foi desenvolvido apenas um único inimigo base que pode ser visto

Fig. 6. A segunda ponta é a "phillips". Fornece um ataque de estocada, cria uma distância entre o jogador e os inimigos e causa mais dano que a "fenda".

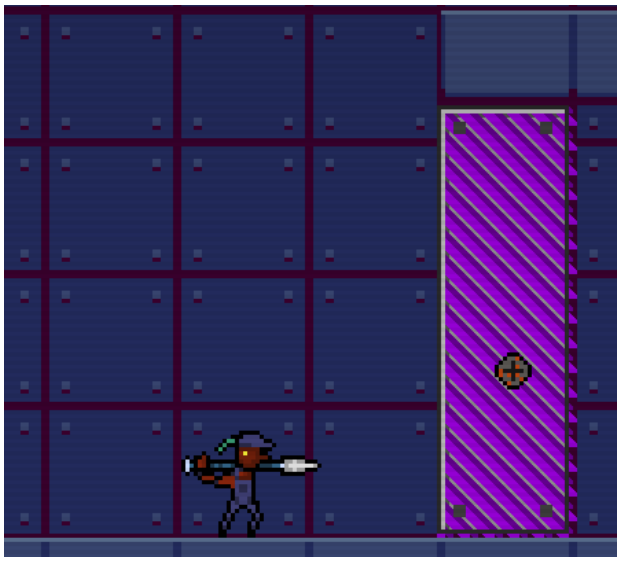

Fig. 7. A principal mecânica do jogo é a de "desparafusar", ao acertar o parafuso com a ponta certa, o jogador pode abrir novos caminhos.

na Figura 8, que avança quando o jogador entra em seu campo de visão e o ataca ao se aproximar.

Com os inimigos desenvolvidos e a mecânica de se impulsionar no ar acertando os parafusos aplicada, foi criado o chefe do jogo, que recebeu o nome de Punktauro, sua lógica é simples: ele persegue o jogador tentando acertá-lo com sua lança, seu único ponto vulnerável é a cabeça, devido à sua altura, o jogador precisa se impulsionar nos parafusos para acertá-lo, pode-se observar a luta na Figura 9.

Para suportar essas mecânicas complexas de forma organizada e escalável, foi adotada uma arquitetura de software baseada em princípios de orientação a objetos.

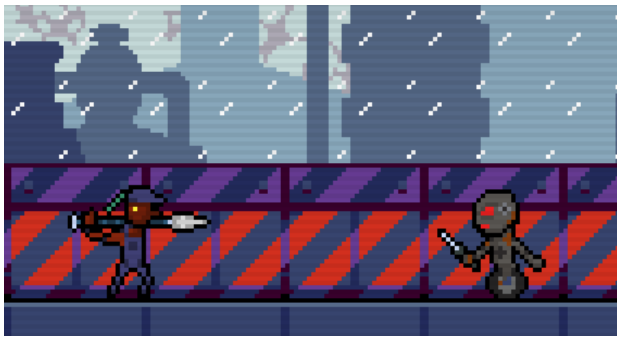

Fig. 8. O inimigo base corre em direção ao jogador e o ataca.

Fig. 9. Luta contra o Punktauro, o chefe do jogo demo.

B. Arquitetura do Jogo

O jogo adota uma arquitetura típica de projetos em C++. O diretório `include/` armazena os arquivos de cabeçalho responsáveis pelas declarações das classes e interfaces, enquanto o diretório `src/` contém as implementações dessas classes. As entidades principais, como `Player` e `Platform`, são modeladas com princípios de abstração e encapsulamento, separando a definição (`.h`) de sua implementação (`.cpp`).

A lógica do jogo é distribuída nos arquivos do diretório `src/`. Arquivos como `Player.cpp` lidam diretamente com as ações do jogador, enquanto outros, como `CollisionEngine.cpp`, são responsáveis pela detecção e resposta de colisões. A classe `GameManager` e *GameWorld* centralizam a criação e a interação dos objetos, funcionando como os orquestradores das entidades durante a execução.

