Lampiões: Um relato de experiência do desenvolvimento de um Jogo 2D de Faroeste Feito em C++ e SDL

Lucas Henrique Teixeira Ribeiro[1], **Mateus Stainer Rakoski**[1], **Arthur Hydeki Ricken**[1], **Evanise Araújo Caldas Ruiz**[1], **Eduardo Henrique Molina da Cruz**[1]

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

{20230006361, 20230006272, 20230006380}@estudantes.ifpr.edu.br
evanise.ruiz@ifpr.edu.br, eduardo.cruz@ifpr.edu.br

Abstract. *This paper is an experience report on the development of Lampiões, a 2D side-scrolling western game written in C++23 with SDL2 and the Box2D physics library, without a complete game engine. We describe the architectural decisions needed to integrate the physics world with game logic: an object hierarchy built around physical bodies, a scene stack for menus and pause, a deferred contact queue for collision handling, and a looping terrain produced by summing sine waves. We also report the manual verification of the main features, the known defects, and informal feedback gathered at a campus course fair.*

Resumo. *Este trabalho é um relato de experiência sobre o desenvolvimento de Lampiões, um jogo 2D de rolagem lateral com temática de faroeste, escrito em C++23 com SDL2 e a biblioteca de física Box2D, sem uma engine completa de jogos. Descrevemos as decisões de arquitetura necessárias para integrar o mundo físico à lógica do jogo: uma hierarquia de objetos construída em torno dos corpos físicos, uma pilha de cenas para menus e pausa, uma fila de contatos processada após o passo de simulação e um terreno cíclico obtido pela soma de senoides. Relatamos também a verificação manual das funcionalidades, os defeitos conhecidos e os comentários informais recebidos em uma mostra de cursos do campus.*

1. Introdução

Engines completas de jogos, também chamadas de *third-party engines* (TPE), como Unity, Godot e Unreal, são uma porta de entrada comum para quem começa a desenvolver jogos. Essas ferramentas oferecem editor, renderização, física, áudio e exportação para várias plataformas, e permitem obter um protótipo funcional em pouco tempo. Uma limitação dessa abordagem, do ponto de vista da formação em Engenharia de Software, é que parte das decisões de arquitetura fica oculta: a estrutura do laço principal, a ordem de atualização dos objetos, a posse dos recursos e o momento em que eles são liberados passam a ser definidos pela *engine*.

Em um trabalho anterior apresentado nesta mesma semana acadêmica, guimaraes2025 mostraram que é viável construir um jogo do gênero *Metroidvania* em C++ com SDL, implementando manualmente a detecção de colisões e a animação. Aquele relato

apontou o combate e a sensação física (*game feel*) como os aspectos mais difíceis de ajustar sem uma TPE. A partir disso, formulamos uma pergunta diferente: o que muda quando a física não é escrita pela equipe, mas delegada a uma biblioteca especializada, mantendo o restante da arquitetura sob controle do projeto?

Para investigar essa pergunta, desenvolvemos *Lampiões*, um jogo 2D de faroeste em que o jogador percorre um terreno acidentado recolhendo o gado espalhado pelo mapa enquanto enfrenta bandidos armados. O jogo usa C++23, SDL2 [SDL Community 2026] para janela, entrada e renderização, as extensões SDL_image, SDL_mixer e SDL_ttf para imagens, som e texto, e o Box2D [Catto 2020] para a simulação física. A questão que orienta o trabalho é: *que decisões de arquitetura um projeto sem uma engine completa precisa tomar para integrar um motor de física externo à lógica de jogo, e quais dificuldades aparecem nesse processo?*

A contribuição deste artigo não está no jogo em si, mas no relato das decisões de arquitetura, das alterações feitas ao longo do desenvolvimento e das limitações que permaneceram. Esperamos que esse registro seja útil a estudantes e desenvolvedores que queiram compreender o que uma *engine* normalmente resolve de forma implícita.

O restante do texto está organizado da seguinte forma. A Seção 2 apresenta os conceitos e trabalhos relacionados. A Seção 3 descreve o procedimento de desenvolvimento e de verificação. A Seção 4 descreve o jogo e sua arquitetura. A Seção 5 discute os resultados e as limitações, e a Seção 6 apresenta as conclusões e os trabalhos futuros.

