INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ
CAMPUS TERESINA CENTRAL
TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GABRIEL DE ALMEIDA LIMA GOMES

TÉCNICAS DE PROGRAMAÇÃO PARA OTIMIZAÇÃO DE JOGOS DESKTOP COM A GODOT ENGINE

TERESINA
2025

GABRIEL DE ALMEIDA LIMA GOMES

TÉCNICAS DE PROGRAMAÇÃO PARA OTIMIZAÇÃO DE JOGOS DESKTOP COM A GODOT ENGINE

Trabalho de Conclusão de Curso (monografia) apresentado como exigência parcial para obtenção do diploma do Curso de Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central.

Orientadora: Profa. Dra. Elanne Cristina Oliveira dos Santos

TERESINA

2025

Dados Internacionais de Catalogação na Publicação (CIP) de acordo com ISBD

Gomes , Gabriel de Almeida Lima

G633t Técnicas de programação para otimização de jogos desktop com a Godot Engine / Gabriel de Almeida Lima Gomes . - 2025.
62 f.: il. color.

Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) - Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central, 2025.
Orientadora : Profa Dra. Elanne Cristina Oliveira dos Santos .

1. Godot Engine. 2. Jogos. 3. Otimização. I.Título.

CDD - 004.21

Elaborado por Maria Rosismar Farias CRB 3/631

GABRIEL DE ALMEIDA LIMA GOMES

TÉCNICAS DE PROGRAMAÇÃO PARA OTIMIZAÇÃO DE JOGOS DESKTOP COM A GODOT ENGINE

Trabalho de Conclusão de Curso (monografia) apresentado como exigência parcial para obtenção do diploma do Curso de Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central.

Aprovada em 28/07/2025.

BANCA EXAMINADORA:

Profa. Dra. Elanne Cristina(Orientadora)
Instituto Federal do Piauí (IFPI)

Prof. M^e^. Duany Dreyton Bezerra Sousa
Instituto Federal do Piauí (IFPI)

Prof. Dr. Ricardo Martins Ramos
Instituto Federal do Piauí (IFPI)

Dedico este trabalho a todos que acreditaram que ele sairia.

AGRADECIMENTOS

Agradeço a todos os que me deram apoio até aqui, e auxiliaram na criação deste trabalho.

“Qualquer tecnologia suficientemente avançada é indistinguível da magia.”
Arthur C. Clarke

RESUMO

O presente estudo visa investigar as principais técnicas de otimização para jogos desenvolvidos utilizando o motor de jogos Godot Engine, com ênfase na plataforma desktop. Nesta pesquisa, serão explorados conceitos essenciais sobre otimização de jogos, tais como técnicas que visam economizar recursos computacionais com o objetivo de atingir melhor desempenho. Para este trabalho, foram apresentadas 3 (três) estratégias de otimização (Object Pooling, Spatial Partition e Dirty Flag) que usam métodos de programação com o objetivo de melhorar a performance e fluidez de jogos, usando uma abordagem que priorize uma execução de código rápida e econômica. Após a apresentação das técnicas, cada uma delas foi implementada no jogo Platformer 2D, um jogo gratuito de código aberto disponível no motor de jogos Godot Engine. Logo em seguida foram realizadas comparações e análises dos resultados obtidos antes e depois de aplicar as otimizações propostas.

Palavras-chaves: Godot Engine; jogos; otimização.

ABSTRACT

This study aims to investigate the main optimization techniques for games developed using the Godot Engine, with a focus on the desktop platform. The research explores essential concepts of game optimization, such as techniques aimed at saving computational resources to achieve better performance. For this work, three optimization strategies (Object Pooling, Spatial Partition, and Dirty Flag) were presented, which utilize programming methods to enhance game performance and smoothness, prioritizing fast and efficient code execution. After introducing the techniques, each was implemented in the Platformer 2D game, a free, open-source game available on the Godot Engine. Subsequently, comparisons and analyses of the results obtained before and after applying the proposed optimizations were conducted.

Keywords: Godot Engine; games; optimization.

LISTA DE ILUSTRAÇÕES

Figura 1 – Estrutura de árvores dentro da Godot Engine. 16
Figura 2 – Ilustração da organização em árvores da Godot Engine. 17
Figura 3 – Captura de tela da aba ”Visual Profiler” da Godot Engine. 18
Figura 4 – Ilustração do fluxo de funcionamento do Object Pool. 22
Figura 5 – Captura de tela de um dos confrontos do jogo Terraria. 24
Figura 6 – Ilustração de caso de uso do Spatial Partition. 25
Figura 7 – Captura de tela do cenário do jogo Civilization VI. 26
Figura 8 – Ilustração do exemplo onde o Dirty Flag é aplicável. 28
Figura 9 – Comparação entre o número de chamadas das funções de transformação de posição, antes e depois da aplicação do Dirty Flag. 28
Figura 10 – Captura de tela do cenário do jogo Sea of Thieves. 29
Figura 11 – Captura de tela do cenário do jogo Platformer2d. 34
Figura 12 - Código da função de disparo de projéteis do Platformer2d. 35
Figura 13 - Código de criação do pool de objetos. 36
Figura 14 - Código de disparo modificado para implementação do Object Pooling. 37
Figura 15 - Código de reciclamento de projéteis. 38
Figura 16 - Código de validação de objetos do pool. 39
Figura 17 - Código das funções de surgimento e destruição de inimigos, respectivamente, implementadas no Platformer2d. 40
Figura 18 - Código da função “ _ready”, responsável pela inicialização do EnemySpawner. 41
Figura 19 - Código de criação do pool de inimigos. 41
Figura 20 - Código de reutilização dos inimigos do pool. 42
Figura 21 - Código de reciclamento dos inimigos. 42
Figura 22 – Captura de tela do consumo de memória do Platformer2d antes da utilização do pooling de projéteis. 43
Figura 23 - Captura de tela do consumo de memória do Platformer2d após a utilização do pooling de projéteis. 44
Figura 24 – Tempo de execução da função “shoot” antes da implementação do pooling. 44
Figura 25 - Tempo de execução da função “shoot” após a implementação do pooling. 45
Figura 26 – Captura de tela do consumo de memória do Platformer2d antes da utilização do pooling de inimigos. 45
Figura 27 – Captura de tela do consumo de memória do Platformer2d depois da utilização do pooling de inimigos. 46

Figura 28 – Tempo de execução da função “_on_spawn_timer_timeout” antes da implementação do pooling. 46
Figura 29 – Tempo de execução da função “_on_spawn_timer_timeout” depois da implementação do pooling. 46
Figura 30 - Código do sistema de detecção de colisão de inimigos sem Spatial Partitioning 48
Figura 31 - Código da estrutura de grid do Spatial Partitioning 49
Figura 32 - Representação visual do Spatial Grid com os inimigos do Platformer2d. 50
Figura 33 - Código de manutenção do grid de Spatial Partition 51
Figura 34 - Código de inicialização e registro de performance da comparação com Spatial Partition 52
Figura 35 – Captura de tela do consumo de tempo e memória do Platformer2d antes da utilização do Spatial Partition. 53
Figura 36 - Captura de tela do consumo de tempo e memória do Platformer2d após a utilização do Spatial Partition. 53
Figura 37 - Código da lógica de controle das animações do jogador no Platformer2d 55
Figura 38 - Código de inicialização das variáveis de controle do Dirty Flag no Platformer2d 56
Figura 39 - Código de implementação da variável changed de controle do Dirty Flag no Platformer2d 56
Figura 40 - Código de implementação do Dirty Flag no controle de animações de jogador no Platformer2d 57
Figura 41 - Código de inicialização e registro de performance da comparação com Dirty Flag 58
Figura 42 – Captura de tela de consumo de tempo e memória do Platformer2d antes da utilização do Dirty Flag. 59
Figura 43 - Captura de tela do consumo de tempo e memória do Platformer2d após a utilização do Dirty Flag. 59

SUMÁRIO

1 INTRODUÇÃO 12
2. FUNDAMENTAÇÃO TEÓRICA 14
2.1 DESENVOLVIMENTO DE JOGOS 14
2.1.1 GameMaker Studio 14
2.1.2 Godot Engine 15
2.2 DESENVOLVIMENTO DE JOGOS COM A GODOT ENGINE 15
2.3 OTIMIZAÇÃO 18
2.3.1 Recursos Computacionais 19
2.3.1.1 Tempo de Execução 19
2.3.1.2 Memória 20
2.3.2 Percepção da Otimização 20
2.4 TÉCNICAS DE OTIMIZAÇÃO 21
2.4.1 Object Pooling 22
2.4.2 Spatial Partition 24
2.4.3 Dirty Flag 26
3 METODOLOGIA 31
3.1 CARACTERIZAÇÃO DA PESQUISA 31
3.2 ETAPAS DA PESQUISA 31
4 O JOGO PLATFORMER 2D 34
4.1 APLICAÇÃO DA TÉCNICA OBJECT POOLING 35
4.1.1 Resultados comparativos do Object Pooling 42
4.1.2 Análise dos resultados do Object Pooling 47
4.2 APLICAÇÃO DA TÉCNICA SPATIAL PARTITION 48
4.2.1 Análise comparativa do Spatial Partition 53
4.2.2 Análise dos resultados do Spatial Partition 54
4.3 APLICAÇÃO DA TÉCNICA DIRTY FLAG 54
4.3.1 Análise comparativa do Dirty Flag 58
4.3.2 Análise dos resultados do Dirty Flag 59
5 CONCLUSÕES 61
REFERÊNCIAS 62

1. INTRODUÇÃO

O desenvolvimento de jogos tem se tornado cada vez mais complexo, impulsionado por um crescente foco na criação de experiências imersivas que exigem alta fidelidade gráfica, bem como mecânicas de jogo cada vez mais ambiciosas. Essa evolução resulta em jogos que demandam poder computacional significativo, o que exige constante adaptação das ferramentas e tecnologias utilizadas para manter o desempenho otimizado. No entanto, à medida que os jogos evoluem, torna-se cada vez mais importante manter práticas de otimização que minimizem a carga computacional necessária para garantir uma execução fluída e consistente, independentemente das limitações do hardware dos dispositivos utilizados pelos jogadores.

Nesse contexto, o desempenho surge como um dos maiores desafios enfrentados pelos desenvolvedores de jogos, uma vez que limitações impostas por componentes de hardware, como GPU (Unidade de Processamento Gráfico) e a CPU (Unidade Central de Processamento), podem causar quedas significativas nas taxas de quadros, afetando negativamente a experiência do usuário e a jogabilidade. (HEINEMAN, 2008).

Considerando esses gargalos, a otimização de jogos não apenas resulta em uma experiência mais satisfatória para os usuários, mas também aumenta a acessibilidade do produto, de dispositivos, permitindo que pessoas com dispositivos menos potentes consigam jogá-lo. Essa melhora no alcance é favorável tanto para os desenvolvedores, ao aumentar o número de jogadores, sua retenção e avaliação do produto, quanto para os consumidores, ao permitir que eles aproveitem o jogo em uma variedade de dispositivos.

Esta pesquisa tem como objetivo principal estudar o uso de técnicas de programação para otimizar o funcionamento de jogos de computadores, bem como analisar a aplicação dessas técnicas em jogos de computadores já existentes. Assim, este trabalho visa explorar três das técnicas mais conhecidas de otimização de código para jogos, os problemas de otimização específicos que cada uma dessas técnicas se propõe a resolver, e desta forma realizar análises comparativas de performance antes e depois do uso delas. Para isso, separamos os seguintes objetivos específicos:

-Identificar game engines para construção de jogos, bem como jogos de computadores de código aberto disponíveis para atualização.

-Analisar as técnicas de programação especificadas para otimização de jogos e em qual contexto sua implementação pode melhorar o desempenho do jogo sem

prejudicar a qualidade visual ou a jogabilidade.

-Implementar e avaliar o uso das técnicas de otimização exploradas em um jogo de código aberto.

Os resultados desta pesquisa buscam demonstrar, por meio da análise dessas técnicas, soluções práticas que possibilitem a melhoria no desempenho de jogos, permitindo que possam ser executados de forma mais fluída e consistente em dispositivos desktop. Entretanto, busca-se também promover a responsabilidade na implementação destas técnicas, visto que cada uma delas é indicada como solução para situações distintas.

Para isto, este trabalho é dividido em 5 (cinco) capítulos. O capítulo 1 apresenta a introdução, onde é exposta a motivação para a realização do trabalho, bem como o objetivo geral e objetivos específicos. O capítulo 2 trata da fundamentação teórica, nele serão contextualizados os conceitos de game engines, conhecimentos essenciais sobre otimização e sua importância para o desempenho de um jogo, bem como serão exploradas técnicas populares de otimização de código para jogos. No capítulo 3, são apresentados os procedimentos metodológicos utilizados para realização da pesquisa. No capítulo 4 será apresentado o jogo Platformer 2D e os métodos de otimização aplicados a ele, bem como os resultados obtidos. No capítulo 5, será apresentada a conclusão resultante da análise dos dados obtidos no experimento.

2. FUNDAMENTAÇÃO TEÓRICA

2.1 DESENVOLVIMENTO DE JOGOS

O desenvolvimento de jogos é uma área ampla que envolve a integração de diversas disciplinas, como programação, design gráfico, design de som e narrativa interativa, em um único produto coeso. Para auxiliar esse processo de criação, existem diversas game engines que oferecem um ambiente para facilitar a implementação desses elementos.

Game engines são plataformas que oferecem, de forma integrada, uma série de ferramentas e funcionalidades que visam facilitar o desenvolvimento de jogos. (GAME DEVELOPER, 2024). Essas plataformas oferecem suporte para a criação e integração de elementos como gráficos, áudio, física, inteligência artificial e scripts, permitindo que os desenvolvedores foquem mais no design e na experiência do jogador.

Cada uma das várias engines disponíveis no mercado possui sua própria maneira de abstrair as muitas partes do desenvolvimento de jogos. Enquanto algumas são projetadas para atender desenvolvedores iniciantes, oferecendo interfaces simplificadas, como a GameMaker Studio, outras focam em atender demandas de estúdios profissionais, fornecendo ferramentas avançadas e alto grau de personalização, como a Unreal Engine. Existe também uma categoria intermediária, representada pela Godot Engine, que combina acessibilidade com uma interface intuitiva e recursos robustos, oferecendo uma plataforma que atende tanto iniciantes quanto desenvolvedores mais experientes.

2.1.1 GameMaker Studio

O GameMaker Studio é amplamente reconhecido como uma engine acessível para desenvolvedores iniciantes. Sua interface simplificada e o uso de uma linguagem de programação própria, chamada GML (GameMaker Language), permitem que mesmo indivíduos com pouca ou nenhuma experiência em programação possam criar jogos rapidamente. Além disso, a plataforma oferece diversas funcionalidades pré-configuradas, como sistemas de colisão, áudio e animações, reduzindo a complexidade do desenvolvimento. Um exemplo notável de jogo criado com o GameMaker Studio é "Undertale", desenvolvido por Toby Fox, que alcançou sucesso mundial devido à sua narrativa envolvente e mecânicas criativas.

A Unreal Engine é uma das game engines mais robustas e amplamente utilizadas

no mercado profissional. Desenvolvida pela Epic Games, ela oferece uma ampla gama de ferramentas avançadas para criação de gráficos realistas, simulação física, inteligência artificial e muito mais. Sua capacidade de renderização de gráficos de alta qualidade é uma das razões pelas quais é a escolha principal de muitos estúdios com alto orçamento. Um exemplo significativo de jogo desenvolvido com a Unreal Engine é "Fortnite", também da Epic Games, que se destaca por sua jogabilidade fluida e visual impressionante.