C. Relacionamento das classes

A forma utilizada para diferenciar as entidades do jogo foi separá-las em "objetos estáticos" e "objetos dinâmicos", ou seja, os objetos que ficam fixos em sua posição e os objetos que atualizam sua posição. Primeiro cria-se a classe `Object`, que terá todas as características em comum entre as entidades. Ela então é herdada por `DynamicObject`, que é herdada por `Player`, `Chicken`, `Enemy` e `Punktauro`. Já `StaticObject`, é herdada por `Door`, `Platform`, `SolidPlatform`, `Wall`, `Screw` e `Decoration`.

A principal interação entre os objetos ocorre na `CollisionEngine`, que detecta as colisões que ocorrem entre os objetos dinâmicos com os objetos estáticos e faz os tratamentos adequados para cada colisão. Também há outras colisões tratadas, como a interação entre `Player` e `Door`. É possível observar tal relação na Figura 10.

Para o aspecto visual das entidades, foi criada a classe `Sprite`, que recebe um conjunto de *sprites* que representam aquela entidade e a classe `Animation`, que recebe *sprite* e percorre aquele conjunto de *sprites*, causando o efeito de animação. A classe *Player* possui um total de 15 *sprites* organizados em um único arquivo `.png`, a classe `Animation` percorre por cada *sprite* através das coordenadas de cada um deles presentes nesse arquivo, (observar a Figura 11). O jogador possui 5 animações diferentes: Parado, Correndo, Batendo, Batendo forte e pulando. Com as animações feitas, basta criar os eventos que ativam cada uma dessas animações. A de correr, por exemplo, sempre acontece quando o personagem estiver atualizando sua posição em X, a de pulo sempre que ele estiver

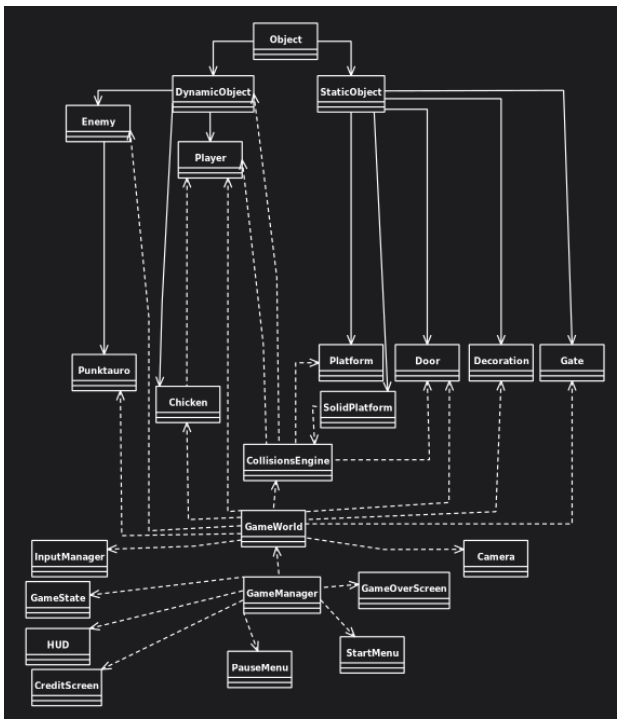

Fig. 10. Diagrama que representa algumas das classes do jogo A Rusty Screw.

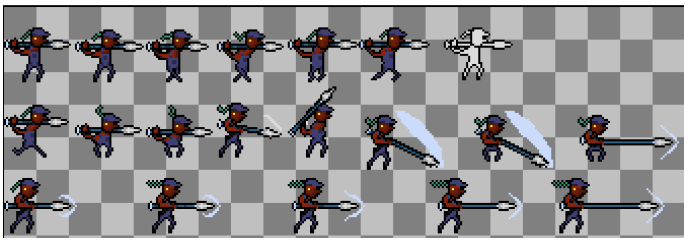

Fig. 11. Para melhor desempenho, os sprites concentram-se em um único arquivo png.

atualizando sua posição em Y e não estiver tocando no chão, a parado quando nenhuma das posições for atualizada.