2. Fundamentação e Trabalhos Relacionados

2.1. Bibliotecas *versus engines*

Uma *engine* completa impõe uma estrutura: ela chama o código do desenvolvedor nos momentos que define. Uma biblioteca, ao contrário, é chamada pelo código do desenvolvedor quando ele decide. A SDL pertence ao segundo grupo: ela abstrai o sistema operacional (janela, teclado, áudio, aceleração gráfica), mas não define como o jogo deve ser organizado. gregory2018 descreve uma *engine* como um conjunto de camadas, da plataforma até os sistemas de alto nível de jogabilidade; ao trabalhar apenas com bibliotecas, a equipe precisa construir as camadas intermediárias.

2.2. Padrões de programação de jogos

nystrom2014 reúne padrões recorrentes no desenvolvimento de jogos, dos quais três foram centrais neste projeto. O *Game Loop* separa cada quadro em leitura de entrada, atualização e desenho, e usa o tempo decorrido para que o jogo não dependa da velocidade da máquina. O *Update Method* atribui a cada objeto do mundo a responsabilidade de se atualizar, cabendo ao gerenciador apenas percorrer a coleção. Por fim, o padrão *State*, catalogado por gamma1994, orienta a organização das telas do jogo como estados empilháveis.

Sobre o passo de tempo, fiedler2004 recomenda que simulações físicas avancem em intervalos fixos, acumulando o tempo real e executando quantos passos forem necessários, porque integradores numéricos se comportam de forma diferente conforme o tamanho do passo. Esse ponto é retomado na Seção 5.

2.3. Motores de física

Escrever um motor de física de corpos rígidos, mesmo em 2D, envolve detecção de colisão entre formas, resolução de contatos, atrito e integração do movimento [Millington 2010]. O Box2D, criado por Erin Catto, é uma biblioteca em C++ que trata esses problemas [Catto 2020]. Na versão utilizada (série 2.4), ele trabalha em unidades do Sistema Internacional (metros, quilogramas, segundos) e organiza a simulação em um *mundo* (`b2World`) que contém *corpos* (`b2Body`); cada corpo possui uma ou mais *fixações* (`b2Fixture`), que associam uma forma geométrica a propriedades como densidade e atrito. Uma fixação pode ser marcada como *sensor*; nesse caso, o Box2D informa a sobreposição, mas não produz resposta física de colisão.

2.4. Trabalhos relacionados

O trabalho mais próximo é o de guimaraes2025, que desenvolveu o jogo *A Rusty Screw* com C++ e SDL. Nele, a classe `CollisionEngine` foi escrita pela própria equipe e trata separadamente os pares de objetos estáticos e dinâmicos. Em *Lampiões*, a separação conceitual entre objetos estáticos e dinâmicos foi mantida, mas passou a corresponder aos tipos de corpo do Box2D, e a detecção de colisão deixou de ser implementada pela equipe. A Seção 5.4 compara as duas abordagens segundo critérios explícitos.

Quanto ao tema, jogos de faroeste com rolagem lateral e tiroteio têm presença antiga nos fliperamas, com títulos como *Gun.Smoke* [Capcom 1985] e *Sunset Riders* [Konami 1991]. O nome *Lampiões* e parte da arte, como o chapéu e o personagem correndo, combinam essa estética com referências ao cangaço nordestino.

3. Metodologia

Este trabalho é um relato de experiência de desenvolvimento. Não se propõe a comparar experimentalmente abordagens nem a medir a qualidade do jogo, mas a documentar as decisões de arquitetura tomadas, os problemas encontrados e as alterações que deles resultaram.

Equipe e período. O projeto foi desenvolvido entre fevereiro e setembro de 2026 por três estudantes, sob orientação docente. O código foi versionado no GitHub[1]: cada integrante trabalhou em um ramo próprio, e as integrações eram feitas por *merge* em um ramo de apresentação, conforme o fluxo descrito por chacon2014pro.

Etapas. O trabalho seguiu quatro etapas, com sobreposição entre elas: (i) *planejamento*, em fevereiro, com a definição da proposta do jogo e a escolha das bibliotecas; (ii) *implementação incremental*, em que cada funcionalidade (movimento, terreno, inimigos, colisões, menus) foi desenvolvida e integrada separadamente; (iii) *revisão*, em reuniões periódicas com o professor orientador, que motivaram refatorações como a reorganização da hierarquia de classes em maio; e (iv) *verificação*, descrita a seguir.