2.1.2 Godot Engine

A Godot Engine é uma game engine gratuita desenvolvida inicialmente por Juan Linietsky e Ariel Manzur em parceria com a empresa OKAM Studio. A Godot teve sua primeira versão estável lançada em 2014 e desde então tem recebido atualizações regulares, incorporando novos paradigmas de desenvolvimento, suporte a diversas linguagens de programação e compatibilidade com múltiplas plataformas.

O principal objetivo da Godot Engine é fornecer um ambiente de desenvolvimento integrado, eliminando a necessidade de ferramentas externas além daquelas usadas para a criação de conteúdo. Além disso, a plataforma é open source, ou seja, a Godot permite que qualquer pessoa acesse, estude, modifique e redistribua seu código-fonte livremente, o que fomenta uma comunidade ativa que colabora para um aperfeiçoamento contínuo da engine.

Entre suas funcionalidades, destaca-se a compatibilidade multiplataforma, abrangendo dispositivos móveis, desktops, web, consoles e até mesmo plataformas de realidade virtual. A engine também oferece suporte a quatro linguagens de programação: GDScript, sua linguagem oficial e de fácil aprendizado; C#, para desenvolvedores familiarizados com o ecossistema .NET; VisualScript, uma opção visual baseada em nós; e GDNative, que permite integração com bibliotecas externas em linguagens como C e C++. Por esses motivos, a Godot Engine, em sua versão 4.4.1, foi escolhida como a plataforma ideal para este estudo, utilizando o GDScript, sua linguagem padrão, como a principal linguagem de programação.

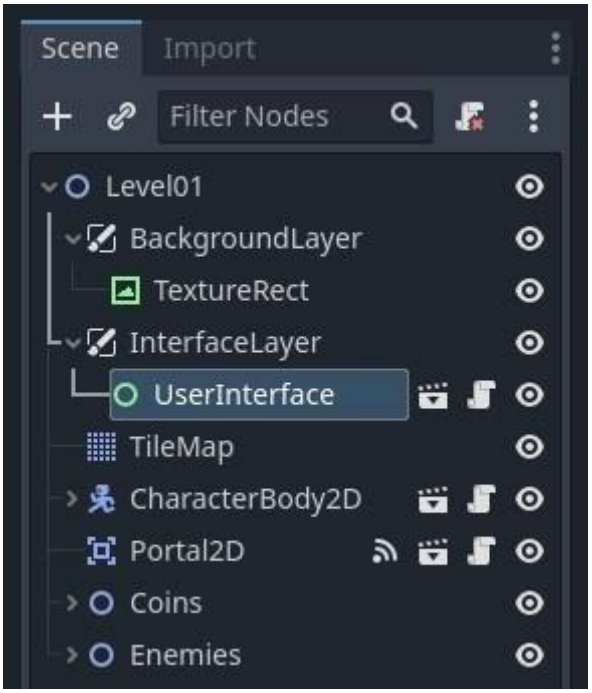
2.2 DESENVOLVIMENTO DE JOGOS COM A GODOT ENGINE

Como uma game engine, a Godot Engine possui sua própria maneira de abstrair o desenvolvimento de um jogo. Na plataforma, um jogo é tratado como uma estrutura formada por diversos nós, que se agrupam em árvores. Em Godot, os nós são objetos que representam os diversos elementos de um jogo, sejam estes 2D, 3D, ou parte de uma interface visual. (GODOT, 2014)

Esses nós podem ser organizados em árvores, dando origem a estruturas chamadas cenas. As cenas são unidades reutilizáveis que podem representar desde um único personagem até um cenário inteiro, com diversos nós atrelados a si. O conjunto de várias cenas que formam um jogo é chamado de árvore de cenas, e ela encapsula todos os objetos presentes no jogo. Em Godot, uma árvore de cenas é uma estrutura hierárquica, onde a cena principal serve como o nó raiz, e de onde outras cenas podem ramificar. Cada cena pode ter seus próprios nós, como personagens, itens e elementos do cenário, criando uma relação de dependência e interação entre os diversos elementos do jogo. (GODOT, 2014)

Em relação à organização da árvore de cenas, é possível visualizá-la como uma estrutura única, com a cena principal no topo, que pode se ramificar para outras cenas, conforme a necessidade do jogo. Por exemplo, uma árvore de cenas pode ser composta pela cena principal, que carrega cenas específicas para menus, fases ou áreas do jogo. Abaixo da cena principal, outras cenas podem ser carregadas, como uma cena para o personagem, uma para o inimigo, uma para a interface, e assim por diante, conforme mostra a Figura 1.

Figura 1 - Estrutura de árvores dentro da Godot Engine

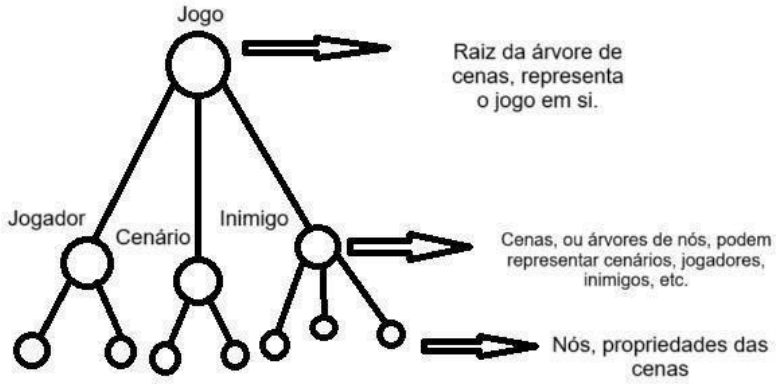


Fonte: GODOT (2014).

Cada uma dessas cenas é uma árvore de nós que, por sua vez, podem ser manipulados conforme o fluxo do jogo. Dessa forma, entende-se resumidamente que uma árvore de cenas é uma estrutura individual na organização de um jogo em Godot, e essa estrutura contém múltiplas outras, as árvores de nós, com propriedades diversas, conforme mostra o exemplo na Figura

2 abaixo.

Figura 2 - Ilustração da organização em árvores da Godot Engine.

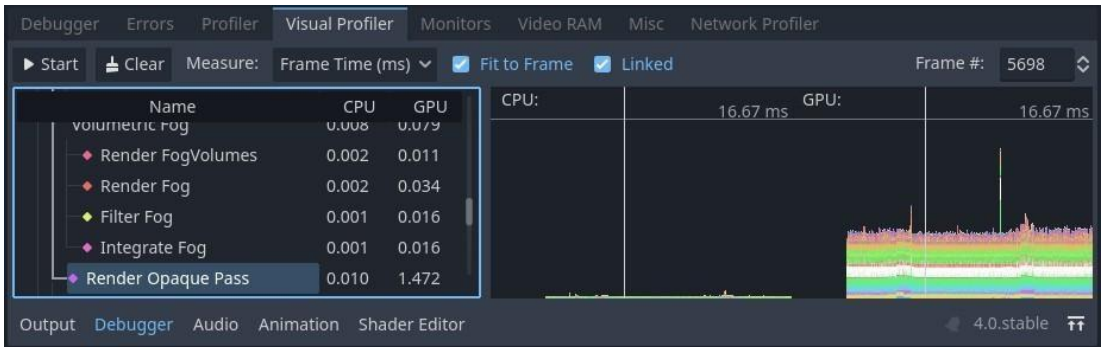


Fonte: Produzido pelo autor.

Em Godot, a comunicação entre diferentes nós pode ser facilitada pelo uso de sinais. Sinais são funções responsáveis por emitir uma espécie de aviso a outros nós quando um determinado evento ocorre. Por exemplo, utilizando sinais, é possível que quando o personagem de um jogador corra, seja enviado um sinal para os nós responsáveis pelos inimigos, tornando-os agressivos e ativando sua inteligência artificial para perseguir o jogador. Um outro exemplo seria quando o personagem coleta um item, o que pode emitir um sinal para um nó de interface de usuário, atualizando o inventário ou mostrando uma mensagem de sucesso na tela. Outro exemplo seria quando um inimigo colide com o jogador, emitindo um sinal para o nó de saúde, que reduziria os pontos de vida do jogador.

A Godot Engine também disponibiliza uma ferramenta de depuração chamada debugger (ou “depurador”). A ferramenta pode ser encontrada em uma aba no aplicativo e permite visualizar diversos índices de desempenho e uso de recursos relacionados ao jogo em desenvolvimento. Informações relevantes, como o tempo de execução de scripts, consumo de memória, uso de CPU e GPU, e processamento de física, são apresentadas em tempo real, permitindo aos desenvolvedores identificar gargalos de performance e otimizá-los de forma precisa. A Figura 3 abaixo apresenta a janela do depurador (“Visual Profiler”) do aplicativo Godot Engine.

Figura 3 - Captura de tela da aba “Visual Profiler” da Godot Engine.



Fonte: GODOT (2014).

Essa ferramenta é especialmente útil para medir o impacto de diferentes elementos do jogo, como animações, colisões e efeitos visuais, no desempenho geral. Por exemplo, ao monitorar a Memória Estática (memória alocada para os objetos no jogo), é possível determinar de forma dinâmica quais elementos estão consumindo mais recursos computacionais.

O debugger será essencial em tópicos posteriores deste trabalho, pois a partir da análise comparativa decorrente dos dados oferecidos por ele, podemos confirmar os impactos positivos das técnicas de otimização aplicadas ao código fonte dos jogos.

2.3 OTIMIZAÇÃO

A definição de otimização pode ser dada como a arte e a ciência de alocar recursos escassos para obter o melhor resultado possível. Para cada problemática que envolve o uso desses recursos finitos, torna-se imperativo formular um modelo de otimização que ataque os gargalos de performance do produto, serviço ou iniciativa de modo a maximizar sua eficiência. (CHINNEK, 2010, p.1).

De modo geral, a otimização envolve dois conceitos relacionados: Maximização e minimização. O estudo “Introduction to Mathematical Optimization” da Universidade de Stanford traz um exemplo simples e prático:

> Otimização vem da mesma raiz que "ótimo", que significa o melhor. Quando você otimiza algo, você está "tornando-o o melhor". Mas "melhor" pode variar. Se você é um jogador de futebol americano, pode querer maximizar suas jardas corridas e minimizar seus tropeços.

Dessa forma, os processos de otimização envolvem minimizar aspectos negativos de uma atividade, enquanto se maximizam os aspectos positivos, de

modo a criar um balanceamento dos recursos envolvidos que mais beneficiem os resultados da tarefa proposta.

Trazendo esse conceito para o mundo de jogos eletrônicos, cada jogo tem um número limitado de recursos computacionais que lhe são disponíveis e é papel do desenvolvedor utilizá-los para criar a melhor experiência possível para o usuário final. (PREISZ, 2011, p.1)

O gerenciamento desses recursos computacionais pode ser feito por meio de diversas técnicas de otimização. De modo geral, essas práticas se dividem entre duas categorias: Técnicas aplicadas aos objetos e cenários do jogo, sendo suas implementações configuradas na própria engine, não necessitando escrever código para se aproveitar delas; Técnicas de programação, onde o desenvolvedor é responsável por implementá-las ao código-fonte do jogo com base em um padrão desejado. Embora este trabalho foque exclusivamente em técnicas aplicadas ao código, é importante notar que essas duas categorias de otimização costumam ser implementadas em conjunto para maximizar o ganho de performance.

2.3.1 Recursos Computacionais

Segundo Preisz (2011), recursos computacionais a serem utilizados por um jogo podem ser enxergados como as duas dimensões nas quais os dispositivos operam: tempo e espaço. O objetivo de um desenvolvedor que se propõe a maximizar a performance de um jogo é utilizar o mínimo desses dois recursos para oferecer a experiência desejada ao usuário, atentando-se a possíveis limitações de hardware. Em termos mais específicos, podemos definir as medidas de tempo e espaço de um jogo em tempo de execução e memória, respectivamente.

2.3.1.1 Tempo de Execução

Para entender o conceito de tempo de execução em um jogo, precisamos compreender como funcionam instruções em termos de computação. Quando um dispositivo recebe uma instrução, por exemplo, a de multiplicar dois números, essa ação é processada por sua CPU. Por sua vez, a CPU demora uma quantidade de tempo para realizar os cálculos e retornar o resultado dessa operação. Essa dinâmica entre CPU e comandos é denominada uma instrução. (PREISZ, 2011, p. 153, 154). Dessa forma, caso instruções desnecessárias sejam enviadas para processamento, levará mais tempo para que um resultado desejado seja obtido.

Embora esteja em um nível de abstração mais alto, o conceito de tempo de execução em um jogo é intimamente ligado ao de instruções. Quando uma função é chamada para que algo aconteça em um jogo, as diversas instruções atreladas a ela são enviadas para processamento. Cada linha de código que se traduza em

mais demanda de processamento do que o necessário está diminuindo a performance geral da aplicação.

A intensidade da queda de performance que esse uso indevido de processamento causa pode variar entre imperceptível e claramente visível para executar. É comum que isso ocorra devido a descuidos de programação no código fonte de um jogo, acarretando experiências negativas como demora nas trocas entre cenários do jogo e nos comportamentos dos personagens. Desse modo, é importante otimizar o código de um jogo para que não haja funções que gerem um conjunto de instruções mais lentas do que o necessário para a CPU.

2.3.1.2 Memória

A memória refere-se ao espaço que um dispositivo disponibiliza para que dados sejam gravados e carregados. Segundo Preisz (2011), a memória, assim como o tempo de execução, representa um dos principais recursos computacionais a serem otimizados, sendo frequentemente limitada pela capacidade do hardware do jogador. Quando um jogo é executado, ele utiliza a memória para carregar texturas, sons, modelos, estados dos personagens e outras informações que precisam ser acessadas rapidamente.

A má gestão da memória pode levar a problemas como quedas de desempenho, travamentos ou até falhas no carregamento de elementos visuais e sonoros do jogo. Um dos problemas mais notáveis que podem levar a essas consequências é uma falha em liberar a memória de recursos que não estão sendo utilizados no momento, algo que em casos mais drásticos pode levar até mesmo a uma sobrecarga do sistema.

Game engines modernas como a Godot Engine oferecem suporte para gerenciamento automático de memória, incluindo a funcionalidade de “coleta de lixo”, que ajuda a liberar recursos não utilizados. No entanto, essa ferramenta não soluciona todos os problemas relacionados à alocação de memória, e existem técnicas de otimização que ajudam a preservar melhor esse recurso quando aplicadas ao código.

2.3.2 Percepção da Otimização

A forma mais comum pela qual um jogador percebe o desempenho de um jogo é por meio da taxa de quadros. A taxa de quadros de um jogo refere-se à frequência em que quadros (imagens individuais que formam a exibição do jogo) são exibidos em sequência. Quanto maior a taxa de quadros, mais convincente é para o jogador a ilusão de que tudo está se movendo de maneira fluída. (PREISZ, 2011, p. 32).

Segundo Preisz (2011, p.33), mais importante que manter uma taxa de quadros muito alta, é mantê-la constante. A percepção humana é capaz de se ajustar a uma taxa de quadros baixa, porém constante, mas oscilações nessa taxa são facilmente perceptíveis. Seria mais desejável, por exemplo, uma taxa de quadros de 30Hz que se mantém constante, do que uma que oscila frequentemente entre 60Hz e 30Hz.a Dessa forma, a taxa de quadros de um jogo é um fator crucial para a experiência de um consumidor, e mantê-la não só alta como constante é uma das tarefas que um desenvolvedor precisa se empenhar para exercer.

Entretanto, existe outra forma importante de avaliar a performance: a demora na resposta aos comandos do usuário, referida simplesmente como 'atraso' ao longo deste trabalho. Segundo Preisz (2011, p.32), muitos fatores influenciam nossa percepção desse atraso. Em algumas situações, o jogador pode tolerar um atraso maior caso isso esteja dentro de suas expectativas. No entanto, em situações em que o usuário espera uma resposta rápida, o atraso pode resultar em uma experiência negativa.