V. Análise e Avaliação dos Resultados

O desenvolvimento do estudo de caso permitiu a validação de diversos conceitos de desenvolvimento de jogos digitais sem o uso de *TPEs*, focando na implementação manual de mecânicas, física e renderização utilizando a biblioteca SDL. Entre os principais avanços, destacam-se:

- Código modularizado, adotou-se princípios como abstração e encapsulamento, permitindo a manutenção e adição de novas funcionalidades.
- Implementação da física de colisões a partir da classe `CollisionEngine`, o sistema de detecção e resposta a colisões foi aprimorado, garantindo interações mais precisas entre entidades.
- Adição de mecânicas de movimentação como a de `Wall Jump` e de "desparafusar".
- Sistema de troca de pontas da arma.
- Sistema de animação eficiente.
- Criação de Inimigos.
- Batalha de Chefe.

Durante o desenvolvimento dessa pesquisa, para melhor entendimento das Arquiteturas de Software, foram criadas três versões diferentes de um jogo simples de terminal, caso haja interesse em conhecer, acesse o GitHub do projeto[6].

Com a demo do Jogo *A Rusty Screw* pronta – o código fonte pode ser acessado no GitHub[7] – foi criado um executável do jogo para Windows e enviado para voluntários experimentarem e responderem um questionário sobre o que acharam do jogo. As perguntas foram:

1) Com que frequência você costuma jogar videogames? (Figura 12)
2) Qual sua familiaridade com jogos do gênero Metroidvania (como *Hollow Knight*, *Super Metroid*, *Castlevania: SOTN*)? (Figura 13)

[6]https://github.com/netorapg/cli_cpp_games
[7]https://github.com/netorapg/A-Rusty-Screw

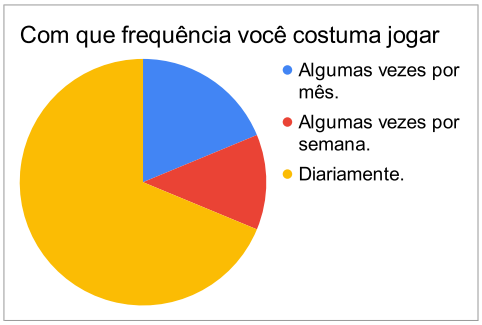

Fig. 12. Frequência que os participantes jogam videogames.

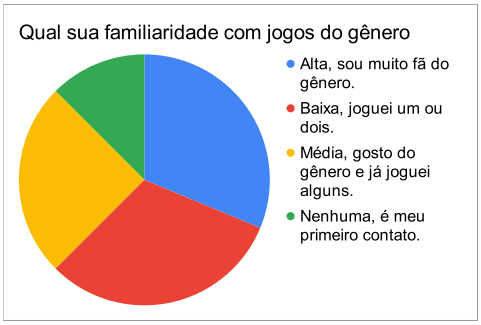

Fig. 13. Familiaridade dos jogadores com o gênero Metroidvania.

3) Você gostou do jogo? (Figura 14)
4) O quanto você gostou do jogo? (Figura 15)
5) Descreva o que mais você gostou e/ou não gostou do jogo, pontue também o que poderia melhorar.
6) O que você achou da mecânica de "desparafusar" acertar parafusos no ar para se impulsionar e abrir portas? (Figura 16)
7) O que achou da luta contra o Punktauro? (Figura 17)
8) Em uma escala de 1 a 5, o quão intuitivos foram os controles básicos do personagem (andar, pular, atacar)? (Figura 18)
9) O que você achou da mecânica de Wall Jump (salto na parede)? (Figura 19)
10) A mecânica de combate foi satisfatória? (Figura 20)

Até o momento da escrita deste artigo, 16 respostas foram enviadas, que podem ser observadas a seguir:

Descreva o que mais você gostou e/ou não gostou do jogo, pontue também o que poderia melhorar.