Registro das alterações. O histórico do repositório foi a principal fonte de registro: até o fim de setembro foram feitos 65 *commits*, cujas mensagens identificam as funcionalidades acrescentadas e as correções feitas após as revisões. Os problemas conhecidos relatados na Seção 5 foram levantados a partir desse histórico e da inspeção do código na versão final.

[1] `https://github.com/Lampioes/lampiao`

Critério de funcionamento. O jogo foi considerado funcional quando todos os cenários da Tabela 1 puderam ser executados do início ao fim sem encerramento inesperado do programa, com o comportamento esperado ou com desvios já identificados como defeitos conhecidos.

Ambiente. O jogo é compilado com CMake (versão mínima 3.20) no padrão C++23, com SDL2 e Box2D. Os testes foram executados em Linux Debian 12 Bookworm, pelo GCC, em resolução de 1920×1080 pixels.

Retorno de jogadores. Além da verificação técnica, o jogo foi disponibilizado em uma mostra de cursos do campus. Essa atividade foi informal, sem instrumento estruturado nem registro do número de participantes, e por isso é tratada apenas como retorno exploratório (Seção 5.5).

4. Desenvolvimento do Jogo *Lampiões*

4.1. Proposta de jogo

O jogador controla um vaqueiro que precisa recuperar dez cabeças de gado espalhadas pelo mapa. O terreno é contínuo e acidentado, com colinas e vales, e atravessa cinco zonas visuais: montanhas, uma transição entre montanha e deserto, o deserto de caliça, uma cidade e novamente o deserto. A cada cinco segundos pode surgir um bandido perto do jogador, até o limite de três simultâneos. Os bandidos atiram quando o jogador entra no alcance e precisam recarregar após cada disparo. O jogador tem cinco vidas, move-se com as setas, pula com a barra de espaço e atira com a tecla `Z` na direção em que está virado. Cada bandido derrotado vale cem pontos, e a partida é vencida ao recolher as dez vacas. A Figura 1 mostra uma tela da partida.

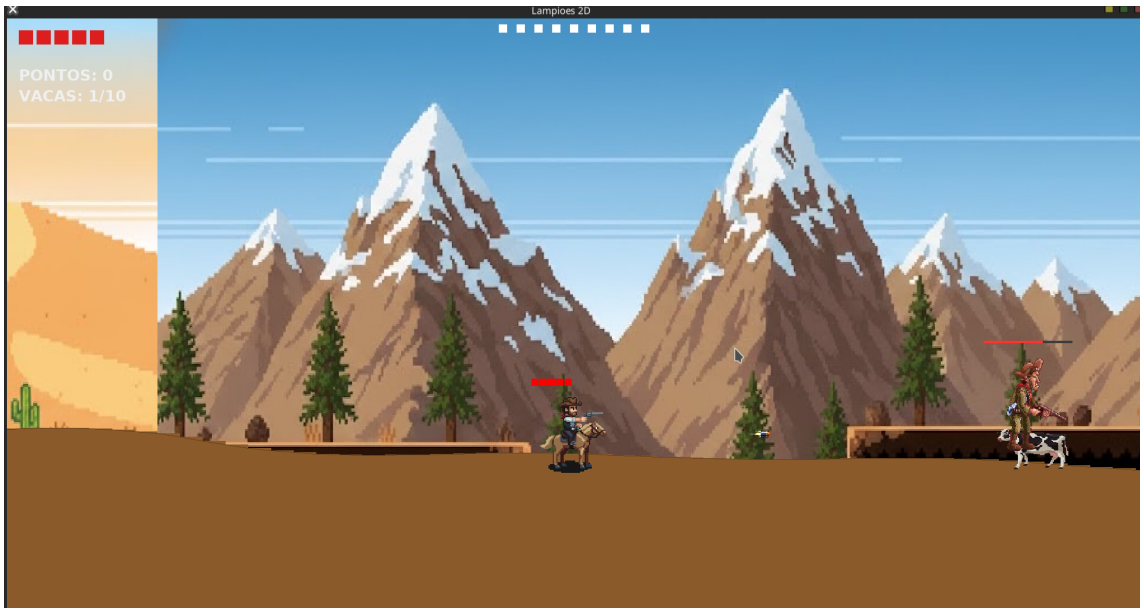


Figura 1. Tela de jogo de *Lampiões*, com o contador de vidas, a pontuação e o número de vacas recolhidas no canto superior esquerdo.