Em termos gerais, uma queda na taxa de quadros é normalmente um resultado de um problema relacionado à memória, já o atraso, relacionado a um problema de tempo de execução. Ambos devem ser solucionados na medida do possível para garantir a melhor experiência ao consumidor. A otimização possui um papel extremamente importante nisso, pois gerenciar os recursos computacionais de um dispositivo é fundamental tanto para gerar quadros de uma maneira mais fluída e constante, quanto para diminuir atrasos não esperados.

2.4 TÉCNICAS DE OTIMIZAÇÃO

Existem diversas maneiras de otimizar o código de um jogo, cada uma melhorando a economia de recursos em níveis diferentes. Este tópico tratará de algumas das técnicas mais conhecidas e eficientes para otimização de código, detalhando como elas devem ser aplicadas, em quais situações, e quais recursos as suas aplicações ajudam a preservar.

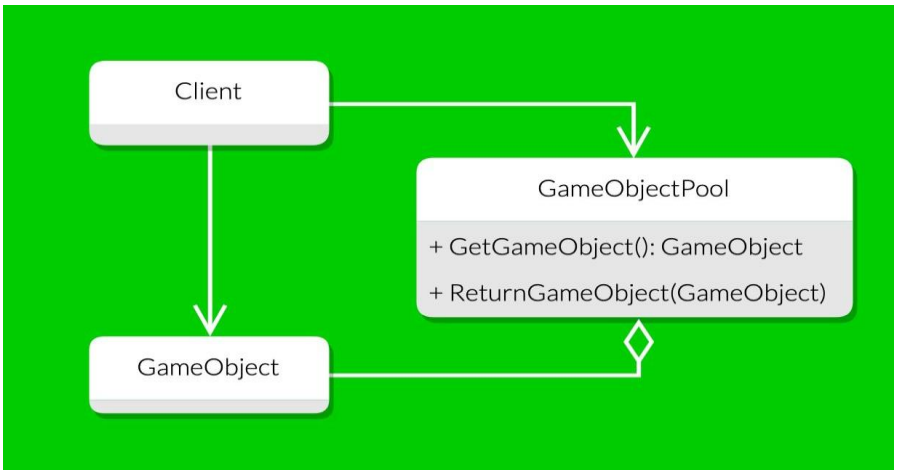
Dentre as diversas abordagens de otimização disponíveis, este trabalho destacou três técnicas específicas: Object Pooling, Dirty Flag e Spatial Partition. Essas técnicas foram selecionadas por sua capacidade de abordar de forma eficiente os principais desafios de otimização abordados neste estudo, especificamente o gerenciamento eficiente de memória e a redução do tempo de execução.

2.4.1 Object Pooling

A técnica de Object Pooling (ou "piscina de objetos") é amplamente utilizada em jogos para melhorar o desempenho e a utilização de memória. Essa prática consiste em reutilizar objetos previamente alocados, evitando o custo de criar e destruir instâncias repetidamente durante a execução do jogo (NYSTROM, 2014).

A implementação do Object Pooling envolve a definição de uma classe ou variável de pool que mantém uma coleção de objetos reutilizáveis. Quando o pool é inicializado, ele cria todos os objetos de uma vez e os inicializa em um estado de "não em uso". Quando um objeto é requisitado, o pool encontra um objeto disponível, inicializa-o e o marca como "em uso". Após o uso, o objeto é retornado ao pool e novamente marcado como "não em uso". Esse processo permite a criação e destruição de objetos de forma eficiente, sem a necessidade de alocar e desalocar memória constantemente (NYSTROM, 2014). Abaixo, a Figura 4 mostra o fluxo de funcionamento de um Object Pool.

Figura 4 - Ilustração do fluxo de funcionamento do Object Pool



Fonte: DEVMAKING (2021).

O Object Pooling pode ser usado para uma variedade de objetos, desde projéteis, personagens, e elementos do cenário. No entanto, é importante garantir que o tamanho do pool seja ajustado conforme as necessidades do jogo. Se o pool for muito pequeno, pode ocorrer a falta de objetos disponíveis, prejudicando a performance do jogo. Por outro lado, se for muito grande, pode haver desperdício de memória, já que os objetos não utilizados ficam ocupando espaço sem necessidade. Por isso, é fundamental que o tamanho do pool seja ajustado de acordo com os requisitos específicos de cada cenário dentro do jogo. (NYSTROM, 2014).

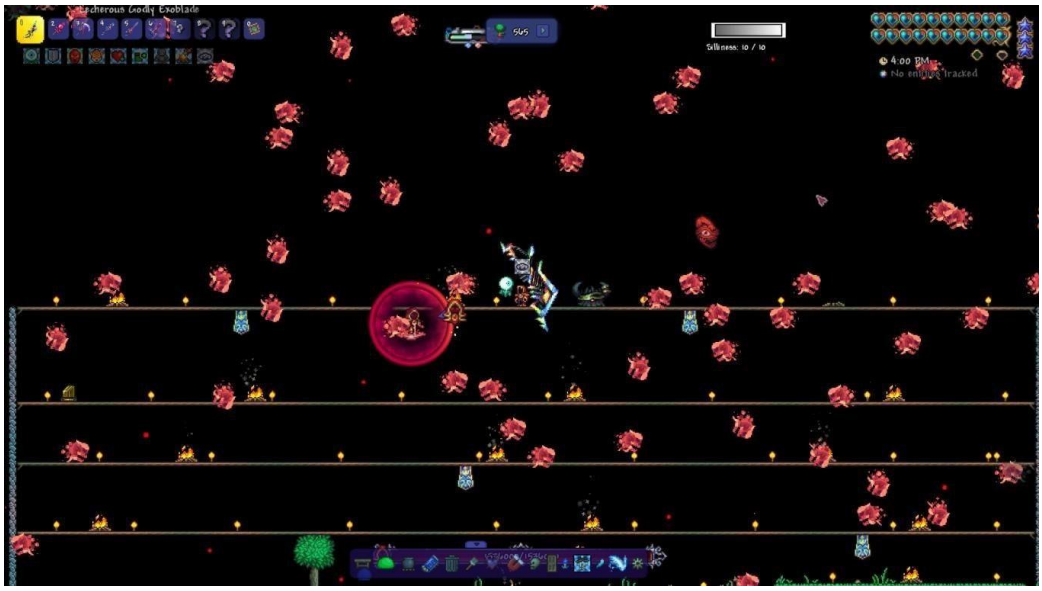
De acordo com Nystrom (2014), é viável a utilização do Object Pooling quando temos certos objetos que são criados e destruídos constantemente e

possuem tamanho similar. Quanto mais alta for a repetição e custo computacional que o alvo do Pooling estiver causando, mais recursos serão economizados e a performance geral do jogo aumentará, porém é importante notar que o uso do Pool em funções causará um maior tempo de execução das funções que o envolvem, sendo uma troca de tempo de execução por memória.

Para apresentar uma aplicação de jogo real onde a otimização com o uso de Pooling seria indicada é possível citar o jogo “Terraria”. O “Terraria” é um jogo com gráficos 2D de aventura e sandbox desenvolvido pela Re-Logic, tanto o personagem do jogador quanto uma porção dos inimigos que ele enfrenta durante o jogo são capazes de lançar diversos tipos de projéteis através de suas armas e poderes, a fim de causar danos um ao outro. Cada projétil tem propriedades diferentes, sejam elas seu dano, sua velocidade, seus efeitos visuais, entre outras características. Esses projéteis são tratados como um único objeto em seu código interno, e esse objeto é instanciado várias vezes conforme o necessário, ou seja, quando uma entidade aciona seus comportamentos. Cada instância de um projétil ocupa espaço temporariamente na memória até ser destruído, o que normalmente acontece após colidir com algum objeto do cenário, ou após um determinado tempo depois de sua criação.

Em Terraria, o jogador é capaz de evoluir seus equipamentos, e conforme o jogo avança, enfrenta inimigos mais difíceis de se derrotar. Dessa forma, durante as lutas mais intensas, tanto seus equipamentos quanto os inimigos encontram-se em um alto nível de poder devido à progressão do jogo, o que geralmente implica em uma quantidade muito maior de projéteis com efeitos visuais bastante notáveis preenchendo a tela. Isso ocasiona, em hardwares menos potentes, em uma queda drástica de taxa de quadros que pode, em alguns casos, tornar o jogo impossível de jogar. A figura 5 ilustra um dos casos em que pode haver uma queda de performance dentro do jogo em questão.

Figura 5 - Captura de tela de um dos confrontos do jogo Terraria.



Fonte: Produzido pelo autor.

O uso do Object Pooling poderia mitigar os impactos negativos da grande quantidade de projéteis gerados simultaneamente durante as batalhas mais intensas. Como cada projétil é tratado como um objeto independente que precisa ser instanciado e destruído dinamicamente, o custo de alocação e desalocação de memória aumenta consideravelmente conforme o número de projéteis na tela cresce. Com o uso de Object Pooling, os projéteis poderiam ser pré-alocados e reutilizados, reduzindo significativamente a sobrecarga na memória e no processamento. Dessa forma, o jogo poderia manter um desempenho mais estável, evitando quedas bruscas na taxa de quadros.

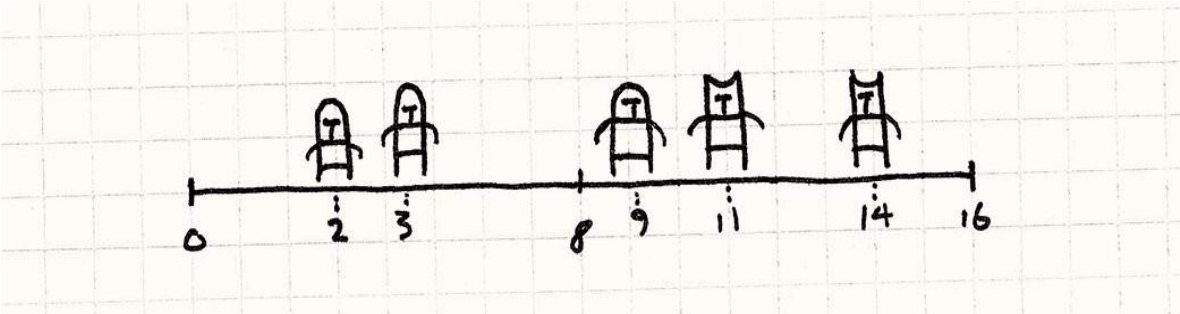
2.4.2 Spatial Partition

A técnica de Spatial Partition (ou "partição espacial") visa organizar os objetos de um jogo em uma estrutura de dados que leve em consideração suas posições no espaço, permitindo que sejam localizados de maneira eficiente. De acordo com Robert Nystrom (2014), a principal motivação por trás desse padrão é reduzir o custo computacional de consultas que buscam objetos próximos a uma posição específica. Isso é especialmente importante em jogos onde há muitos objetos em movimento no ambiente, e é necessário identificar rapidamente quais interagem entre si, como no caso de unidades em um campo de batalha.

Em um jogo, a posição dos objetos é uma característica fundamental. Desde a interação de unidades em um jogo de estratégia até a simulação de sons que

variam conforme a distância, entender onde os objetos estão localizados no mundo virtual é uma questão central. Quando há uma grande quantidade de objetos, realizar buscas repetidas em estruturas de dados desorganizadas pode se tornar um gargalo de performance. A utilização de uma Spatial Partition, conforme mostra o exemplo na Figura 6, organiza esses objetos em uma estrutura que facilita encontrar aqueles próximos a uma determinada localização, sem precisar percorrer toda a lista de objetos (NYSTROM, 2014).

Figura 6 - Ilustração de caso de uso do Spatial Partition, com a posição de vários personagens organizada em índices



Fonte: NYSTROM (2014).

Porém, embora essa técnica traga ganhos significativos de performance, ela também tem desvantagens. A organização dos objetos exige memória adicional para manter as estruturas de dados e a reorganização desses objetos à medida que eles se movem pelo ambiente. Isso representa uma troca entre tempo de execução e memória, sendo crucial avaliar a relação entre os dois fatores para decidir se a sua implementação é vantajosa (NYSTROM, 2014).

Dessa forma, a aplicação do padrão Spatial Partition é indicada em jogos onde a interação entre objetos é uma parte central da experiência e o número de objetos é grande o suficiente para que uma busca desordenada se torne ineficiente. A memória adicional necessária e a complexidade adicional na atualização das posições dos objetos devem ser cuidadosamente avaliadas, mas quando bem implementada, essa técnica pode trazer melhorias substanciais no desempenho do jogo (NYSTROM, 2014).

Para apresentar uma aplicação de jogo real onde a otimização com o uso de Spatial Partition poderia ser indicada é possível citar o jogo “Civilization VI”. O Civilization VI é um jogo de estratégia baseado em turnos desenvolvido pela Firaxis Games, onde cada jogador assume o papel de uma civilização histórica e deve gerir recursos e explorar o mapa do jogo de maneira inteligente para alcançar a vitória. Cada unidade, cidade e elemento do mapa do jogo ocupa uma posição dentro de uma grade hexagonal, e a eficiência na busca e interação entre esses elementos é crucial para a jogabilidade.

Em Civilization VI, a movimentação das unidades, a influência territorial e a visibilidade do mapa dependem diretamente da posição espacial dos objetos. Durante um turno, o jogo precisa calcular quais unidades podem interagir entre si, quais cidades estão dentro da área de influência de um jogador e quais regiões do mapa devem ser reveladas com base na linha de visão das unidades. Se essas operações forem realizadas sem uma estrutura otimizada, a performance do jogo poderia ser severamente impactada, especialmente em mapas de grande escala com um número elevado de unidades e cidades. Abaixo, na figura 7, é possível visualizar o cenário do jogo.

Figura 7 – Captura de tela do cenário do jogo Civilization VI.

Fonte: Produzido pelo autor.

Para mitigar esse tipo de problema, torna-se viável a utilização da técnica de Spatial Partitioning. A técnica pode ser usada para dividir a área do jogo em diversas grades menores, onde cada um desses espaços seria indexado para garantir consultas efetivas em um jogo que as exige em todo o seu cenário. Desse modo, a implementação dessa técnica no jogo poderia trazer um ganho considerável de tempo de execução, trazendo uma experiência mais agradável ao jogador.

2.4.3 Dirty Flag

A técnica de Dirty Flag envolve o rastreamento de alterações em dados primários por meio de uma "flag" (ou “bandeira”), uma variável que indica quando

os dados derivados, que são calculados a partir dos dados primários, estão desatualizados. O objetivo desse padrão é evitar a execução de cálculos custosos quando não há necessidade, economizando assim tempo de execução e, consequentemente, melhorando o desempenho do jogo. (NYSTROM, 2014).

O funcionamento básico desta técnica consiste em marcar a flag sempre que ocorre uma alteração nos dados primários. Quando os dados derivados forem requisitados, o sistema verifica se a flag está ativada. Caso esteja, o processo de cálculo é executado novamente, e a flag é limpa. Caso contrário, os dados derivados previamente armazenados em cache são utilizados, evitando a necessidade de reprocessamento. (NYSTROM, 2014)

Nystrom (2014) traz um exemplo prático de aplicação do Dirty Flag, abordando seu uso em jogos com gráficos hierárquicos, como em um grafo de cenas. Um grafo de cenas é uma estrutura de dados que organiza os objetos de uma cena, permitindo que a posição de cada objeto seja derivada de suas transformações locais e de seus pais na hierarquia. Isso facilita a movimentação em conjunto de objetos interligados, dado que cada movimentação de um objeto acima na hierarquia implica na movimentação automática de objetos abaixo dele.