Com base nos comentários, o jogo foi amplamente elogiado por sua estética, incluindo a pixel art, o design dos personagens e as animações, que criam um ambiente agradável e visualmente coeso. A premissa e as mecânicas centrais, como o sistema de troca de ferramentas para interagir com parafusos e resolver quebra-cabeças, foram consideradas divertidas e

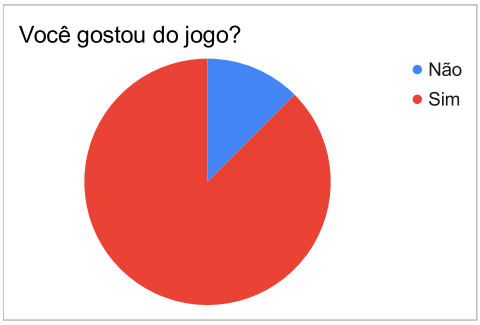

Fig. 14. Se os jogadores gostaram ou não do jogo.

Fig. 15. O quanto que os participantes gostaram do jogo.

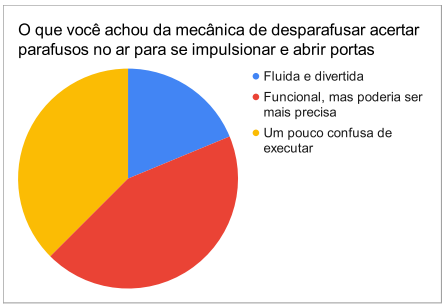

Fig. 16. Sobre a mecânica de "desparafusar".

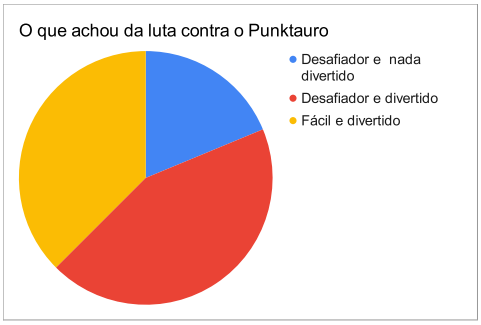

Fig. 17. O que acharam da luta contra o Chefe.

promissoras. A movimentação base e elementos como o *wall jump* também receberam *feedback* positivo.

No entanto, um ponto crítico recorrente foi a confusão e a falta de clareza, especialmente para novos jogadores. Muitos sentiram-se perdidos em relação ao caminho a seguir, às plataformas interativas e à ausência de um tutorial que explicasse as mecânicas básicas. O combate foi outro ponto de insatisfação comum, descrito como "truncado" devido à falta de *knockback* (recuo) nos inimigos, o que torna os confrontos pouco dinâmicos. A batalha contra o chefe, embora interessante, foi criticada por não indicar claramente seu ponto fraco, ter picos de dificuldade desiguais entre suas fases e apresentar efeitos sonoros muito altos. As sugestões de melhoria focam, portanto, em adicionar mais guias para o jogador, refinar o sistema de combate com melhor *feedback* de impacto e aprimorar a clareza e progressão da luta contra o chefe.

Ao analisar a pesquisa com os usuários, é possível conectar diretamente o *feedback* da experiência de jogo à questão de pesquisa central deste trabalho. Por um lado, os elogios à estética e às mecânicas de movimentação, como o *wall jump*, validam as boas práticas de se ter controle total sobre o código, permitindo a criação de sistemas fluidos e customizados. Por outro lado, as críticas ao combate "truncado" pela falta de *knockback* exemplificam um dos *principais desafios técnicos da abordagem*: a complexidade de se programar manualmente cada aspecto do jogo e as interações físicas que uma *TPE* frequentemente simplifica. Assim, os resultados não apenas mostram que é possível criar jogos sem TPEs, mas também servem como um espelho prático dos benefícios e obstáculos inerentes a este caminho de desenvolvimento.