4.2. Laço principal e pilha de cenas

A classe `GameManager` inicializa as bibliotecas, carrega os recursos compartilhados (sprites, fontes e música de fundo) e executa o laço principal. A cada iteração, ela mede

o tempo decorrido com o contador de alta resolução da SDL, limita esse valor a 0,1 s para evitar saltos quando a janela é arrastada ou o sistema interrompe a execução por um instante, e então chama, em ordem, o tratamento de eventos, a atualização e o desenho.

O `GameManager` não conhece a partida em si. Ele mantém uma pilha de cenas, e toda cena herda de uma classe abstrata `Scene` com os métodos `handleEvent`, `update` e `draw`. Existem dois tipos concretos: `GameWorld`, que representa a partida, e `MenuScene`, que monta o menu principal, a tela de pausa ou a tela de controles, conforme os parâmetros de construção.

Para que nenhuma cena manipule a pilha diretamente, ela apenas registra um *pedido* (`PedidoCena`) com uma de quatro ações: empilhar, desempilhar, trocar ou sair. Depois de atualizar a cena do topo, o `GameManager` lê o pedido e o aplica. Quando o jogador pressiona `Esc` durante a partida, por exemplo, o `GameWorld` solicita o empilhamento da tela de pausa. A partida permanece na pilha e deixa de receber atualizações, mas seu estado é preservado. A opção *Continuar* desempilha a pausa, e a opção *Voltar ao menu* substitui a pilha inteira, destruindo a partida e liberando seu mundo físico.

A pausa e a tela de controles se declaram *transparentes*. No desenho, o `GameManager` procura, do topo para a base, a primeira cena opaca e desenha a partir dela. Assim, a partida congelada aparece ao fundo, escurecida, com o menu de pausa sobreposto, sem que nenhuma cena precise conhecer as cenas abaixo dela.

4.3. Hierarquia de objetos

Todo elemento da partida herda de `GameObject`, que define a interface comum: tratar eventos, atualizar, desenhar, informar se ainda está ativo e destruir seu corpo físico. Logo abaixo está `PhysicsObject`, que guarda o ponteiro para o `b2Body`. A partir dele, a hierarquia se divide em `StaticObject` e `DynamicObject`, que criam, respectivamente, corpos estáticos e dinâmicos do Box2D. A Figura 2 resume essa estrutura.

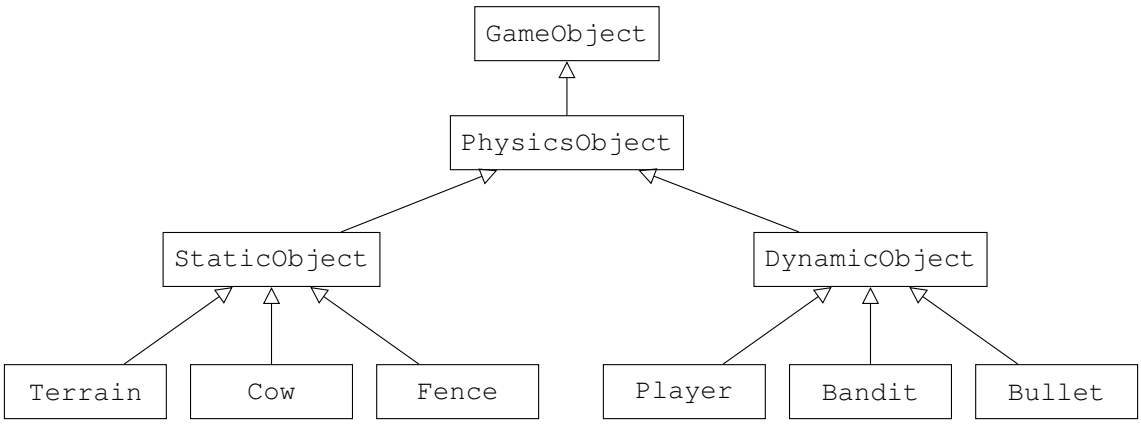


Figura 2. Hierarquia das entidades da partida. A separação entre objetos estáticos e dinâmicos acompanha os tipos de corpo do Box2D.