No entanto, calcular as posições de todos os objetos a cada quadro é ineficiente, especialmente para objetos que não mudam de posição. Para otimizar, as posições podem ser armazenadas em cache (memória temporária) e recalculadas apenas quando necessário. O problema surge quando múltiplas alterações na hierarquia acontecem em um mesmo quadro, levando a cálculos redundantes de posições, onde a posição de um objeto é recalculada várias vezes inutilmente. (NYSTROM, 2014)

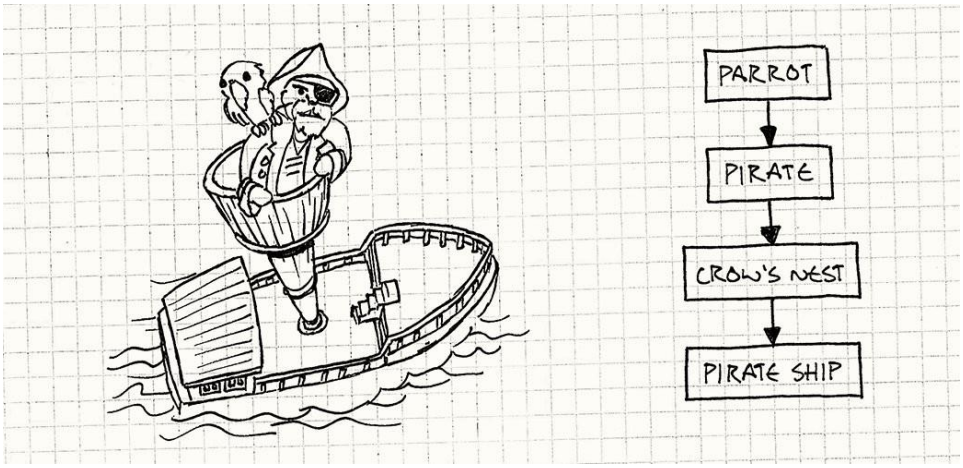
O Dirty Flag resolve essa questão ao marcar objetos cuja transformação global precisa ser atualizada, adiando o cálculo até o momento em que for realmente necessário, como durante o processo de renderização. Isso evita cálculos desnecessários e melhora o desempenho, garantindo que apenas as transformações relevantes sejam atualizadas. (NYSTROM, 2014).

Nystrom (2014) ilustra a técnica do Dirty Flag por meio de um exemplo que envolve um navio, que dentro de si contém um pirata, um papagaio e um ninho no mastro do navio. Todas essas entidades são objetos dentro de um jogo, de modo que as posições dos objetos são calculadas a partir dos objetos dos quais eles possuem uma dependência, ou seja, o papagaio depende do pirata para calcular sua posição, já o pirata depende do ninho, e o ninho depende do navio.

Quando a posição do navio é alterada, as posições dos outros objetos são recalculadas em cascata, sem a devida atenção a chamadas desnecessárias. Isso significa que, ao alterar a posição do navio, a posição do ninho é recalculada, e por consequência, a do pirata e a do papagaio também. O mesmo acontecerá com o

pirata e todos os objetos atrelados a ele. A Figura 8 abaixo ilustra o exemplo. (NYSTROM, 2014).

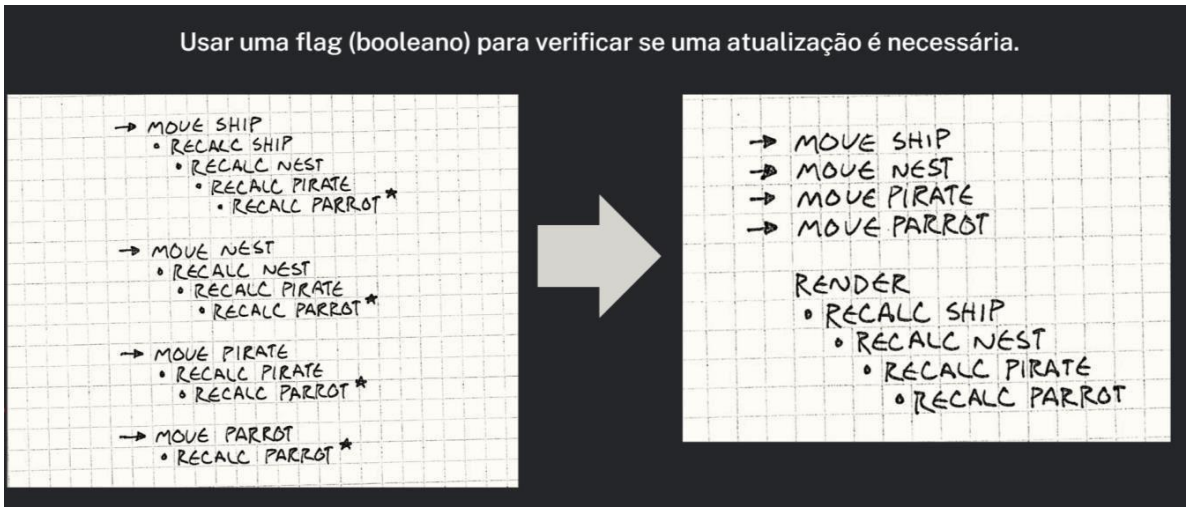
Figura 8 - Ilustração de exemplo onde o Dirty Flag é aplicável.



Fonte: NYSTROM (2014).

A aplicação do Dirty Flag permite que, cada vez que a posição do navio seja recalculada, os objetos atrelados a ele tenham suas posições calculadas apenas uma vez, evitando a chamada recursiva que consumiria mais tempo de execução, podendo causar um impacto na performance e consequentemente uma experiência negativa para o usuário. A Figura 9 ilustra a seguir uma comparação entre as duas abordagens. (NYSTROM, 2014).

Figura 9 - Comparação entre o número de chamadas das funções de transformação de posição, antes e depois da aplicação do Dirty Flag



Fonte: NYSTROM (2014).

Porém, assim como a técnica de Spatial Partition, não são todas as ocasiões em que o Dirty Flag é viável, pois também representa uma troca entre tempo de execução e memória. Descartar o cálculo em tempo real de dados derivados implica em mantê-los na memória caso seja necessário resgatá-los, do contrário não seria possível confirmar com precisão seus valores. Desse modo, utilizar o Dirty Flag é recomendado quando for benéfico e possível um consumo de memória maior em favor do tempo de execução. (NYSTROM, 2014).

Para apresentar uma aplicação de jogo real onde a otimização com o uso de Dirty Flag poderia ser indicada é possível citar o jogo “Sea of Thieves”. O jogo Sea of Thieves, desenvolvido pela Rare, é uma experiência multijogador onde os jogadores assumem o papel de piratas navegando por um vasto oceano, controlando navios em tempo real e interagindo com diversos elementos do cenário e de outros jogadores. O jogo apresenta uma física complexa para simular o balanço das embarcações, a movimentação dos personagens dentro dos navios e a interação com objetos dinâmicos, como barris, cordas e mastros. Esses elementos exigem cálculos constantes para atualizar suas posições, rotações e colisões. No entanto, recalcular esses valores a cada quadro para todos os objetos simultaneamente representaria um grande custo computacional, podendo impactar a taxa de quadros e a estabilidade da experiência de jogo. Veja abaixo, na figura 10, uma imagem que ilustra o jogo em questão.

Figura 10 – Captura de tela do cenário do jogo Sea of Thieves.

Fonte: Produzido pelo autor.

Em um caso como esse, a técnica de Dirty Flag torna-se viável para otimizar esse processo ao garantir que apenas os elementos que sofreram mudanças tenham seus valores recalculados. Por exemplo, dentro do jogo, quando um jogador

move o leme do navio, a posição e rotação da embarcação são alteradas. Contudo, elementos como barris e caixas dentro do navio não precisam ser recalculados a cada atualização do jogo, a menos que tenham sido deslocados por algum jogador ou pela própria física do jogo. Dessa forma, é evitada a necessidade de reprocessamento desnecessário e garantido um desempenho mais eficiente.

3 METODOLOGIA

3.1 CARACTERIZAÇÃO DA PESQUISA

Para realizar esta pesquisa, optou-se por realizar uma pesquisa bibliográfica e experimental. Na pesquisa bibliográfica objetiva-se realizar um estudo sobre as técnicas de otimização existentes utilizadas em jogos de computador, assim compreendendo as características, vantagens e desvantagens das abordagens existentes e em que situações estas abordagens são mais adequadas.

Após o levantamento bibliográfico será escolhido um jogo específico e uma técnica de otimização, com o objetivo de ambos serem utilizados na realização do experimento. Assim, a pesquisa também se caracteriza como experimental pois, após estudo bibliográfico, será realizada a implementação e teste de uma técnica de otimização em um ambiente controlado (jogo específico), com o objetivo de avaliar desempenho e eficácia do jogo através de comparação da execução dele antes e depois do uso da técnica.

Para realizar este experimento, será selecionado um jogo específico que possua código aberto e que apresente uma situação prática e real onde há uma oportunidade de introdução para uma técnica de otimização.

Em seguida, serão analisados quais possíveis gargalos de otimização estão comprometendo a performance do jogo, tendo então sido reconhecidos será possível escolher uma das técnicas de otimização que tenha maior conexão com o problema proposto. A técnica em questão será então aplicada ao código fonte do jogo, e em seguida será feita uma análise comparativa do uso do recurso a ser economizado, antes e depois da implementação da técnica, fazendo uso da ferramenta de depuração da Godot para visualizar essas informações.

3.2 ETAPAS DA PESQUISA

Desta forma, a pesquisa seguiu as seguintes fases:

1. Implementação da técnica de otimização:

Um jogo específico foi selecionado para a implementação das técnicas abordadas no estudo bibliográfico deste trabalho. Para o critério de seleção, foi observada a facilidade de alteração do código-fonte do jogo bem como situações em que a implementação das técnicas seja adequada para otimizar seções da aplicação. Para este intuito, foi necessário determinar quais possíveis gargalos teóricos de otimização poderiam ter efeito na

performance do jogo e então foram aplicadas técnicas de otimização que tinham o potencial de amenizar ou eliminar o problema proposto. Em seguida, foram feitas análises comparativas do uso dos recursos a serem economizados, antes e depois da implementação das técnicas, fazendo uso da ferramenta de depuração da Godot e demais funções de logging disponíveis na engine para visualizar essas informações.

2. Comparação de Desempenho:

Para realizar o experimento e comparação de desempenho do jogo antes e depois das aplicações das técnicas de otimização foram utilizados os seguintes critérios:

- Tempo de execução em funções:

Se as funções do jogo, incluindo as relacionadas à técnica implementada, forem complexas e demorarem muito a executar, o tempo de execução pode se tornar um gargalo e comprometer o tempo de atualização do estado do jogo. Nesse caso, o jogo demora mais tempo para responder à cada ação do jogador, causando uma experiência negativa.

- Consumo de memória.

Se o jogo não controlar adequadamente a alocação de memória para os objetos do jogo, o consumo de memória pode se tornar um gargalo e comprometer a taxa de quadros da aplicação. Isto pode, como explorado anteriormente, tornar o jogo impossível de jogar dependendo da escala do problema que ele apresenta, em termos de tamanho dos objetos e quantas vezes estes são instanciados e alocados na memória.

3. Análise e apresentação dos resultados.

Por meio do depurador da Godot Engine e de funções de logging disponíveis na engine, foram obtidos dados referentes ao uso de recursos computacionais após a execução da aplicação, antes e depois da implementação da técnica de otimização selecionada. Para o consumo de memória, a aba “Monitor” do depurador foi utilizada para visualizar o pico de memória alocada para os objetos do jogo. Para o tempo de execução, a aba “Profiler” foi utilizada para visualizar o pico de tempo que as funções alteradas

levam para processar. Após a coleta dos dados, foram feitas análises que desenvolveram conclusões sobre a eficiência das técnicas na solução dos problemas propostos.

4 O JOGO PLATFORMER 2D

O Platformer2D é um jogo em duas dimensões, com um personagem controlável, que possui uma arma capaz de lançar projéteis. O jogador é capaz de mover-se em todas as direções, sendo capaz de pular para alcançar plataformas mais altas. O jogo conta com alguns inimigos, os quais os projéteis lançados pelo personagem jogável são capazes de destruir. Também há algumas moedas espalhadas pelo cenário, que o jogador é capaz de coletar. Tanto os coletáveis quanto os inimigos surgem apenas uma vez, e ao serem derrotados e coletados respectivamente, são excluídos permanentemente da memória até que o jogo seja reiniciado. A figura 11 abaixo mostra o cenário do jogo em questão.

Figura 11 - Captura de tela do cenário do jogo Platformer2d.

Fonte: Produzido pelo autor.

O Platformer2D possui uma característica essencial que o leva a ser um candidato adequado para simular um problema real com o Pooling, sendo esta sua mecânica de projéteis, que é semelhante à de um jogo previamente apresentado neste trabalho, Terraria. Para cada tiro realizado pelo jogador, um novo projétil é instanciado como um objeto que ocupa determinado espaço na memória, o que sem a otimização necessária pode causar uma queda de desempenho devido ao custo desnecessário de processamento.

Para os fins deste trabalho, o Platformer2d foi escolhido por conta de sua natureza de código aberto, sendo um dos jogos de demonstração da Godot. Isso torna possível, alterando seu código fonte, simular essa problemática de forma mais isolada, bem como analisar facilmente os resultados obtidos após aplicação

da técnica de otimização.

4.1 APLICAÇÃO DA TÉCNICA OBJECT POOLING

Os trechos do jogo selecionados para a aplicação da otimização com Object Pooling foram os eventos relacionados ao disparo de projéteis pelo jogador, juntamente aos inimigos que surgem no cenário, ambos com suas particularidades.

Quanto à aplicação da técnica nos projéteis, vemos que, na sua forma original, a chamada deste evento cria um objeto toda vez que o jogador atira um projétil, o que pode gerar um uso desnecessário de memória e eventual perda de desempenho caso muitos projéteis estejam em cena. Abaixo, na figura 12, podemos visualizar o código original relacionado a este evento:

Figura 12 - Código da função de disparo de projéteis do Platformer2d

```
func shoot(direction: float = 1.0) -> bool:
    if not timer.is_stopped():
        return false
    var bullet := BULLET_SCENE.instantiate() as Bullet
    bullet.global_position = global_position
    bullet.linear_velocity = Vector2(direction * BULLET_VELOCITY, 0.0)

    bullet.set_as_top_level(true)
    add_child(bullet)
    sound_shoot.play()
    timer.start()
    return true
```

Fonte: Produzido pelo autor.

Analisando a complexidade do evento “shoot” da Figura 10 acima, é possível observar que, se forem realizadas N chamadas ao evento “shoot” (evento relacionado ao disparo de um projétil), serão realizadas também N chamadas a instanciação do objeto “bullet”. Assim, quanto maior o número de projéteis, consequentemente haverá maior uso de memória ao realizar operações envolvidas na criação e definição das propriedades de cada projétil.

Considerando que dependendo da quantidade de memória disponível e da quantidade de projéteis envolvidos, a criação de um objeto para cada projétil disparado pode ser considerada um ponto de gargalo no jogo, uma estratégia de solução para este problema seria utilizar a técnica de otimização Poolling. A solução consiste na criação de uma lista de objetos e também a modificação do evento “shoot” de maneira que toda vez em que se realizar um disparo, antes de ser criado um novo objeto “bullet”, seja verificado se existe um objeto similar na lista. Caso ele

exista, é possível reutilizá-lo, e dessa forma não existirá necessidade de realizar a instanciação do objeto novamente.

A primeira etapa da implementação do Object Pooling foi a criação do pool de objetos, neste caso, de projéteis lançados pelo jogador. Isso pode ser feito em GDScript com a simples criação de uma lista (ou array) que armazenará os objetos que representam esses projéteis. O tamanho da lista será indefinido, de modo que um número ilimitado de objetos pode ser armazenado nela, conforme mostra a Figura 13 abaixo.

Figura 13 - Código de criação do pool de projéteis.

Fonte: Produzido pelo autor.

