Explorando C++ e SDL na Criação de um Jogo 2D do Gênero Metroidvania

Renato Augusto Platz Guimarães Neto[1], **Eduardo Henrique Molina da Cruz**[1]

[1]Campus Paranavaí – Instituto Federal do Paraná (IFPR) – Paranavaí, PR, Brasil

renatoplatz11@gmail.com, eduardo.cruz@ifpr.edu.br

Abstract. *This work explores the development of digital games in C++ with the SDL library, adopting an approach that dispenses with third-party game engines (TPEs). Through the practical construction of a Metroidvania game, we detail the manual implementation of essential systems, such as physics and animation. User testing validated the approach, demonstrating a balance between the creative freedom to develop original mechanics and the inherent difficulty of refining complex systems such as combat. We conclude that low-level development represents a robust and valuable alternative for training game programmers, solidifying fundamental technical knowledge in the field.*

Resumo. *Este trabalho explora o desenvolvimento de jogos digitais em C++ com a biblioteca SDL, adotando uma abordagem que prescinde de motores de jogo de terceiros (TPEs). Por meio da construção prática de um jogo Metroidvania, detalhamos a implementação de sistemas essenciais, como física e animação, de forma manual. Testes com usuários validaram a abordagem, demonstrando um equilíbrio entre a liberdade criativa para desenvolver mecânicas originais e a dificuldade inerente ao refinamento de sistemas complexos como o combate. Conclui-se que o desenvolvimento de baixo nível representa uma alternativa robusta e de grande valor para a formação de programadores de jogos, solidificando o conhecimento técnico fundamental da área.*

1. Introdução

O mercado de jogos digitais, que em 2024 alcançou uma receita de quase 188 bilhões de dólares, superando as indústrias de cinema e música combinadas [Giannini 2024], tornou-se um campo de vastas oportunidades para engenheiros de software. A porta de entrada para muitos aspirantes são as third-party engines (TPEs), ferramentas que democratizaram a criação de jogos. Embora essenciais, especialmente para equipes independentes, as TPEs abstraem funcionalidades cruciais como motores gráficos e de física, o que pode dificultar que iniciantes desenvolvam uma compreensão sólida dos fundamentos da programação de jogos. A proficiência nessas ferramentas é maximizada quando o desenvolvedor já possui uma base técnica robusta.

Para preencher essa lacuna, este trabalho investiga o desenvolvimento de jogos sem o auxílio de uma TPE, utilizando a linguagem C++ e a biblioteca de código aberto Simple DirectMedia Layer (SDL). O objetivo é identificar os desafios — como gerenciamento de memória e arquitetura de código — e as vantagens — como flexibilidade e

otimização de desempenho — dessa abordagem de baixo nível. Assim, este estudo é guiado pela 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 investigação será conduzida através de um estudo de caso prático: a implementação de um jogo do gênero Metroidvania. A complexidade inerente a este estilo, que demanda sistemas de física, movimentação variada e gerenciamento de estados, o torna um campo ideal para analisar e validar práticas de desenvolvimento eficazes.

Este artigo está organizado da seguinte forma. A Seção 2 discute TPEs e trabalhos correlatos. A Seção 3 detalha o desenvolvimento do estudo de caso. A Seção 4 avalia os resultados obtidos. Por fim, a Seção 5 apresenta as conclusões e aponta direções para pesquisas futuras.

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

No início da indústria, as *engines* não eram separadas dos jogos. Cada estúdio desenvolvia seus próprios sistemas específicos, feitos sob medida para cada título. Contudo, graças 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), 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, foi lançada junto com o jogo Unreal em 1998 [Jensen 2023]. 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, que então evoluiu para uma das *engines* mais importantes do mercado [Marques 2022]. 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 [Douglas 2022]. 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 *Tripple A*[1] quanto *Indie*[2], ainda são lançados vários jogos com *engines* customizadas.[3] Existem várias vantagens ainda em

[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*

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.