VI. Conclusões e Trabalhos Futuros

Este trabalho apresenta o processo de desenvolvimento de um jogo do gênero Metroidvania, utilizando C++ e a biblioteca SDL, com o objetivo de explorar a criação de jogos digitais sem o auxílio de *TPEs*. A jornada se demonstrou desafiadora, porém enriquecedora do ponto de vista técnico, permitindo aprofundar em conceitos fundamentais como gerenciamento de estados, física de colisões, renderização e controle de entrada.

Os resultados validam a proposta inicial, destacando a viabilidade técnica da construção de um jogo funcional a partir de ferramentas de baixo nível A evolução da arquitetura do código e a implementação de mecânicas robustas, como movimentação, colisão e animações, reforçam essa constatação. Vale ressaltar que a pesquisa limitou-se aos alunos do campus. Para ampliar a busca por resultados, é interessante apresentar o jogo em eventos ou hospedá-lo em plataformas *online*.

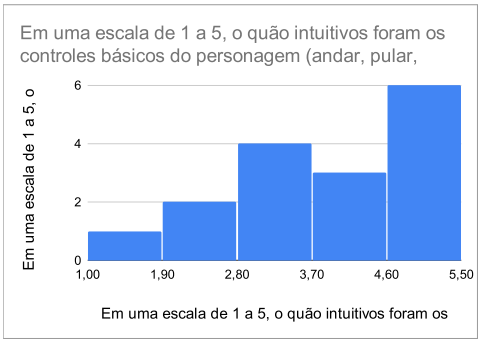

Fig. 18. Sobre os controles básicos do personagem.

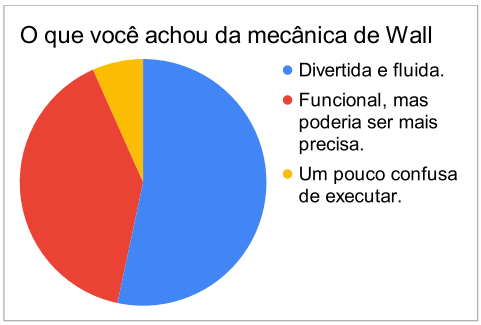

Fig. 19. Sobre a mecânica de Wall Jump.

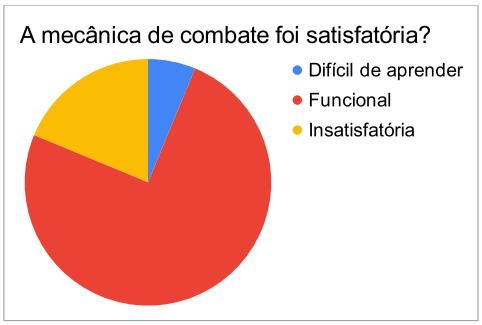

Fig. 20. Sobre se o combate foi satisfatório.

A experiência adquirida durante o desenvolvimento, aliada à valiosa avaliação dos jogadores, não apenas consolidou o aprendizado, mas também abriu novos horizontes para a continuidade do projeto. Como trabalhos futuros, destacam-se as seguintes possibilidades de aprofundamento:

- **Expansão do conteúdo:** Criação de novos cenários, áreas exploráveis e desafios, além da implementação de novos inimigos com padrões de comportamento diversos.
- **Refinamento da jogabilidade:** Melhorias no sistema de combate, com a implementação de um *knockback*, e no sistema de "desparafusar", permitindo o uso em quatro direções, além de aprimoramentos na HUD.
- **Generalização do código:** Exploração do reaproveitamento da base de código para a criação de novos jogos, caminhando para a estruturação de uma *engine* própria.

Diante do exposto, conclui-se que o desenvolvimento de jogos sem *TPEs*, além de ser uma abordagem viável, representa uma excelente oportunidade para o aprofundamento prático e teórico nas bases da engenharia de software aplicada a jogos.