A segunda etapa consistiu na reutilização dos objetos armazenados no pool. Sempre que o jogador dispara um projétil, o jogo verifica se há um objeto disponível no pool para ser utilizado. Quando encontrado, a localização do projétil é atualizada para prepará-lo para o ataque, bem como sua física, animações, e outros comportamentos atrelados a ele passam a ser ativados para que ele desempenhe seu papel no jogo, evitando a criação de novos objetos desnecessários.

Na Figura 14 abaixo é possível visualizar o trecho de código que apresenta a identificação de um objeto válido no pool, e logo em seguida, o momento do disparo do projétil no jogo.

Figura 14 - Código de disparo modificado para implementação do Object Pooling.

```
func shoot(direction: float = 1.0) -> bool:
    if not timer.is_stopped():
        return false

    var bullet: Bullet

    if bullet_pool.size() > 0 and is_instance_valid(bullet_pool[(bullet_pool.size() - 1)]):
            bullet = bullet_pool.pop_back()
            if bullet.is_queued_for_deletion() or bullet.get_parent() == null:
                return shoot(direction)
            if bullet.get_parent() != null:
                bullet.get_parent().remove_child(bullet)
            bullet.visible = true
            bullet.set_process(true)
    else:
        bullet = BULLET_SCENE.instantiate() as Bullet
        bullet.connect("ready_to_recycle", Callable(self, "_recycle_bullet"))

    bullet.global_position = global_position
    bullet.linear_velocity = Vector2(direction * BULLET_VELOCITY, 0.0)
    bullet.set_as_top_level(true)
    add_child(bullet)

    sound_shoot.play()
    timer.start()
```

Fonte: Produzido pelo autor.

Analisando o código fonte da Figura 14 podemos observar que, agora, a criação de objetos do tipo “bullet” não está mais diretamente relacionada a quantidade de vezes que o evento “shoot” é invocado. O melhor caso, nessa implementação, é quando para todas as vezes que o evento “shoot” for chamado existir um objeto “bullet” na lista de Pooling que possa ser recuperado sem a necessidade de instanciação de um novo objeto, nesse caso o gasto de recurso de memória com instanciação de um novo objeto não ocorreria e somente a atualização das propriedades do objeto seria realizada, mesmo que várias chamadas ao evento fossem executadas.

O pior caso, nessa implementação, seria onde todos os objetos “bullet” disparados ainda não fazem parte do pool, nesse caso a quantidade de instanciações seria a mesma da quantidade de chamadas ao evento. Assim, é importante observar que, para que a implementação proposta na Figura 14 alcance

o objetivo desejado, é necessário um gerenciamento adequado da criação de novos objetos.

Ainda sobre a implementação da Figura 14, é possível observar que, em situações em que o pool está vazio e não há objetos disponíveis no momento de um disparo, o sistema permite a criação de um novo projétil. Após ser utilizado, esse projétil é adicionado ao pool, garantindo que a mecânica de jogo não seja interrompida.

Na sequência, foi implementada a lógica de desativação e retorno ao pool. Sempre que um projétil colide com outro objeto ou alcança o tempo máximo de vida pré definido, ele é desativado, visualmente desaparecendo para o usuário, e é imediatamente retornado ao pool, ficando disponível para futuros disparos. A Figura 15 abaixo apresenta o trecho de código relacionado a lógica de desativação e retorno ao pool.

Figura 15 - Código de reciclagem de projéteis.

```
func _recycle_bullet(bullet: Bullet) -> void:
    if not is_instance_valid(bullet) or bullet.is_queued_for_deletion() or bullet.get_parent() == null:
        return
    bullet_pool.append(bullet)
    for bullet_p in bullet_pool:
        if not is_instance_valid(bullet_p):
            bullet_pool.erase(bullet_p)

    bullet.hide()
    bullet.set_process(false)
```

Fonte: Produzido pelo autor.

É importante notar também que, devido ao sistema de coleta de lixo da Godot, que pode marcar automaticamente para deleção os objetos que desativamos para posterior utilização, é necessário implementar uma verificação que garanta que os projéteis no pool sempre serão válidos. Na Godot Engine, um objeto marcado para deleção da memória, ou que já tenha sido deletado dela, não é uma instância válida que possa ser utilizada no jogo. Dessa forma, foi implementada uma varredura periódica, em que toda vez que um objeto é adicionado ao pool, é acionada uma função que atualiza o pool para conter apenas objetos válidos para uso. A figura 17 abaixo ilustra a função descrita.

Figura 16 - Código de validação de objetos do pool.

```
func _clean_bullet_pool() -> void:
    var valid_bullets := []
    for bullet in bullet_pool:
        if is_instance_valid(bullet) and not bullet.is_queued_for_deletion():
            valid_bullets.append(bullet)
    bullet_pool = valid_bullets
```

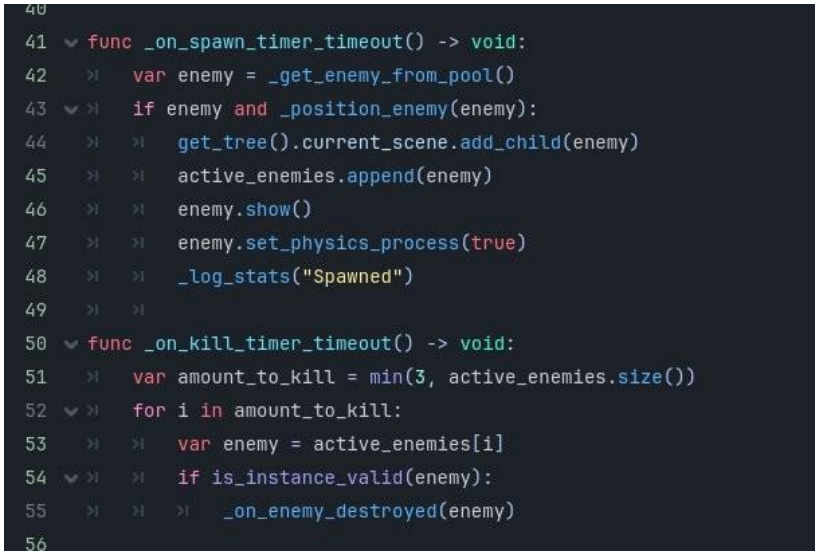
Fonte: Produzido pelo autor.

Além da implementação do Object Pooling no sistema de disparo de projéteis, outro trecho selecionado para demonstrar a eficácia dessa técnica de otimização foram os inimigos do jogo. No Platformer2d, alguns inimigos são dispostos no cenário assim que o jogo é iniciado, e possuem um comportamento definido que os permite movimentar-se pelo cenário. Os inimigos podem ser destruídos pelos projéteis que o jogador dispara, fazendo com que seus objetos sejam descartados da memória e não ressurjam mais até que o jogo seja reiniciado.

Para melhor demonstrar uma implementação do Pooling neste tipo de objeto, foi desenvolvido um script especificamente para este estudo: o “EnemySpawner” (ou Invocador de Inimigos). Este script gerencia a criação e reutilização de inimigos, permitindo que seja feita uma análise mais proveitosa para os fins deste trabalho. O EnemySpawner foi desenvolvido em duas versões, uma que não utiliza o Object Pooling, destruindo os inimigos da memória quando estes forem derrotados, e outra que utiliza a técnica, reciclando os objetos.

Na versão sem Object Pooling, o EnemySpawner cria um objeto de inimigo a cada evento de surgimento de um inimigo. Esse processo é desencadeado por um temporizador que adiciona inimigos à cena a cada 1 segundo. A cada 10 segundos, para que se possa observar o comportamento do objeto de modo consistente, o script destrói automaticamente 3 inimigos do cenário. Nesta versão do código, inimigos destruídos tem seus objetos descartados da memória. A Figura 17 exibe o trecho de código correspondente a essa ação.

Figura 17 - Código das funções de surgimento e destruição de inimigos, respectivamente, implementadas no Platformer2d.



```
41 func _on_spawn_timer_timeout() -> void:
42     var enemy = _get_enemy_from_pool()
43     if enemy and _position_enemy(enemy):
44         get_tree().current_scene.add_child(enemy)
45         active_enemies.append(enemy)
46         enemy.show()
47         enemy.set_physics_process(true)
48         _log_stats("Spawned")
49
50 func _on_kill_timer_timeout() -> void:
51     var amount_to_kill = min(3, active_enemies.size())
52     for i in amount_to_kill:
53         var enemy = active_enemies[i]
54         if is_instance_valid(enemy):
55             _on_enemy_destroyed(enemy)
56
```

Fonte: Produzido pelo autor.

A análise da complexidade do código revela que N chamadas ao evento de surgimento de inimigo resultam em N instanciações de objetos, cada uma demandando alocação de memória adicional. Assim como no caso dos projéteis, esse processo pode gerar gargalos de desempenho devido ao acúmulo de custos computacionais, porém objetos mais complexos como inimigos tornam esse cuidado ainda mais importante.

Em jogos como o Platformer2d, inimigos permanecem ativos no cenário até que sejam destruídos. Isso significa que, ao contrário de objetos que são limpos do cenário periodicamente como os projéteis, os inimigos podem se acumular indefinidamente. Ademais, os inimigos possuem configurações adicionais que permitem seu funcionamento, como sua inteligência artificial e suas várias animações que representam cada estado do seu comportamento. A soma desses fatores significa que uma otimização de código nesse quesito é essencial para que não haja uma grave queda de desempenho.

A implementação inicial do EnemySpawner consistiu na criação de um sistema que instancia inimigos em intervalos regulares, posicionando-os aleatoriamente ao redor do jogador, dentro de um raio definido. O componente utiliza um temporizador para invocar inimigos e verificar colisões para garantir que os inimigos sejam posicionados em locais válidos. Além disso, como citado anteriormente, um segundo temporizador foi adicionado para simular a destruição de inimigos a cada 10 segundos, facilitando a análise de desempenho. A figura 18 abaixo ilustra a inicialização desse componente na função “_ready”:

Figura 18 - Código da função “_ready”, responsável pela inicialização do EnemySpawner.

```gdscript
func _ready() -> void:
    if not is_instance_valid(player):
        push_error("EnemySpawner: No valid player assigned!")
        return

    spawn_timer = Timer.new()
    var kill_timer: Timer
    spawn_timer.wait_time = spawn_interval
    spawn_timer.one_shot = false
    spawn_timer.timeout.connect(_on_spawn_timer_timeout)
    add_child(spawn_timer)
    spawn_timer.start()

    kill_timer = Timer.new()
    kill_timer.wait_time = 10.0
    kill_timer.one_shot = false
    kill_timer.timeout.connect(_on_kill_timer_timeout)
    add_child(kill_timer)
    kill_timer.start()
    set_process(true)

```

Fonte: Produzido pelo autor.

A primeira etapa da implementação do Object Pooling no EnemySpawner envolveu a criação de um pool de inimigos. Essa estrutura é semelhante ao pool de projéteis, possuindo as mesmas propriedades, isto é, uma simples lista dinâmica que conterá os objetos reciclados pela implementação da técnica. A figura 19 abaixo mostra o código correspondente à criação do pool de inimigos.

Figura 19 - Código de criação do pool de inimigos.

```gdscript
13
14    var enemy_pool: Array[CharacterBody2D] = []
```

Fonte: Produzido pelo autor.

A segunda etapa consistiu na lógica de reutilização dos inimigos armazenados no pool. Quando o evento de spawn é acionado, o sistema verifica se há um inimigo disponível no pool. Caso um objeto válido seja encontrado, suas propriedades, como posição e estado físico, são redefinidas, e o inimigo é reativado na cena. Se o pool estiver vazio, um novo inimigo é instanciado, mas permanece elegível para reutilização futura. A figura 20 abaixo mostra a função responsável por gerenciar essa lógica:

Figura 20 - Código de reutilização dos inimigos do pool.

```
50
51 func _get_enemy_from_pool() -> CharacterBody2D:
52     while enemy_pool.size() > 0:
53         var pooled = enemy_pool.pop_back()
54         if is_instance_valid(pooled) and not pooled.is_queued_for_deletion():
55             _reset_enemy(pooled)
56             return pooled
57
58     var enemy = ENEMY_SCENE.instantiate() as CharacterBody2D
59     total_instantiated += 1
60
61     if enemy.has_signal("destroyed"):
62         enemy.connect("destroyed", _on_enemy_destroyed.bind(enemy))
63
64     _reset_enemy(enemy)
65     return enemy
```

Fonte: Produzido pelo autor.

A terceira etapa envolveu o gerenciamento dos inimigos após sua destruição, garantindo que fossem retornados ao pool para reutilização. Quando um inimigo é destruído, ele é desativado, removido da cena ativa e adicionado novamente ao pool. Durante esse processo, suas propriedades são redefinidas para evitar comportamentos indesejados em utilizações futuras. A figura 21 a seguir mostra o trecho do código responsável por esse comportamento:

Figura 21 - Código de reciclagem dos inimigos.

```
93
94 func _on_enemy_destroyed(enemy: CharacterBody2D) -> void:
95     if not is_instance_valid(enemy):
96         return
97     enemy.hide()
98     enemy.set_physics_process(false)
99     if enemy.get_parent():
100        enemy.get_parent().remove_child(enemy)
101    active_enemies.erase(enemy)
102    enemy_pool.append(enemy)
103    _log_stats("Destroyed")
104
```

Fonte: Produzido pelo autor.

4.1.1 Resultados comparativos do Object Pooling

Neste tópico serão dispostos resultados comparativos entre o desempenho da aplicação, em termos de consumo de memória e tempo, antes e depois da implementação da técnica. Primeiramente, serão expostos os resultados referentes ao Object Pooling aplicado a projéteis e, em seguida, aqueles referentes ao Pooling de inimigos.

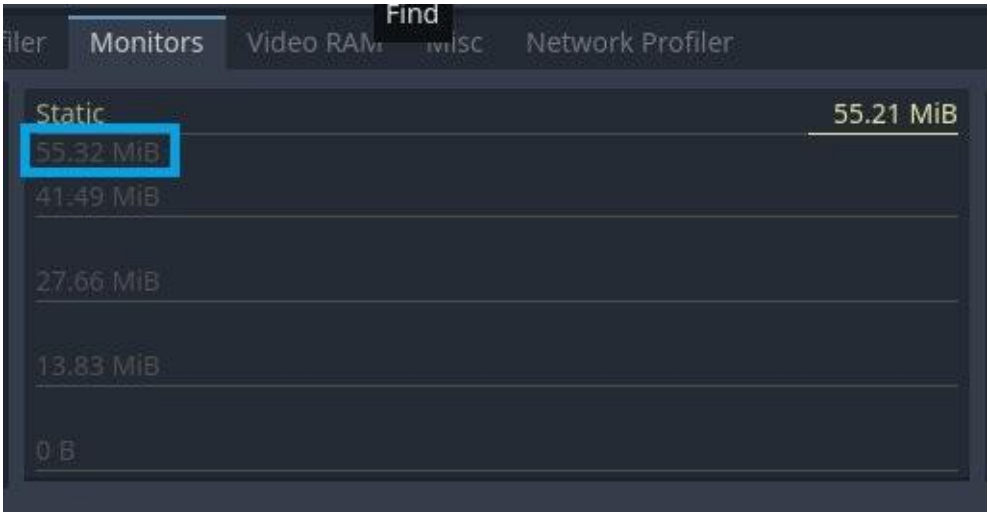
Para a implementação da técnica aos projéteis, a comparação foi realizada considerando o seguinte cenário do jogo: Após o jogador assumir o controle do

personagem, ele o direciona para a parede na parte esquerda do cenário e atira continuamente contra ela. Neste cenário, não são considerados a presença de inimigos ou outros objetos presentes no Platformer2d.