2.1. Comparativo entre Third-Party Engines

As *third-party engines* 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 engines no mercado, cada uma com foco e propostas diferentes, além de licenças diferentes. Observe as Tabelas 1 e 2, onde são comparadas as Engines Godot, Love2d, Unity e Unreal Engine, que estão entre as mais conhecidas do mercado de Thirdy-Party Engines.

Tabela 1. Comparativo de Engines (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 2. Comparativo de Engines (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

customizadas, desde jogos de estúdios grandes, até jogos feitos por uma única pessoa https://gist.github.com/raysan5/909dc6cf33ed40223eb0dfe625c0de74

2.1.1. Thirdy-Party Engine para jogos 2D ou projetos 3D com estilização "indie"

A **Godot Engine** se destaca. Seus recursos 2D são considerados "Excelentes", 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*.

2.1.2. 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 controle total do código, aprender os fundamentos da criação de jogos ou simplesmente criar um protótipo funcional em pouco tempo, focando na lógica 2D. Dois jogos de destaque são *Balatro* e *Gravity Circuit*.

2.1.3. Thirdy-Party Engine para projetos multiplataforma (Mobile, PC, Consoles) e jogos 3D versáteis

A **Unity** é, historicamente, 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 dezenas de plataformas diferentes. 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*.

2.1.4. Thirdy-Party Engine para 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 grandes estúdios que buscam o limite do que é visualmente possível. O custo dessa potência é uma maior complexidade e recursos 2D mais "Limitados", mostrando que seu foco é claramente outro. Dois jogos de destaque são *Lies of P* e *Mortal Kombat 1*.

As *TPEs* são ferramentas extremamente poderosas 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.

2.2. Custom Engines e Trabalhos Relacionados

Desenvolver um jogo sem o uso de *engines* 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, serão abordados alguns exemplos de jogos desenvolvidos sem o uso de *TPEs*.

Um dos jogos mais famosos que não utiliza uma *TPE* é **Stardew Valley**. Buscando desesperadamente um emprego, Eric Barone decidiu desenvolver este 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 [Vinha 2024], possui nota 89 no *Metacritic* e mais de 753.746 avaliações extremamente positivas na plataforma Steam [ConcernedApe 2016]. *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 até construir relacionamentos 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 [Schreier 2018], acreditava que aprenderia muito mais e poderia construir cada parte do projeto à sua maneira[4]. Posteriormente, o jogo foi migrado para a *engine* MonoGame, permitindo sua expansão para outras plataformas.

Outro jogo muito famoso que não usa *TPE* é **Dwarf Fortress**. 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 teve sua primeira versão lançada em 2006, sendo programado nas linguagens *C* e *C++*.

Segundo *Spliced Online* [Spliced Online 2025], 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 altamente 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.

O jogo *Factorio*, 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 [Wube Software LTD 2016]. 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 [jiri 2018].

Factorio é um jogo extremamente 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 levaria cerca de 240 minutos de jogo para alcançar a fronteira saindo do centro. Isso significa que o mundo é infinito [Factorio Wiki 2020].

[4]Além da programação, Barone também foi responsável pela criação das artes e trilhas sonoras do jogo.
[5]`https://www.bay12games.com/dwarves/index.html`

2.3. 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 e detalhados já criados. Já *Factorio* se destaca por manter um nível de complexidade extremamente alto, aliado a uma otimização excepcional. Da mesma forma, no desenvolvimento do estudo de caso do jogo desenvolvido neste trabalho, observa-se benefícios semelhantes: maior liberdade e controle na criação das mecânicas, além da modularidade, manutenibilidade e otimização.

3. Desenvolvimento do jogo *A RUSTY SCREW*

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

3.1. Adesão ao gênero metroidvania

O jogo *A Rusty Screw*, observado na Figura 1 – 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 habilidades e 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, 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 2 e 3 — mas também para resolver quebra-cabeças ou remover obstáculos no cenário, como pode ser visto na Figura 4.