Essa não foi a primeira versão. Inicialmente, cada entidade criava seu corpo físico sem uma estratégia padronizada, o que gerava código repetido. A hierarquia atual surgiu de uma refatoração no fim de maio, quando se observou que a criação do corpo dependia apenas de dois fatores: o tipo (estático ou dinâmico) e um pequeno conjunto de

opções. Por isso, `DynamicObject` recebe uma estrutura de configuração com rotação fixa, modo *bullet* e escala de gravidade. O jogador e os bandidos usam rotação fixa, para não tombarem nas ladeiras; os projéteis desligam a gravidade e ativam o modo *bullet*, no qual o Box2D aplica detecção contínua de colisão e evita que um objeto rápido atravesse outro entre dois passos.

As vacas são objetos estáticos com a fixação marcada como sensor. Nas primeiras versões, elas eram corpos sólidos e bloqueavam o jogador. Como o objetivo é recolhê-las, e não contorná-las, a fixação foi convertida em sensor: o Box2D continua informando o contato, mas não produz resposta física de colisão. Os projéteis seguem o mesmo princípio. A cerca (`Fence`) foi implementada com pontos de vida e desenho que varia conforme o dano, mas ainda não foi posicionada no mapa.

4.4. Unidades e conversão

O Box2D é calibrado para objetos entre 0,1 e 10 metros, enquanto a tela trabalha em pixels. Adotou-se a escala de 30 pixels por metro: toda posição é convertida para metros ao criar um corpo e de volta para pixels no desenho, descontando a posição da câmera. A câmera acompanha o jogador apenas no eixo horizontal, mantendo-o no centro da tela. A gravidade do mundo foi definida em $10\,\mathrm{m/s^2}$.

4.5. Colisões

O Box2D informa os contatos por meio de uma classe de retorno (`b2ContactListener`). Esse aviso ocorre *durante* o passo de simulação, e a documentação proíbe criar ou destruir corpos nesse momento [Catto 2020]. Assim, se uma bala atinge um bandido e o bandido morre, ele não pode ser removido imediatamente.

A solução foi transformar o *listener* em uma fila. Cada fixação carrega, no campo de dados do usuário, um ponteiro para uma estrutura `DadosEntidade` com o tipo da entidade (jogador, bandido, vaca, terreno, bala do jogador, bala do bandido etc.) e um identificador numérico. O método `BeginContact` apenas armazena o par de estruturas envolvidas. Após o passo, o `GameWorld` esvazia a fila e decide o tratamento de cada par. Para reduzir o número de combinações, os dois elementos do par são ordenados pelo tipo antes da comparação; assim, “bala contra bandido” e “bandido contra bala” recaem no mesmo caso.

A remoção também é adiada. Objetos novos, como projéteis e bandidos, vão para uma lista de espera e só entram na coleção principal no fim do quadro. Objetos inativos são retirados com `std::remove_if`, depois de terem o corpo físico destruído e as estruturas `DadosEntidade` de suas fixações liberadas. A ordem de um quadro da partida é, portanto: passo da física, atualização dos objetos, ações do jogador, disparos dos bandidos, surgimento de novos bandidos, processamento das colisões, remoção dos objetos inativos, inclusão dos novos objetos e atualização da câmera.

4.6. Jogador, inimigos e projéteis

O movimento do jogador passou por várias versões. Na atual, cada toque em uma seta define um ponto-alvo dez metros à frente, e o corpo recebe velocidade horizontal constante até alcançá-lo ou até a tecla ser solta. Em ladeiras mais íngremes, o jogador podia

ficar parado contra o terreno tentando atingir um alvo inalcançável. Para tratar esse caso, o `Player` compara a posição de cada quadro com a do anterior e, se permanecer mais de três quadros sem se deslocar, cancela o movimento. O pulo aplica um impulso vertical e só é permitido quando o jogador está apoiado, o que é detectado pelo contato com o terreno.