Com os disparos do personagem, o consumo de memória se eleva e seu pico é registrado no depurador da Godot. O mesmo acontece para o tempo de execução das funções relacionadas ao evento de disparo, onde também é possível visualizar quanto tempo a aplicação está demorando para processar essas funções.

Na métrica de consumo de memória foi observado uma redução evidente de consumo de memória após a implementação do pool: O pico de consumo de memória antes da utilização da técnica foi de, como destacado na figura 22 abaixo, cerca de 55.32 MiB (Mebibytes).

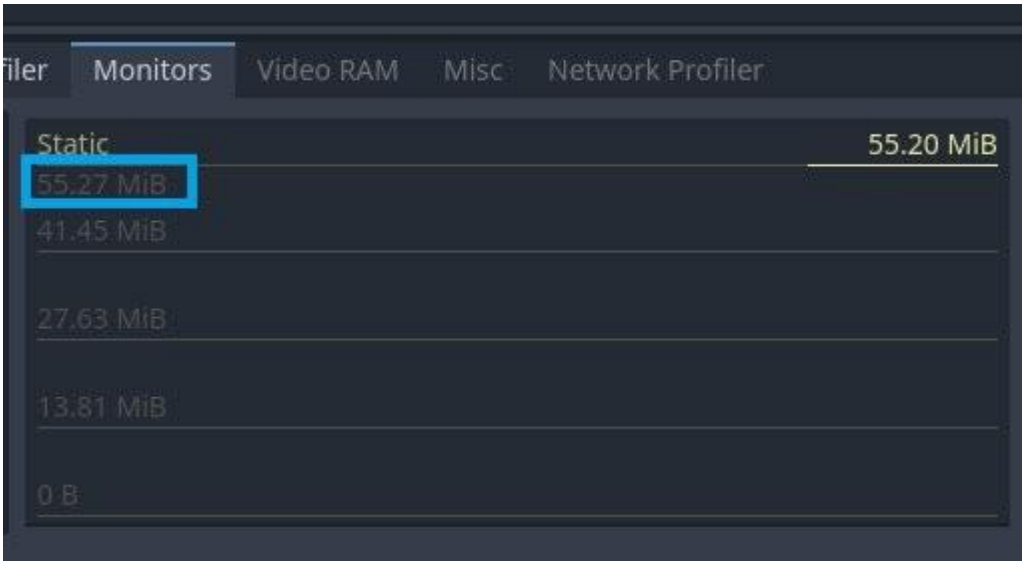
Figura 22 – Captura de tela do consumo de memória do Platformer2d antes da utilização do pooling de projéteis.



Fonte: Produzido pelo autor.

Em comparação, na mesma situação, o pico de consumo de memória após a utilização da técnica foi de, como destacado na figura 23 abaixo, cerca de 55.27 MiB (Mebibytes).

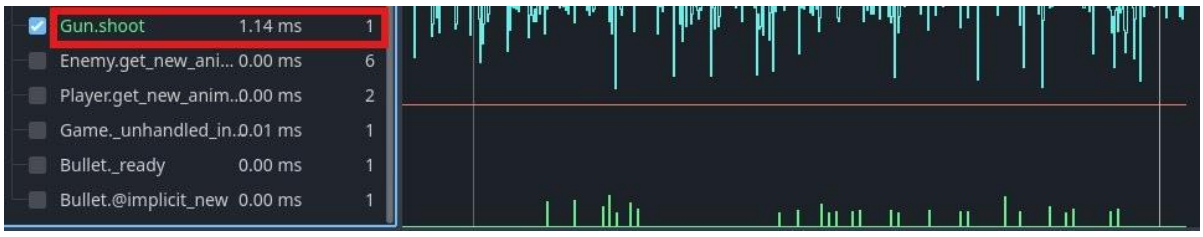
Figura 23 - Captura de tela do consumo de memória do Platformer2d após a utilização do pooling de projéteis.



Fonte: Produzido pelo autor.

Para a métrica de tempo de execução, foi observado um aumento de tempo de execução da função “shoot”. O pico de tempo de execução da função “shoot” antes da implementação do pooling foi de 1.12 milissegundos, como destacado na figura 24 abaixo:

Figura 24 – Tempo de execução da função “shoot” antes da implementação do pooling.



Fonte: Produzido pelo autor.

O pico de tempo de execução da função “shoot” depois da implementação do pooling foi de 3.73 milissegundos, como destacado na figura 25 abaixo:

Figura 25 - Tempo de execução da função “shoot” após a implementação do pooling.

Fonte: Produzido pelo autor.

Em relação à implementação da técnica para os inimigos, a comparação foi feita na seguinte situação: O jogo foi executado e deixado aberto por 30 segundos, onde assim os dados foram coletados a partir das ferramentas de depuração.

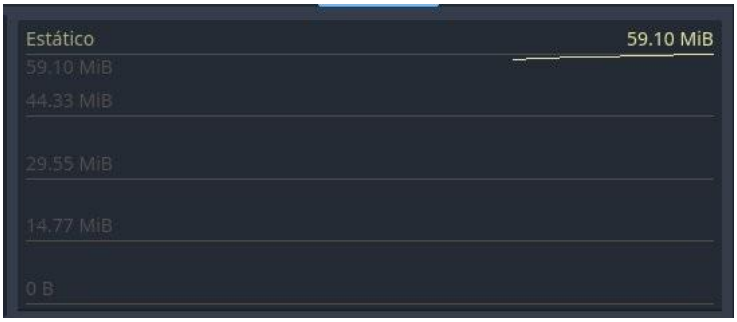
Na métrica de consumo de memória foi observado uma redução evidente de consumo de memória após a implementação do pool: O pico de consumo de memória antes da utilização da técnica foi de, como destacado na figura 26 abaixo, cerca de 59.20 MiB (Mebibytes).

Figura 26 – Captura de tela do consumo de memória do Platformer2d antes da utilização do pooling de inimigos.

Fonte: Produzido pelo autor.

Em comparação, na mesma situação, o pico de consumo de memória após a utilização da técnica foi de, como destacado na figura 27 abaixo, cerca de 59.10 MiB (Mebibytes).

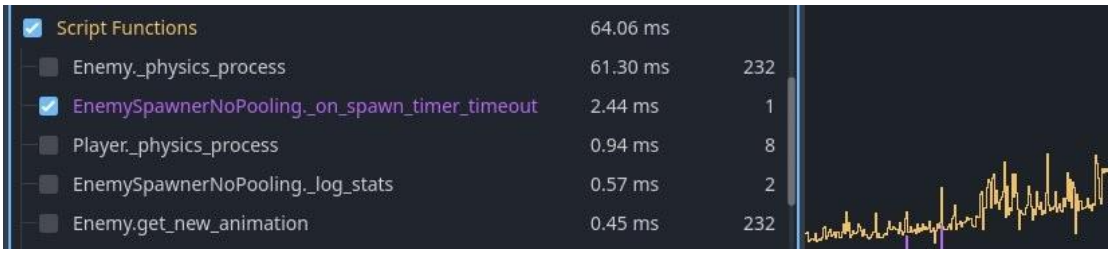
Figura 27 – Captura de tela do consumo de memória do Platformer2d depois da utilização do pooling de inimigos.



Fonte: Produzido pelo autor.

Para a métrica de tempo de execução, foi observado um aumento de tempo de execução da função "_on_spawn_timer_timeout", responsável pelo surgimento de inimigos. O pico de tempo de execução da função antes da implementação do pooling foi de 2.44 milissegundos, como destacado na figura 28 abaixo:

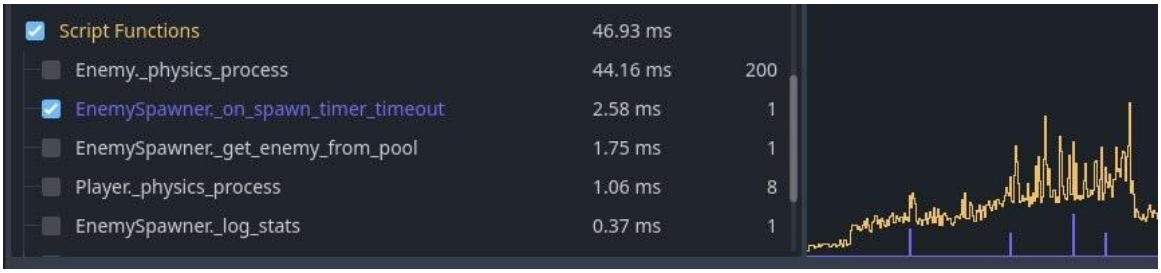
Figura 28 – Tempo de execução da função "_on_spawn_timer_timeout" antes da implementação do pooling.



Fonte: Produzido pelo autor.

O pico de tempo de execução da função depois da implementação do pooling foi de 2.58 milissegundos, como destacado na figura 29 abaixo:

Figura 29 – Tempo de execução da função "_on_spawn_timer_timeout" depois da implementação do pooling.



Fonte: Produzido pelo autor.

4.1.2 Análise dos resultados do Object Pooling

Com os dados obtidos através da execução da aplicação com as implementações antes e após o pooling, incluindo as comparações de consumo de memória e tempo de execução, somos capazes de definir algumas conclusões.

A primeira conclusão é que o uso do pool de fato ocasionou uma redução no consumo de memória através de sua lógica, o que cumpre as expectativas definidas na fundamentação teórica deste trabalho. O uso do Pooling economizou cerca de 5 MiB quando aplicado aos projéteis do Platformer2d, e 10 MiB quando aplicado em seus inimigos, em relação às abordagens que não utilizam a técnica, o que deve aumentar ainda mais em uma aplicação cujos objetos tenham maior impacto na memória.

A segunda conclusão é que o uso do pool de fato agiu como uma troca entre tempo de execução em favor de menor consumo de memória, assim como exposto anteriormente na fundamentação teórica. As funções de administração do pool adicionam ao tempo das funções projéteis e surgimento de inimigos, dessa forma, a ação demora mais tempo para finalizar.

Ademais, foi possível atestar a escalabilidade do Object Pooling ao testar sua implementação em dois objetos de complexidades diferentes. Quando aplicado aos inimigos do Platformer2d, que possuem comportamentos mais avançados e permanecem indefinidamente no cenário até serem destruídos, houve uma economia maior do que quando aplicado aos projéteis do jogo, que são mais leves e desaparecem rapidamente após disparados. Isto demonstra que, como explorado neste estudo, o Pooling é uma técnica que ganha mais performance conforme a complexidade dos objetos envolvidos.

Dessa forma, esses resultados comprovam a eficiência da técnica de Object Pooling em reduzir o consumo de memória e potencialmente aumentar o desempenho de um jogo, oferecendo uma melhor experiência para os consumidores e beneficiando a indústria de jogos como um todo. No entanto, os dados obtidos também reafirmam a necessidade de se fazer uma análise dedicada à escolha entre uma técnica ou outra de otimização, ou mesmo à necessidade de utilizar essas técnicas em primeiro lugar em uma determinada aplicação. Essas técnicas devem ser implementadas conforme o necessário dependendo do caso em questão, não como um padrão absoluto para todos os jogos.

4.1.3 APLICAÇÃO DA TÉCNICA SPATIAL PARTITION

O trecho do jogo selecionado para a aplicação da otimização com Spatial Partitioning foi o evento relacionado à detecção de colisão entre inimigos e outros objetos, especialmente em cenários com muitos elementos interativos.

Na sua forma original, a detecção de colisão dos inimigos no Platformer2d utiliza o sistema de física integrado do Godot, que realiza verificações de colisão automaticamente. No entanto, em cenas com muitos objetos dinâmicos, como os inimigos, o custo dessas verificações pode aumentar, potencialmente impactando o desempenho. Abaixo, na figura 30, podemos visualizar o código original relacionado a este sistema:

Figura 30 - Código do sistema de detecção de colisão de inimigos sem Spatial Partitioning

```gdscript
func _physics_process(delta: float) -> void:
    if _state == State.WALKING and velocity.is_zero_approx():
        velocity.x = WALK_SPEED
    velocity.y += gravity * delta
    if not floor_detector_left.is_colliding():
        velocity.x = WALK_SPEED
    elif not floor_detector_right.is_colliding():
        velocity.x = -WALK_SPEED

    if is_on_wall():
        velocity.x = -velocity.x

    move_and_slide()

    if velocity.x > 0.0:
        sprite.scale.x = 0.8
    elif velocity.x < 0.0:
        sprite.scale.x = -0.8

    var animation := get_new_animation()
    if animation != animation_player.current_animation:
        animation_player.play(animation)
```

Fonte: Produzido pelo autor.

O sistema de detecção de colisão dos inimigos no Platformer2D utiliza a física integrada da Godot, que realiza automaticamente as verificações de colisão entre os objetos da cena com base em suas formas de colisão. Essas formas são estruturas geométricas que definem a área lógica de interação de um objeto, independentemente de sua aparência visual. Um personagem com aparência humana, por exemplo, pode ter sua colisão representada por um simples retângulo, usado apenas para os cálculos físicos do jogo. A Godot Engine identifica a sobreposição entre essas estruturas para definir as colisões do jogo.

Esse sistema é atualizado constantemente, a cada quadro do jogo, sendo

essencial para o funcionamento de uma grande parte dos jogos no mercado. No entanto, em cenas com muitos objetos dinâmicos e múltiplas verificações de colisão, o custo computacional pode aumentar, especialmente em cenários com interações complexas. Uma estratégia para otimizar essas verificações é implementar um sistema personalizado de spatial partitioning, como uma grade espacial. Isso reduz significativamente o número de cálculos necessários, melhorando o desempenho em cenas complexas.

A primeira etapa da implementação do Spatial Partitioning foi a criação da estrutura de grid. Isso foi feito com a criação de um dicionário que mapeia células representadas por coordenadas para uma lista de objetos, permitindo que verificações de colisão sejam restritas aos objetos nas células adjacentes à posição de interesse, em vez de considerar todos os objetos na cena. O trecho de código que destaca a inicialização da estrutura, juntamente a variáveis para logging, pode ser visualizada na Figura 31 abaixo:

Figura 31 - Código da estrutura de grid do Spatial Partitioning

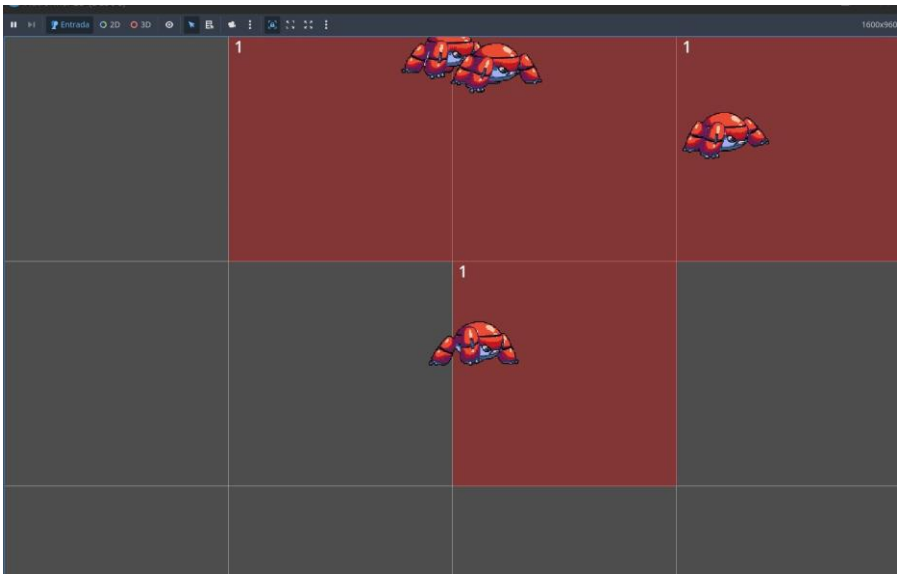
```
class_name SpatialGrid extends Node

@export var cell_size: Vector2 = Vector2(200, 200)
@export var grid_size: Vector2 = Vector2(10000, 10000)


@export var debug_draw: bool = true
@export var debug_color: Color = Color(1, 1, 1, 0.5)
@export var debug_cell_color: Color = Color(1, 0, 0, 0.3)
@export var debug_text_color: Color = Color(1, 1, 1, 1.0)

@export var enable_performance_logging: bool = true
@export var log_interval: float = 1.0

var grid: Dictionary = {}

var _last_log_time: float = 0.0
var _total_objects_checked: int = 0
var _total_collision_checks: int = 0
var _total_collision_time: float = 0.0
var _total_update_time: float = 0.0
var _total_add_time: float = 0.0
var _total_remove_time: float = 0.0
var _total_operations: int = 0
```

Fonte: Produzido pelo autor.