Assim como os jogos desse gênero, como *Super Metroid* [Nintendo and Intelligent Systems 1994] e Castlevania – Symphony of The Night

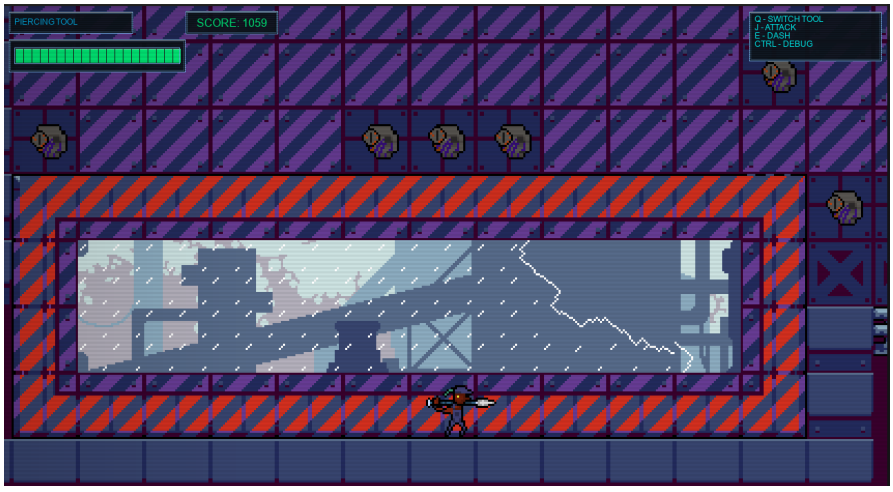

Figura 1. Jogo A Rusty Screw.

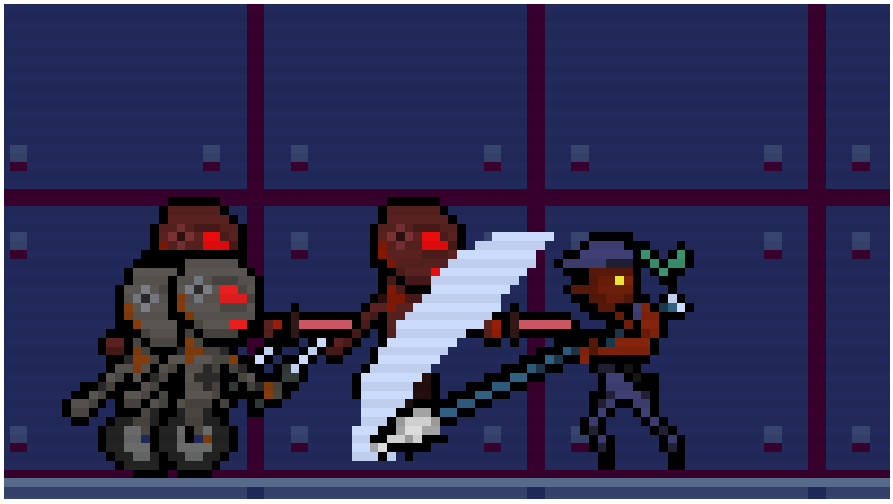

Figura 2. A ponta inicial do jogador é a de "fenda", que permite um ataque cortante.