Os bandidos são intencionalmente simples. Eles surgem a 60% da largura da tela, à esquerda ou à direita do jogador, com o lado sorteado, e não se deslocam. Se o jogador estiver a menos de 400 pixels, atiram em sua direção, aguardam 1,5 s de recarga, indicada por uma barra azul acima da cabeça, e voltam a contar o intervalo entre disparos. Cada bandido resiste a três tiros.

O jogador e os bandidos não criam projéteis diretamente. O `Player` apenas sinaliza que atirou naquele quadro, e o `GameWorld` consulta essa sinalização, reproduz o som e cria a bala. Dessa forma, o jogador não depende do mundo nem do sistema de áudio. Cada projétil tem velocidade constante e desaparece após três segundos ou ao tocar o terreno.

4.7. Sprites, animação e áudio

Diferentemente de guimaraes2025, que reuniram todos os quadros do personagem em uma única folha de sprites, em *Lampiões* cada imagem fica em um arquivo. A classe `SpriteCatalog` carrega todas elas uma única vez, na inicialização, e as armazena em um vetor de tamanho fixo indexado por uma enumeração; as demais classes recebem referências constantes para as sprites carregadas. A classe `Sprite` guarda a textura da SDL em um `std::unique_ptr` com um destrutor personalizado, que chama `SDL_DestroyTexture`. A cópia é proibida e o movimento é permitido, de modo que cada textura tem um único dono e é liberada automaticamente, segundo o idioma RAII [Stroustrup 2013]. A classe `Animation` guarda apenas ponteiros não proprietários para os quadros e troca de quadro a cada 0,15 s. A corrida do jogador tem três quadros, e o segundo também é usado como pose parada.

O áudio usa a SDL_mixer, com uma trilha de fundo em laço e efeitos para o pulo e o disparo. O texto dos menus e do painel de informações é gerado com a SDL_ttf. A Figura 3 mostra o menu principal.

5. Análise dos Resultados

5.1. Verificação funcional

A Tabela 1 resume os cenários verificados manualmente na versão final. Os defeitos indicados na coluna de observações são detalhados na Seção 5.3.

Com base nesses cenários, o jogo pode ser percorrido do início ao fim, com menu, pausa, condições de vitória e de derrota, áudio e um terreno contínuo percorrível nas duas direções. A verificação foi manual e não cobre combinações de eventos raras, como muitos projéteis simultâneos, nem diferentes configurações de hardware.

5.2. Decisões de arquitetura

Em relação à questão de pesquisa, três observações se destacaram.



Figura 3. Menu principal. A mesma classe `MenuScene` monta também as telas de pausa e de controles.

O uso de um motor de física deslocou a dificuldade. Não foi necessário implementar testes de interseção, resolução de sobreposição nem atrito, e o contato com as ladeiras do terreno foi tratado pelo Box2D. Em contrapartida, foi preciso compreender o modelo da biblioteca: unidades em metros, a diferença entre corpos e fixações, os sensores e, principalmente, a proibição de alterar o mundo durante o passo. A fila de contatos e as listas de inclusão e remoção adiadas decorreram dessa restrição e foram a parte da arquitetura que mais exigiu estudo da equipe.

A pilha de cenas facilitou a inclusão de novas telas e o gerenciamento da pausa. Ela foi implementada no fim do projeto, junto com o menu. Antes dela, o laço principal e a partida estavam acoplados. Após a mudança, a tela de controles exigiu apenas um novo caso na `MenuScene`, e a pausa não demandou código adicional na partida além do registro do pedido.

A hierarquia de classes precisou ser refeita. A primeira versão misturava responsabilidades, e a revisão com o professor orientador levou à criação de `PhysicsObject` e à padronização da criação dos corpos. A lição foi que a hierarquia de herança deve refletir o que os objetos realmente têm em comum, que neste caso era a relação com o mundo físico.

5.3. Limitações

Para explicitar as limitações da implementação, separamos os defeitos conhecidos, que fazem o jogo se comportar de forma diferente da pretendida, das funcionalidades ainda não implementadas.

Defeitos conhecidos:

- **D1 – Passo de física variável.** O mundo avança com o mesmo intervalo medido no quadro, contrariando a recomendação de fiedler2004. Como consequência, a altura do pulo e outros movimentos podem variar entre máquinas com taxas de quadros diferentes, e os resultados da Tabela 1 valem apenas para o ambiente em que foram obtidos.