Para a implementação feita neste trabalho, as células do grid são definidas

dinamicamente, de modo que apenas células ativas, ou seja, que contém objetos em si, fazem parte da estrutura e consomem memória no sistema. Dessa forma o gasto de memória que o grid consome fica limitado ao número de objetos dispostos nas várias células da partição. Veja abaixo, na figura 32, uma demonstração visual do grid ativo:

Figura 32 - Representação visual do Spatial Grid com os inimigos do Platformer2d.



Fonte: Produzido pelo autor.

A segunda etapa consistiu na implementação do sistema de manutenção de posição dos objetos no grid espacial, garantindo que a localização dos objetos seja atualizada de forma eficiente à medida que se movem na cena. Sempre que um objeto se move, o sistema calcula as células correspondentes à sua nova posição utilizando a função get_cell_index, que converte as coordenadas do objeto em índices de células no grid com base no tamanho definido das células. A atualização da posição é realizada em duas etapas principais: primeiro, o objeto é removido das células antigas por meio da função remove_from_grid, que verifica o índice da célula associada à posição anterior e elimina o objeto da lista de objetos dessa célula, apagando-a completamente se ficar vazia. Em seguida, o objeto é adicionado às novas células correspondentes à sua nova posição pela função add_to_grid, que insere o objeto na lista da célula apropriada, criando-a se necessário. Esse processo é encapsulado pela função update_object, que coordena a remoção e adição do objeto, garantindo que o grid reflita precisamente a nova disposição dos objetos, como ilustrado na Figura 33:

Figura 33 - Código de manutenção do grid de Spatial Partition

```
func add_to_grid(object: Node2D, position: Vector2) -> void:
    var start_time := Time.get_ticks_usec() / 1000000.0
    var cell_index = get_cell_index(position)
    if not grid.has(cell_index):
        grid[cell_index] = []
    if not grid[cell_index].has(object):
        grid[cell_index].append(object)
        if debug_draw:
            debug_draw_node.queue_redraw()

    _total_operations += 1
    _total_add_time += (Time.get_ticks_usec() / 1000000.0) - start_time

func remove_from_grid(object: Node2D, position: Vector2) -> void:
    var start_time := Time.get_ticks_usec() / 1000000.0
    var cell_index = get_cell_index(position)
    if grid.has(cell_index):
        grid[cell_index].erase(object)
        if grid[cell_index].is_empty():
            grid.erase(cell_index)
        if debug_draw:
            debug_draw_node.queue_redraw()

    _total_operations += 1
    _total_remove_time += (Time.get_ticks_usec() / 1000000.0) - start_time

func get_objects_in_area(position: Vector2) -> Array:
    var start_time := Time.get_ticks_usec() / 1000000.0
    var objects: Array = []
    var cell_index = get_cell_index(position)
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            var neighbor = Vector2i(cell_index.x + dx, cell_index.y + dy)
            if grid.has(neighbor):
                objects.append_array(grid[neighbor])
                _total_objects_checked += grid[neighbor].size()

    _total_collision_checks += 1
    _total_collision_time += (Time.get_ticks_usec() / 1000000.0) - start_time
    return objects

func update_object(object: Node2D, old_position: Vector2, new_position: Vector2) -> void:
    var start_time := Time.get_ticks_usec() / 1000000.0
    remove_from_grid(object, old_position)
    add_to_grid(object, new_position)
    _total_update_time += (Time.get_ticks_usec() / 1000000.0) - start_time
```

Fonte: Produzido pelo autor.

Analisando o código fonte da Figura 33, observa-se que a verificação de colisão agora depende exclusivamente do número de objetos nas células adjacentes à posição de interesse, e não do total de objetos na cena. Isso é verificado pela função get_objects_in_area, que utiliza índices da estrutura do grid (-1, 0, 1) para coletar as células em uma determinada área.

O melhor caso ocorre quando os objetos estão uniformemente distribuídos no espaço, resultando em poucos objetos por célula. Nesse cenário, o número de verificações de colisão é significativamente reduzido, mesmo em cenas com muitos objetos, oferecendo vantagens em relação a sistemas que verificam todas as combinações possíveis de objetos ou em casos onde otimizações adicionais ao sistema de física do Godot são necessárias.

O pior caso ocorre quando todos os objetos estão concentrados em uma única célula do grid, exigindo verificações de colisão entre todos os objetos dessa célula, o que pode resultar em desempenho semelhante a um sistema de colisão

não otimizado.

Na sequência, foi implementada uma cena separada apenas para realizar um teste efetivo de comparação das duas implementações. O script que conduz essa cena simula simultaneamente as duas abordagens e produz uma saída no terminal com informações de depuração extraídas diretamente do motor rodando o jogo. A Figura 34 abaixo apresenta o trecho de código relacionado a esta lógica:

Figura 34 - Código de inicialização e registro de performance da comparação com Spatial Partition

```
func _ready() -> void:
    spatial_grid = SpatialGrid.new()
    spatial_grid.enable_performance_logging = true
    add_child(spatial_grid)

    no_grid = NoGridCollision.new()
    no_grid.enable_performance_logging = true
    add_child(no_grid)

    spawn_timer = Timer.new()
    spawn_timer.wait_time = spawn_interval
    spawn_timer.timeout.connect(_on_spawn_timer_timeout)
    add_child(spawn_timer)

    for i in range(max_clusters):
        var angle = (2 * PI * i) / max_clusters
        var center = Vector2(
            cos(angle) * 400.0,
            sin(angle) * 400.0
        )
        cluster_centers.append(center)

    spawn_timer.start()


func _finish_test() -> void:
    if test_complete:
        return

    test_complete = true
    spawn_timer.stop()

    print("\n[TESTE FINALIZADO]")
    print(" ├ Tempo total:          %.1f s" % elapsed_time)
    print(" ├ Objetos criados:      %d" % test_objects.size())
    print(" └ Comparação final:")

    for obj in test_objects:
        if is_instance_valid(obj):
            spatial_grid.remove_from_grid(obj, obj.global_position)
            no_grid.remove_object(obj)
            obj.queue_free()

    test_objects.clear()

    if spatial_grid:
        spatial_grid._log_performance_stats()
    if no_grid:
        no_grid._log_performance_stats()

    queue_free()
```

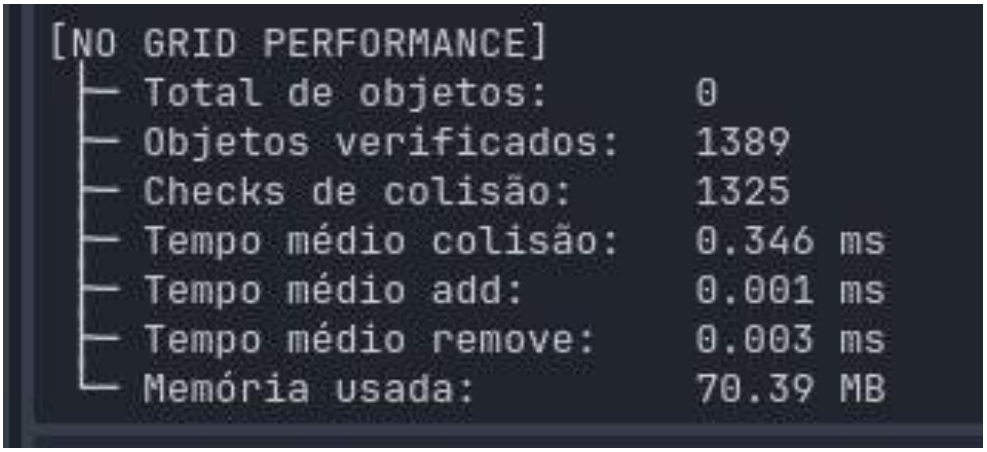
Fonte: Produzido pelo autor.

4.1.4 Análise comparativa do Spatial Partition

Neste tópico serão dispostos resultados comparativos entre o desempenho da aplicação, em termos de consumo de memória e tempo, antes e depois da implementação da técnica.

Na métrica de consumo de tempo foi observada uma redução evidente de tempo após a implementação do Spatial Partition: O tempo médio gasto para a manutenção da colisão antes da utilização da técnica foi de, como destacado na saída do script de teste na figura 35 abaixo, cerca de 0.346 ms (milissegundos).

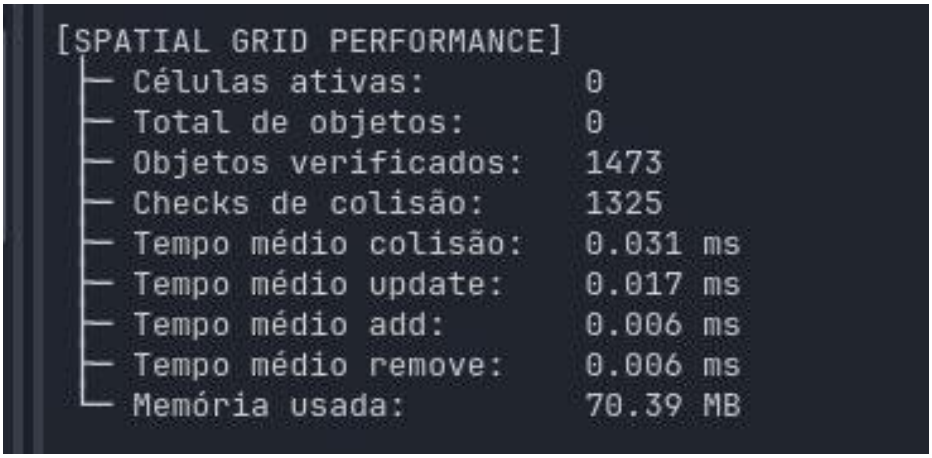
Figura 35 – Captura de tela do consumo de tempo e memória do Platformer2d antes da utilização do Spatial Partition.



Fonte: Produzido pelo autor.

Em comparação, o tempo médio gasto para a manutenção da colisão após a utilização da técnica foi de, como destacado na figura 36 abaixo, cerca de 0.031 ms.

Figura 36 - Captura de tela do consumo de tempo e memória do Platformer2d após a utilização do Spatial Partition.



Fonte: Produzido pelo autor.

Para a métrica de consumo de memória, não foi observado uma alteração significativa. Apesar do uso da estrutura adicional para manter o grid, essa métrica permaneceu estável. Em ambas as implementações, foi registrado um consumo máximo de memória de aproximadamente 70.39MiB no final dos testes como destacado nas figuras 35 e 36.

4.1.5 Análise dos resultados do Spatial Partition

Com os dados obtidos através da execução da aplicação com as implementações antes e após a implementação do Spatial Partitioning, incluindo as comparações de tempo de execução e número de verificações, somos capazes de definir algumas conclusões significativas.

A primeira conclusão é que o uso do grid espacial ocasionou uma redução drástica no tempo de verificação de colisão, o que cumpre as expectativas definidas na fundamentação teórica deste trabalho. O sistema com Spatial Partitioning reduziu o tempo médio de manutenção de colisão em 0,315 ms, uma melhoria significativa em relação à abordagem tradicional. Esta redução se torna ainda mais significativa quando consideramos que o teste foi realizado em pouco mais de 1000 objetos, e a diferença tende a aumentar proporcionalmente com o número de objetos na cena.

A segunda conclusão é que, diferentemente do que poderia ser esperado, o uso do Spatial Partition não resultou em um aumento significativo no consumo de memória. Ambos os sistemas mantiveram um uso de memória similar de 70.39MiB, demonstrando que a otimização foi alcançada sem um sacrifício significativo de memória. Este resultado pode ser explicado pelo fato da implementação feita para este trabalho ter limitado o consumo de memória da estrutura ao número relativamente pequeno de células ativas durante os testes, já que apenas células ativas têm memória alocada para si. Em cenários maiores, com mais objetos e células ativas, seria possível observar um aumento mais perceptível no consumo de memória do grid.

Dessa forma, estes resultados comprovam a eficiência da técnica de Spatial Partitioning em reduzir o tempo de processamento e potencialmente aumentar o desempenho de um jogo, especialmente em cenários com muitos objetos que interagem entre si.

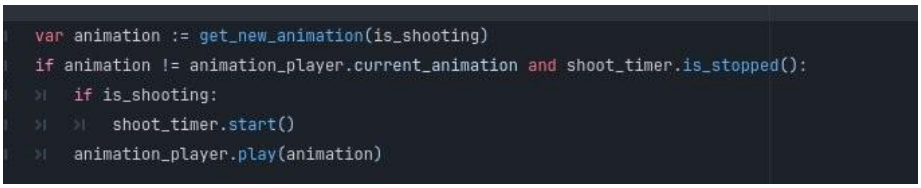
4.2 APLICAÇÃO DA TÉCNICA DIRTY FLAG

O trecho do jogo selecionado para aplicação da técnica de otimização

conhecida como Dirty Flag envolve a lógica de controle de animações do personagem do jogador, aumentando a performance ao limitar cálculos desnecessários.

Na sua forma original, a lógica de controle das animações poderia ser chamada a cada quadro, mesmo quando a animação atual não se alterava. Em jogos com muitos objetos, cálculos que ocorrem desnecessariamente a todo o momento como esse podem causar problemas de performance, cabendo assim uma otimização com o uso do Dirty Flag. Veja abaixo, na imagem 37, o código que representa a lógica descrita:

Figura 37 - Código da lógica de controle das animações do jogador no Platformer2d



Fonte: Produzido pelo autor.

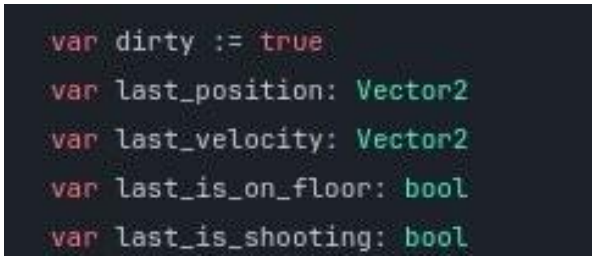
Na figura 37, a animação do personagem é determinada pela função get_new_animation e armazenada na variável “animation”. A troca da animação do personagem só ocorre se duas condições forem atendidas: a animação calculada for diferente da animação atual, o temporizador de tiro “shoot_timer” estiver parado. Se essas condições forem satisfeitas, a animação é trocada e, caso o personagem esteja atirando, o temporizador de tiro é iniciado. O temporizador “shoot_timer” é utilizado aqui para garantir que a animação de disparo, que é uma ação acionada pelo jogador, não toque várias vezes enquanto o jogador já está atirando.

Como descrito anteriormente neste estudo, a implementação do Dirty Flag se dá por meio de variáveis de controle utilizadas para verificar o estado atual de objetos, monitorando-os para que cálculos e chamadas de funções só ocorram quando essas propriedades forem efetivamente alteradas. Com a inserção dessa técnica, espera-se que a lógica descrita acima seja executada apenas quando o estado do jogador se modificar, reduzindo o número de processamentos desnecessários por quadro e, consequentemente, melhorando a performance geral do jogo.