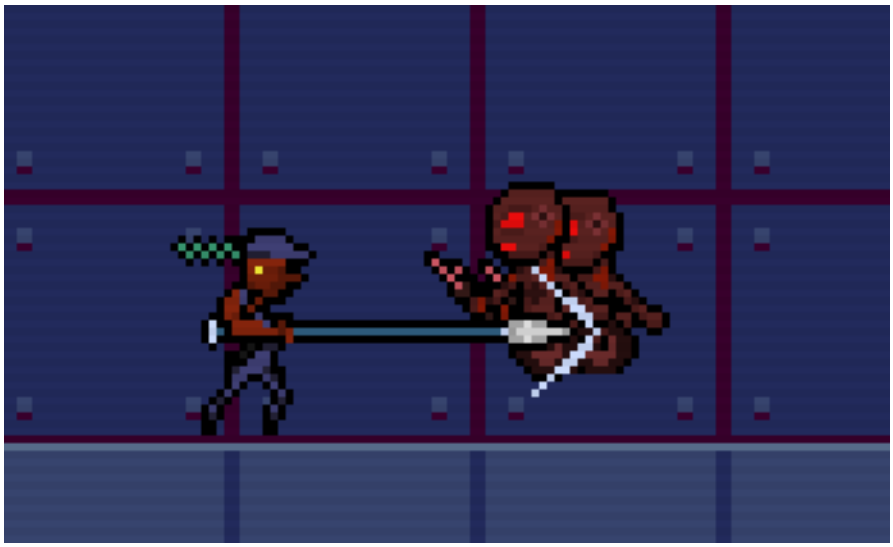

Figura 3. A segunda ponta é a de "phillips", ela fornece um ataque de estocada, criando uma distância entre o jogador e os inimigos, além de causar mais dano que a de "fenda".

[Konami 1997], é preciso ter inimigos, por conta do tempo de desenvolvimento, foi desenvolvido apenas um único inimigo base que pode ser visto na Figura 5, 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 6.

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

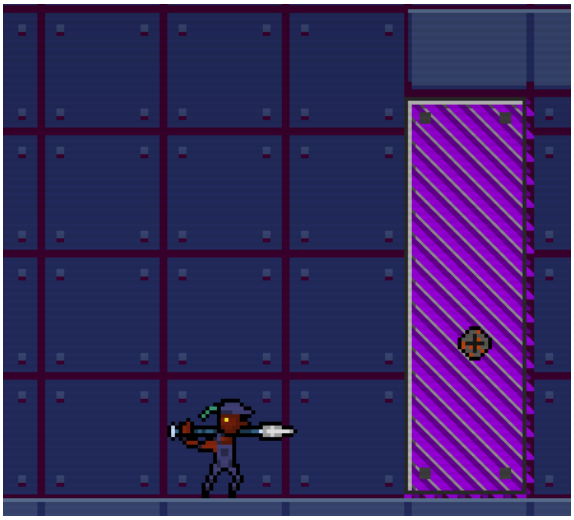

Figura 4. A principal mecânica do jogo é a de "desparafusar", ao acertar o parafuso com a ponta certa, o jogador pode abrir novos caminhos.

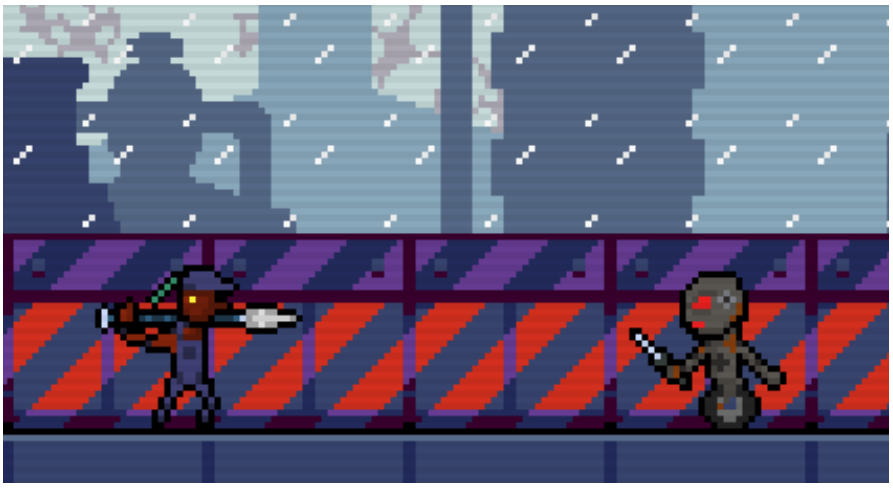

Figura 5. O inimigo base, ao ver o jogador, corre em sua direção e o ataca.