Tabela 1. Cenários de verificação manual. *Atende*: comportamento esperado observado; *Parcial*: funciona, mas com defeito conhecido.

Cenário	Procedimento e resultado esperado	Situação	Observações
Movimentação	Setas em trechos planos e em ladeiras; o jogador se desloca até o ponto-alvo com animação de corrida	Atende	Em ladeiras íngremes, o movimento é cancelado pela detecção de travamento, como previsto
Pulo	Espaço com o jogador apoiado e no ar; impulso apenas quando apoiado	Parcial	Altura do pulo varia com a taxa de quadros (D1)
Colisão com o terreno	Percorrer as cinco zonas e a emenda do ciclo; o jogador não atravessa o terreno	Atende	–
Coleta de vacas	Tocar cada vaca; contador incrementa, vaca é removida e o jogador não é bloqueado	Atende	–
Disparo do jogador	Acertar um bandido três vezes; bandido removido e 100 pontos somados	Parcial	Outras balas do jogador em voo também são removidas (D2)
Disparo do bandido	Entrar no alcance; barra de recarga exibida e uma vida perdida ao ser atingido	Atende	–
Pausa	`Esc`, *Continuar* e *Voltar ao menu*; estado preservado ao continuar, partida destruída ao voltar	Atende	–
Vitória	Recolher as dez vacas; tela de vitória exibida	Atende	–
Derrota	Perder as cinco vidas; tela de derrota exibida	Atende	–

- **D2 – Remoção em massa de projéteis.** Os bandidos têm identificador, mas os projéteis não. Quando uma bala do jogador acerta um bandido, todas as balas do jogador em voo são removidas, e o mesmo ocorre quando qualquer bala toca o terreno. Esse defeito pode alterar o resultado de uma partida: um tiro que acertaria um segundo bandido deixa de existir, e um disparo que atinge o chão pode anular disparos válidos. Por isso, a dinâmica de combate observada no teste com visitantes não corresponde exatamente ao comportamento pretendido.

Fragilidades de implementação (não alteram o comportamento observado):

- **Memória gerenciada manualmente nos dados de colisão.** As estruturas `DadosEntidade` são alocadas com `new` e liberadas ao destruir o corpo, ao contrário do restante do código, que usa ponteiros inteligentes. Isso aumenta o risco de vazamentos em alterações futuras.
- **Busca por tipo a cada quadro.** Para localizar bandidos, vacas ou balas, o `GameWorld` percorre todos os objetos com `dynamic_cast`. O custo cresce linearmente com o número de objetos; com as poucas dezenas presentes em uma partida, não observamos queda de desempenho, mas não foram feitas medições, e a abordagem tende a se tornar inadequada com um número maior de entidades.
- **Caminhos relativos.** Sprites e sons são carregados com caminhos do tipo `../sprites`, o que exige executar o jogo a partir da pasta de compilação.

Funcionalidades não implementadas: movimento e comportamento dos bandidos, posicionamento das cercas no mapa e animações de pulo e de disparo.

São justamente detalhes desse tipo, como a frequência do passo de física e o ciclo de vida dos objetos ligados a ele, que uma *engine* completa costuma resolver de forma implícita.

5.4. Comparação com o trabalho anterior

A Tabela 2 compara *Lampiões* e *A Rusty Screw* [Renato Augusto Platz Guimarães Neto 2025] segundo critérios de arquitetura. A comparação é descritiva e não pretende indicar superioridade de uma abordagem, já que os jogos têm gêneros, escopos e equipes diferentes.

Tabela 2. Comparação entre as abordagens dos dois projetos.