Referências

[1] A. Giannini, "A aposta da Riot Games no brasil para liderar o mercado de games na américa latina," Veja, 2024, acesso em: 25 maio 2025. [Online]. Available: https://veja.abril.com.br/economia/a-aposta-da-riot-games-no-brasil-para-liderar-o-mercado-de-games-na-america-latina
[2] I. L. M. Ricarte, *Programação Orientada a Objetos: Desenvolvimento Avançado em C++*. Campinas, SP: Departamento de Engenharia de Computação e Automação Industrial, UNICAMP, 1995.
[3] R. Nystrom, *Game Programming Patterns*. Genever Benning, 2014.
[4] S. Mitchell, *SDL Game Development*. Packt Publishing, 2013.
[5] Mendix, "O que é arquitetura baseada em componentes?" 2024, acesso em: 15 maio 2025. [Online]. Available: https://www-mendix-com.translate.goog/blog/what-is-component-based-architecture/?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc
[6] Amazon Web Services, "O que é arquitetura orientada a eventos (EDA)?" 2024, acesso em: 15 maio 2025. [Online]. Available: https://aws.amazon.com/pt/what-is/eda/
[7] Cabo, "How doom was programmed," 2019, acesso em: 8 de outubro 2025. [Online]. Available: https://medium.com/@chriscable_70151/how-doom-was-programmed-e510a178929c
[8] K. T. Jensen, "25 anos depois: a história do irreal e de uma dinastia épica," 2023, acesso em: 26 setembro 2025. [Online]. Available: https://www.pcmag.com/news/25-years-later-the-history-of-unreal-and-an-epic-dynasty
[9] M. Marques, "Gamemaker: uma engine especializada no desenvolvimento de jogos indie 2d," 2022, acesso em: 26 setembro 2025. [Online]. Available: https://warpzone.me/gamemaker-uma-engine-especializada-no-desenvolvimento-de-jogos-indie-2d/
[10] Douglas, "Como surgiu a unity engine?" 2022, acesso em: 26 setembro 2025. [Online]. Available: https://www.crieseusjogos.com.br/como-surgiu-a-unity-engine/
[11] F. Vinha, "Stardew valley vende mais do que Tetris e Zelda no mundo todo," Omelete, 2024, acesso em: 23 maio 2025. [Online]. Available: https://www.omelete.com.br/games/stardew-valley-vende-mais-do-que-tetris-e-zelda-no-mundo-todo
[12] ConcernedApe, "Stardew valley," Steam, 2016, acesso em: 15 maio 2025. [Online]. Available: https://store.steampowered.com/app/413150/Stardew_Valley/
[13] J. Schreier, *Sangue, Suor e Pixels: Os Dramas, as Vitórias e as Curiosas Histórias por Trás dos Videogames*. HarperCollins Brasil, 2018.
[14] Spliced Online, "What engine is Dwarf Fortress made in?" 2025, acesso em: 15 maio 2025. [Online]. Available: https://splicedonline.com/what-engine-is-dwarf-fortress-made-in/?utm_source=chatgpt.com
[15] Wube Software LTD, "Factorio," Steam, 2016, acesso em: 28 maio 2025. [Online]. Available: https://store.steampowered.com/app/427520/Factorio/?l=portuguese
[16] jiri, "Friday facts #230 - engine modernisation," Factorio Blog, 2018, acesso em: 28 maio 2025. [Online]. Available: https://factorio.com/blog/post/fff-230
[17] Factorio Wiki, "Map generator," 2020, acesso em: 28 maio 2025. [Online]. Available: https://wiki.factorio.com/Map_generator/pt-br
[18] Nintendo and Intelligent Systems, "Super Metroid [jogo eletrônico]," Super Nintendo Entertainment System (SNES), 1994.
[19] Konami, "Castlevania: Symphony of the Night [jogo eletrônico]," Playstation, 1997.