Figura 6. Luta contra o Punktauro, o chefe do jogo demo.

3.2. Arquitetura do Jogo e Relacionamento das classes

O jogo adota uma arquitetura típica de projetos em C++. 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.

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 será herdada por `Player`, `Chicken`, `Enemy` e `Punktauro`. Já `StaticObject`, será herdada por `Door`, `Platform`, `SolidPlatform`, `Wall`, `Screw` e `Decoration`.

Além de criar essa entidades, elas precisam interagir entre sí. Tal interação ocorre na `CollisionEngine`, que detectará as colisões que ocorrem entre os objetos dinâmicos com os objetos que herdam de `StaticObject` e fará 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 7.

Para o aspecto visual das entidades, foi criada a classe `Sprite`, que receberá 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

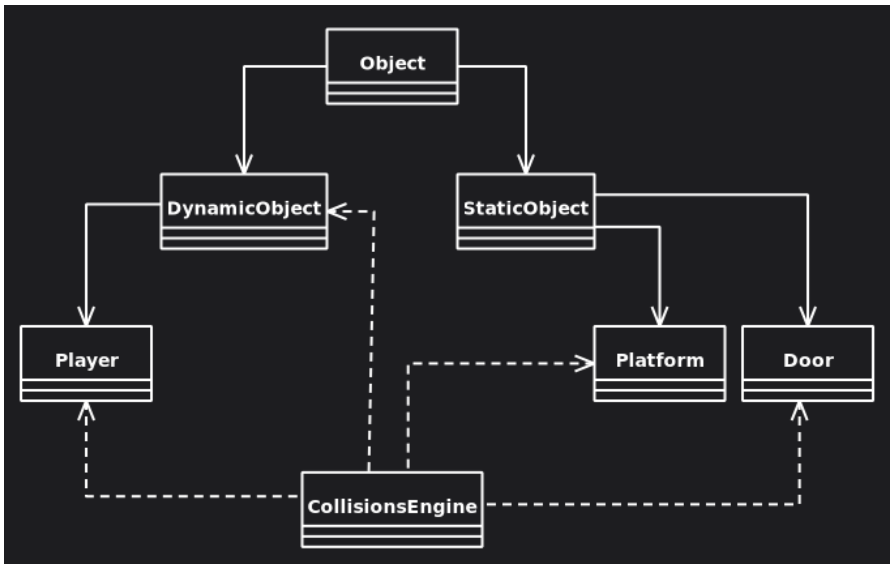

Figura 7. Diagrama que representa algumas das classes do jogo A Rusty Screw e seu relacionamento.

presentes nesse arquivo, (observar a Figura 8). O jogador possui 5 animações diferentes: Parado, Correndo, Batendo, Batendo forte e pulando. Com as animações feitas, basta criar os eventos que ativarão 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 atualizando sua posição em Y e não estiver tocando no chão, a parado quando nenhuma das posições for atualizada.

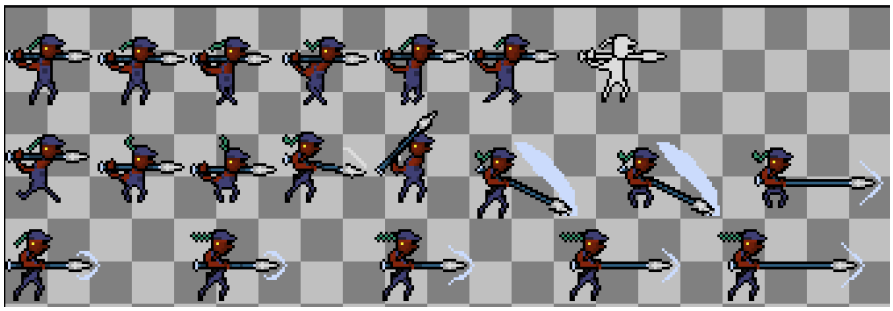

Figura 8. Para melhor desempenho, todos os sprites são concentrados em um único arquivo png.