A primeira etapa da implementação do Dirty Flag consistiu na criação de variáveis que armazenam o último estado relevante do jogador: sua posição, velocidade, estado de contato com o chão e se ele está atirando. A cada ciclo de lógica do personagem, esses valores são comparados com o estado atual. Caso qualquer uma dessas variáveis tenha sofrido alteração, a variável dirty é ativada. A

figura 38 abaixo destaca a inicialização das variáveis de controle.

Figura 38 - Código de inicialização das variáveis de controle do Dirty Flag no Platformer2d

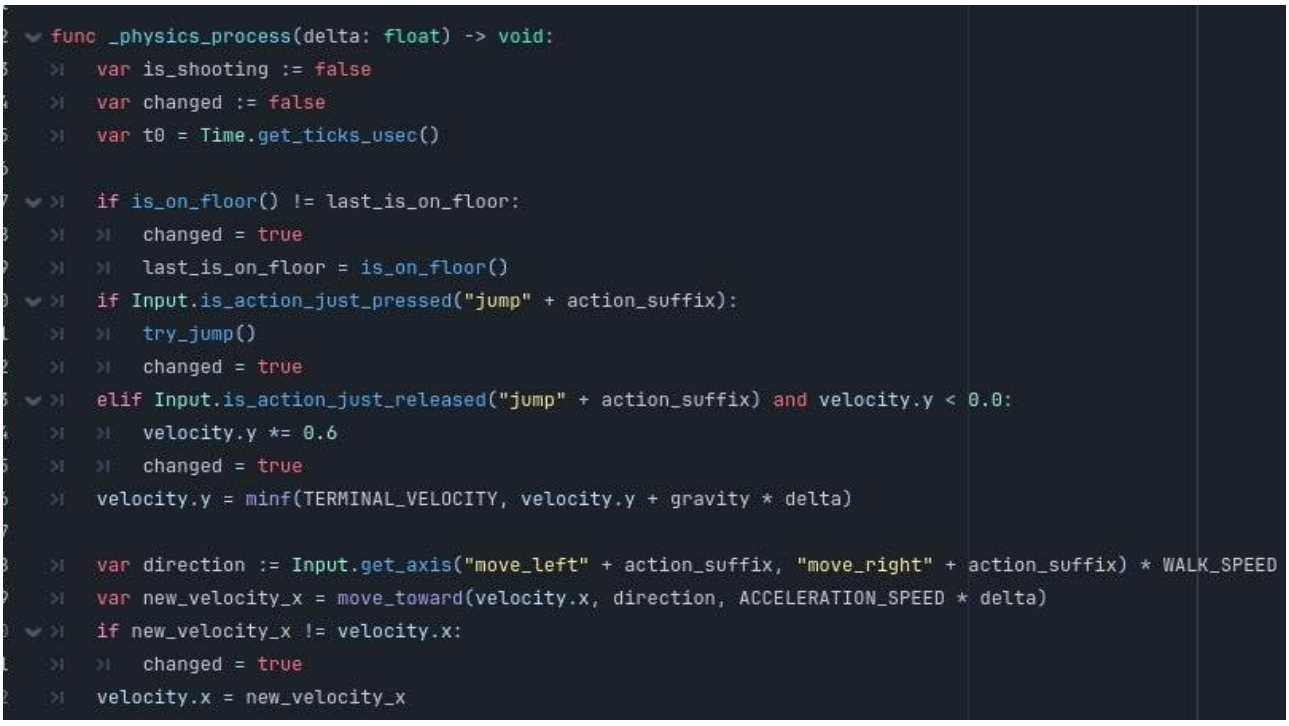


```
var dirty := true
var last_position: Vector2
var last_velocity: Vector2
var last_is_on_floor: bool
var last_is_shooting: bool
```

Fonte: Produzido pelo autor.

Ademais, a variável changed foi introduzida como uma flag auxiliar na função _physics_process, sendo ativada quando ocorrem alterações específicas, como mudanças no estado de contato com o chão, ações de pulo, soltura do botão de pulo com redução de velocidade vertical, alterações na velocidade horizontal ou disparos. Em conjunto com as outras variáveis de controle já citadas, changed é avaliada na verificação final para determinar se a variável dirty deve ser ativada. Veja, na figura 39 abaixo, como a variável “changed” foi implementada de acordo com o exposto:

Figura 39 - Código de implementação da variável changed de controle do Dirty Flag no Platformer2d

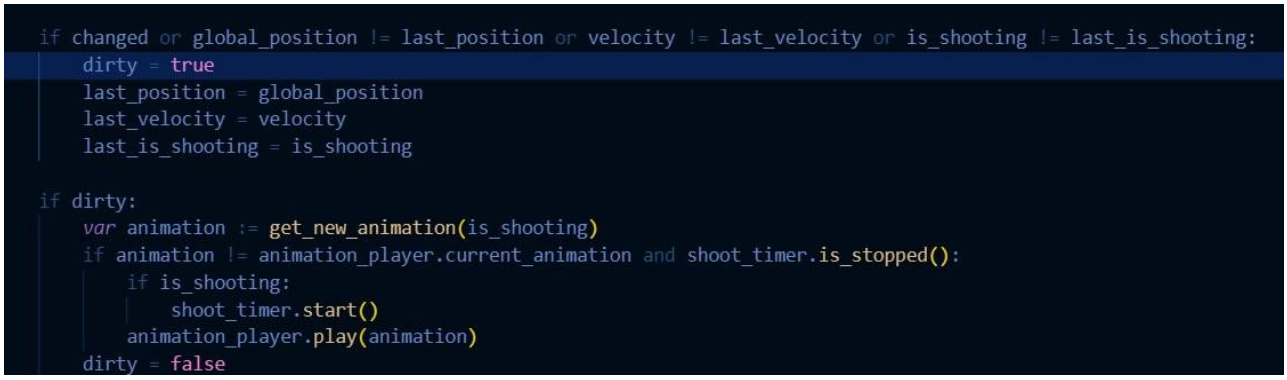


```
func _physics_process(delta: float) -> void:
    var is_shooting := false
    var changed := false
    var t0 = Time.get_ticks_usec()

    if is_on_floor() != last_is_on_floor:
        changed = true
        last_is_on_floor = is_on_floor()
    if Input.is_action_just_pressed("jump" + action_suffix):
        try_jump()
        changed = true
    elif Input.is_action_just_released("jump" + action_suffix) and velocity.y < 0.0:
        velocity.y *= 0.6
        changed = true
    velocity.y = minf(TERMINAL_VELOCITY, velocity.y + gravity * delta)

    var direction := Input.get_axis("move_left" + action_suffix, "move_right" + action_suffix) * WALK_SPEED
    var new_velocity_x = move_toward(velocity.x, direction, ACCELERATION_SPEED * delta)
    if new_velocity_x != velocity.x:
        changed = true
    velocity.x = new_velocity_x
```

Fonte: Produzido pelo autor.

Em seguida, para a segunda etapa da implementação, essas variáveis foram utilizadas *para* monitorar o objeto jogador, onde a verificação da troca de animações ocorreria somente quando algo fosse alterado em sua posição, velocidade ou ações. Na figura 40 abaixo, é possível visualizar o código a lógica alterada com o uso do Dirty Flag:

Figura 40 - Código de implementação do Dirty Flag no controle de animações de jogador no Platformer2d



```
if changed or global_position != last_position or velocity != last_velocity or is_shooting != last_is_shooting:
    dirty = true
    last_position = global_position
    last_velocity = velocity
    last_is_shooting = is_shooting

if dirty:
    var animation := get_new_animation(is_shooting)
    if animation != animation_player.current_animation and shoot_timer.is_stopped():
        if is_shooting:
            shoot_timer.start()
        animation_player.play(animation)
    dirty = false
```

Fonte: Produzido pelo autor.

Analisando o código-fonte apresentado na Figura 40, observa-se que a lógica de animação do personagem jogador passou a depender exclusivamente de mudanças em variáveis de estado. Caso nenhuma mudança seja detectada em relação ao estado anterior, a lógica de atualização de animação é completamente ignorada naquele quadro, evitando chamadas desnecessárias à função de controle das animações.

Nessa implementação, o melhor cenário de desempenho ocorre quando o jogador permanece em um estado estático ou com mudanças mínimas de estado ao longo de vários quadros consecutivos. Nesses casos, a variável dirty permanece desativada, e nenhuma lógica de animação é processada, resultando em economia de tempo significativa.

O pior caso ocorre quando o jogador está em constante alteração de estado, como em saltos contínuos, movimentos constantes de direção e disparos frequentes. Nessa situação, a variável dirty é ativada constantemente, e a lógica de verificação e atualização de animação será executada a cada quadro, alcançando um desempenho semelhante à implementação tradicional.

Figura 41 - Código de inicialização e registro de performance da comparação com Dirty Flag

Fonte: Produzido pelo autor.

```
func _ready():
    var metade = int(NUM_PLAYERS / 2)

    for i in range(metade):
        var player = PLAYER_SCENE.instantiate()
        add_child(player)
        players.append(player)

    for i in range(NUM_PLAYERS - metade):
        var dplayer = PLAYER_SCENE.instantiate()
        dplayer.set_script(DIRTY_PLAYER_SCENE)
        add_child(dplayer)
        dirty_players.append(dplayer)
    set_process(true)

func _process(delta):
    if not running:
        return
    elapsed_time += delta
    _simulate_players(players)
    _simulate_players(dirty_players)
    if elapsed_time >= TEST_DURATION and not log_final_ja_foi:
        running = false
        _log_final()
        print("Teste concluído!")
        log_final_ja_foi = true

        for p in players:
            p.enable_performance_logging = false
        for p in dirty_players:
            p.enable_performance_logging = false
```

Na figura 41 acima, o script de teste instancia jogadores em uma cena, com metade deles implementando o Dirty Flag, enquanto a outra metade implementa a troca de animações original. O teste então roda por um determinado tempo e faz o registro das métricas de tempo e memória da aplicação.

4.2.1 Análise comparativa do Dirty Flag

Neste tópico serão dispostos resultados comparativos entre o desempenho da aplicação, em termos de consumo de memória e tempo, antes e depois da implementação da técnica.

Na métrica de consumo de tempo foi observada uma redução de tempo após a implementação do Dirty Flag: O tempo médio gasto para o controle das animações de jogador, como destacado na saída do script de teste na figura 42 abaixo, foi de cerca de 0.052439 ms.

Figura 42 – Captura de tela de consumo de tempo e memória do Platformer2d antes da utilização do Dirty Flag.

Fonte: Produzido pelo autor.

Em comparação, O tempo médio gasto para essa mesma função após a utilização da técnica foi de, como destacado na figura 43 abaixo, cerca de 0.046788 ms.

Figura 43 - Captura de tela do consumo de tempo e memória do Platformer2d após a utilização do Dirty Flag.

Fonte: Produzido pelo autor.

Para a métrica de consumo de memória, não foi observado uma alteração significativa. Em ambas as implementações, foi registrado um consumo máximo de memória de aproximadamente 53.44MiB no final dos testes como destacado nas figuras 42 e 43.

4.2.2 Análise dos resultados do Dirty Flag

Com os dados obtidos através da execução da aplicação com as implementações antes e após a implementação do Dirty Flag, incluindo as

comparações de tempo de execução e número de verificações, somos capazes de definir algumas conclusões significativas.

A primeira conclusão é que o uso do Dirty Flag ocasionou uma redução de tempo de verificação de colisão, o que cumpre as expectativas definidas na fundamentação teórica deste trabalho. O sistema com Dirty Flag reduziu o tempo médio de controle das animações do jogador em 0.005561 ms, uma melhoria que parece pequena em relação à abordagem tradicional, porém é importante destacar a escalabilidade da técnica, onde quanto mais tempo e mais cálculos fossem envolvidos, mais esta seria efetiva em sua função.

A segunda conclusão é que o uso do Dirty Flag não resultou em um aumento significativo no consumo de memória. Ambos os sistemas mantiveram um uso de memória similar de 53.44 MiB, demonstrando que a otimização foi alcançada sem um sacrifício significativo de memória. Este resultado pode ser explicado pelo fato de o jogador não utilizar dados derivados no seu controle de animações. Como explorado anteriormente na fundamentação teórica do Dirty Flag, dados derivados são informações sobre o estado de um objeto que são calculados com base em outros dados, chamados de dados primários. Com nenhum dado derivado presente no objeto do jogador, não é necessário armazenar estados para prevenir inconsistências, permitindo que a técnica seja implementada neste caso específico sem perdas significativas de memória.

Dessa forma, estes resultados comprovam a eficiência da técnica de Dirty Flag em reduzir o tempo de processamento e potencialmente aumentar o desempenho de um jogo, especialmente em cenários com muitos cálculos que não precisam ser necessariamente realizados a cada quadro.

5 CONCLUSÕES

O presente estudo teve como objetivo expor a importância de implementar otimizações em jogos para que seu desempenho seja satisfatório para o maior número possível de usuários. Apontou também para o fato de que a escolha de qual (ou quais) técnica(s) de otimização utilizar depende das particularidades do jogo e devem ser realizadas somente após análise prévia do balanço de recursos computacionais disponíveis para que o aplicativo possa rodar em uma variedade de dispositivos. Uma outra situação importante abordada neste trabalho foi a percepção do usuário sobre a queda de performance em um jogo, e como isso afeta a sua experiência com ele.

Os resultados obtidos demonstraram que a aplicação de técnicas escaláveis de otimização de código, tais como as exploradas neste estudo, se aplicadas em um cenário apropriado, são capazes de reduzir de forma significativa o consumo de recursos, mantendo a jogabilidade intacta para o usuário final.

Com estes resultados, espera-se que seja fomentada a prática de otimização responsável em jogos, especialmente aqueles que demandam mais dos dispositivos do usuário, para que cada vez mais pessoas possam experienciar esses produtos e a indústria seja fomentada de forma benéfica para todos. Para estudos futuros, sugere-se a investigação das técnicas de otimização exploradas em aplicações mais complexas, como jogos 3D ou multiplayer.

REFERÊNCIAS

CHINNEK, John. **Practical Optimization: a Gentle Introduction**, 2020. Disponível em: https://www.optimization101.org/2020/12/welcome.html. Acesso em: 8 dez 2024.

PREISZ, Eric. **Video Game Optimization**.
Boston: Course Technology, 2011.

YSTROM, Robert. **Video Game Optimization**, 2014. Disponível em: https://gameprogrammingpatterns.com/contents.html. Acesso em: 8 dez 2024.

GODOT. **Documentação da Godot Engine 4.3**, 2014. Disponível em: https://docs.godotengine.org/pt-br/4.x/index.html. Acesso em: 13 dez. 2024

GODOT. **Overview of Godot's key concepts**, 2014. Disponível em: https://docs.godotengine.org/en/stable/getting_started/introduction/key_concepts_overview.html. Acesso em: 13 dez 2024.

HEINEMAN, Becky. **Sponsored Feature: Common Performance Issues in Game Programming**, 2008. Disponível em: https://www.gamedeveloper.com/programming/sponsored-feature-common-performance-issues-in-game-programming. Acesso em 8 dez 2024.

GAME DEVELOPER. **What is game programming and how do you become a game programmer?**, 2024. Disponível em: https://www.gamedeveloper.com/programming/what-is-game-programming-and-how-do-you-become-a-game-programmer-. Acesso em: 8 dez 2024.

Disponível em: https://www.gamedeveloper.com/programming/what-is-game-programming-and-how-do-you-become-a-game-programmer-. Acesso em: 8 dez. 2024.

DEVMAKING. **Object Pool**, 2021. Disponível em: https://www.devmaking.com/learn/design-patterns/object-pool-pattern/ Acesso em: 8 dez. 2024.

STANFORD UNIVERSITY. Optimization, 2015. Disponível em: https://web.stanford.edu/group/sisl/k12/optimization/MO-unit1-pdfs/1.1optimization.pdf Acesso em: 6 jul. 2025.