Critério	***A Rusty Screw***	***Lampiões***
Detecção e resolução de colisões	Implementadas pela equipe (`CollisionEngine`)	Delegadas ao Box2D
Tratamento dos eventos de colisão	No próprio módulo de colisão, que separa pares estáticos e dinâmicos	Fila de contatos processada após o passo, com remoção adiada
Organização dos objetos	Separação entre objetos estáticos e dinâmicos	Mesma separação, mapeada para os tipos de corpo do Box2D
Recursos gráficos	Folha única de sprites por personagem	Uma imagem por arquivo, carregadas em um catálogo central
Principal dificuldade relatada	Escrever a física e obter boa sensação de combate	Adaptar a arquitetura às restrições do motor e ajustar o controle do personagem

As críticas recebidas por guimaraes2025 ao combate, pela ausência de *knockback*, e as dificuldades encontradas neste projeto indicam um mesmo tipo de desafio em pontos diferentes. Lá, a dificuldade estava em escrever a física; aqui, a física estava disponível, mas a sensação de controle ainda dependeu de ajustes manuais, como o movimento por ponto-alvo e a detecção de travamento. Nesta experiência, o motor de física não resolveu sozinho o *game feel*.

5.5. Retorno exploratório de jogadores

O jogo foi disponibilizado durante a mostra de cursos do IFPR – Campus Paranavaí, para visitantes interessados no curso de Engenharia de Software. Os visitantes jogaram livremente e comentaram a experiência com a equipe logo em seguida. Os comentários foram, em geral, favoráveis, e o elemento mais mencionado foi o som do disparo.

Por não haver questionário estruturado nem registro do número de participantes, esses comentários representam apenas a impressão inicial da equipe e não permitem conclusões sobre usabilidade ou diversão. A frequência com que o som do disparo foi citado sugere a hipótese de que o retorno sonoro contribui para a percepção de impacto do tiro, mas essa hipótese precisa ser verificada em uma avaliação estruturada.

6. Conclusões e Trabalhos Futuros

Este artigo relatou o desenvolvimento de *Lampiões*, um jogo 2D de faroeste feito em C++ com SDL2 e Box2D, sem uma *engine* completa de jogos, mas com um motor de física externo. Nesta experiência, a combinação se mostrou um meio-termo adequado para fins de

aprendizagem: a parte matemática da física ficou a cargo de uma biblioteca consolidada, enquanto a equipe precisou decidir e implementar o laço principal, a gestão de telas, o ciclo de vida dos objetos e a ligação entre o mundo físico e a lógica do jogo. Foram essas decisões, e não a física, que ocuparam a maior parte do tempo de desenvolvimento.

Para outros estudantes que sigam um caminho semelhante, as principais lições deste relato são: tratar as restrições do motor de física (como a proibição de alterar o mundo durante o passo) como requisitos de arquitetura desde o início; adotar um passo de física fixo; e estruturar a hierarquia de classes a partir do que os objetos realmente compartilham.

Os resultados apresentados têm alcance limitado: a verificação foi manual e em um único ambiente, os defeitos D1 e D2 afetam o comportamento das partidas, e o retorno dos jogadores foi informal. Como continuação, pretendemos:

- adotar um passo fixo para a física, com acumulador de tempo, conforme fiedler2004;
- atribuir identificadores aos projéteis para corrigir a remoção em massa e substituir a alocação manual dos dados de colisão por uma estrutura com posse definida;
- implementar movimento e comportamento para os bandidos, posicionar as cercas no mapa e acrescentar animações de pulo e de disparo;
- realizar uma avaliação estruturada com jogadores, com questionário, nos moldes de guimaraes2025, para medir a dificuldade e a sensação de controle e verificar a hipótese sobre o papel do áudio;
- preparar a distribuição para Windows, já iniciada em um ramo separado do repositório.

Referências

Capcom (1985). Gun.smoke. Jogo de fliperama.

Catto, E. (2020). *Box2D v2.4.1 Documentation*. Disponível em: `https://box2d.org/documentation/`. Acesso em: 16 out. 2026.

Konami (1991). Sunset riders. Jogo de fliperama.

Millington, I. (2010). *Game Physics Engine Development*. Morgan Kaufmann, Burlington, MA, 2 edition.

Renato Augusto Platz Guimarães Neto, E. H. M. d. C. (2025). Explorando c++ e sdl na criação de um jogo 2d do gˆenero metroidvania. In *SETIF*, Paranavaí, PR. IFPR.

SDL Community (2026). *Simple DirectMedia Layer 2.0 Documentation Wiki*. Disponível em: `https://wiki.libsdl.org/SDL2/`. Acesso em: 16 out. 2026.

Stroustrup, B. (2013). *The C++ Programming Language*. Addison-Wesley, Upper Saddle River, NJ, 4 edition.