4. 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:

- O código foi modularizado, adotando princípios como abstração e encapsulamento, o que facilitou a manutenção e adição de novas funcionalidades.
- Implementação robusta 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 divertidas 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, cada versão sendo feita em uma das arquiteturas e uma apenas seguindo com o paradigma POO, 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.

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

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 o *game feel* e as interações físicas que uma *TPE* frequentemente simplifica. Assim, os resultados não apenas mostram que é possível criar jogos divertidos sem TPEs, mas também servem como um espelho prático dos benefícios e obstáculos inerentes a este caminho de desenvolvimento.

5. Conclusões e Trabalhos Futuros

Este trabalho apresentou 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 extremamente enriquecedora do ponto de vista educacional e técnico, permitindo aprofundar em conceitos fundamentais como gerenciamento de estados, física de colisões, renderização e controle de entrada.

Os resultados obtidos ao longo do projeto 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.

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 para maior clareza.
- **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

ConcernedApe (2016). Stardew valley. `https://store.steampowered.com/app/413150/Stardew_Valley/`. Acesso em: 15 maio 2025.

Douglas (2022). Como surgiu a unity engine? Crie Seus Jogos. Disponível em: `https://www.crieseusjogos.com.br/como-surgiu-a-unity-engine/`. Acesso em: 26 setembro 2025.

Factorio Wiki (2020). Map generator. Factorio Wiki. Disponível em: https://wiki.factorio.com/Map_generator/pt-br. Acesso em: 28 maio 2025.

Giannini, A. (2024). A aposta da Riot Games no brasil para liderar o mercado de games na américa latina. Veja. Disponível em: https://veja.abril.com.br/economia/a-aposta-da-riot-games-no-brasil-para-liderar-o-mercado-de-games-na-america-latina. Acesso em: 25 maio 2025.

Jensen, K. T. (2023). 25 anos depois: a história do irreal e de uma dinastia épica. PcMag. Disponível em: https://www.pcmag.com/news/25-years-later-the-history-of-unreal-and-an-epic-dynasty. Acesso em: 26 setembro 2025.

jiri (2018). Friday facts #230 - engine modernisation. Factorio. Disponível em: https://factorio.com/blog/post/fff-230. Acesso em: 28 maio 2025.

Konami (1997). Castlevania: Symphony of the Night [jogo eletrônico]. Playstation.

Marques, M. (2022). Gamemaker: uma engine especializada no desenvolvimento de jogos indie 2d. Warpzone. Disponível em: https://warpzone.me/gamemaker-uma-engine-especializada-no-desenvolvimento-de-jogos-indie-2d/. Acesso em: 26 setembro 2025.

Nintendo and Intelligent Systems (1994). Super Metroid [jogo eletrônico]. Super Nintendo Entertainment System (SNES).

Schreier, J. (2018). *Sangue, Suor e Pixels: Os Dramas, as Vitórias e as Curiosas Histórias por Trás dos Videogames*. HarperCollins Brasil.

Spliced Online (2025). What engine is Dwarf Fortress made in? Spliced Online. Disponível em: https://splicedonline.com/what-engine-is-dwarf-fortress-made-in/?utm_source=chatgpt.com. Acesso em: 15 maio 2025.

Vinha, F. (2024). Stardew valley vende mais do que Tetris e Zelda no mundo todo. https://www.omelete.com.br/games/stardew-valley-vende-mais-do-que-tetris-e-zelda-no-mundo-todo. Acesso em: 23 maio 2025.

Wube Software LTD (2016). Factorio. https://store.steampowered.com/app/427520/Factorio/?l=portuguese. Acesso em: 28 maio 2025.