UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

HUMBERTO SAGAVE LENTZ

Desenvolvimento de um jogo de luta 2D com *rollback netcode*

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof[a]. Dr[a]. Renata Galante

Porto Alegre
2022

“Shut your eyes and see.”

— JAMES JOYCE

RESUMO

O objetivo deste trabalho é apresentar o desenvolvimento de um jogo digital online (em grande parte autoral) 2D do gênero de luta que visa tratar principalmente de problemas de sincronização. Jogos de luta são conhecidos pelas reações e reflexos necessários por parte de seus jogadores e, no contexto de jogos onlines, a latência de rede impõe alguns desafios para manter uma experiência agradável para os usuários. O design geral do jogo também pode ser influenciado pela decisão de estabelecer uma experiência online. As motivações, ferramentas e outras decisões que foram tomadas durante todo o processo são demonstradas e discutidas. Entre alguns dos tópicos estão a escolha da engine, o desenvolvimento de um protótipo e a criação dos visuais, animações, efeitos sonoros e músicas. Testes para avaliar o funcionamento do jogo em alguns cenários de rede são realizados e analisados. Resultados mostram uma aplicação com o impacto da latência reduzido que poderá servir como um produto viável mínimo ou *minimum viable product*.

Palavras-chave: Desenvolvimento. jogo de luta. online. design.

Development of a 2D fighting game with rollback netcode

ABSTRACT

The goal of this work is to present the development of an online 2D fighting game where its main feature is to handle synchronization issues. Fighting games are known for the reactions and reflexes needed from their players and, within the context of online games, latency brings some challenges when trying to keep the experience smooth for the users. The overall design of the game can be influenced by the decision of making an online experience. The tools that were used and the decisions that were made during the process are shown and discussed. Some of the topics include the game engine, development of a prototype, visuals and animations, sound effects and music. Tests to assess the game's quality on some network scenarios were made and reviewed. Results show an application with reduced latency impact with enough features to be a minimum viable product.

Keywords: Development. fighting game. online. design.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

P2P	Peer-to-Peer
MIT	Massachusetts Institute of Technology
GGPO	Good Game, Peace Out
FM	Frequency Modulation
CRT	Cathode Ray Tube
NAT	Network Address Translation
API	Application Programming Interface
UDP	User Datagram Protocol
MVP	Minimum Viable Product

SUMÁRIO

1 INTRODUÇÃO

Na década de 1990, jogos como Mortal Kombat e Street Fighter obtiveram grande sucesso ao ponto de serem considerados como progenitores do gênero de luta (WOOLUMS, 2017). Os jogos apresentavam o objetivo simples de reduzir o indicador de vida do oponente dentro de um intervalo de tempo usando variados chutes, socos e ataques especiais e ainda lidar com as reações do mesmo oponente. A internet não estava tão presente ainda nesse momento e a forma de jogadores se reunirem e jogarem era por meio dos *arcades* e consoles. Com o avanço da internet a possibilidade de jogos a distância se tornou factível, porém desafios de implementação que envolvem a latência de rede (o tempo necessário para um pacote de dados ser capturado, transmitido, processado por dispositivos no caminho, recebido e por fim decodificado) se tornaram evidentes.

Se anteriormente um jogo entre dois usuários não sofria de problemas técnicos de rede, agora depende de toda uma infraestrutura que não está sob controle nem do desenvolvedor da aplicação. O atraso da rede afeta o input dos jogadores e é preciso soluções para manter a experiência. Mesmo com comunicações *peer-to-peer* que evitam ter algum servidor intermediário e assim auxiliam reduzindo a latência, a comunicação ainda vai sofrer de problemas de sincronização por outros motivos como *clock skew*.

O objetivo deste trabalho é analisar as soluções existentes principalmente no contexto de rede, estabelecer o design geral de um jogo de luta 2D e realizar toda a implementação com as ferramentas adequadas.

É analisada a solução clássica *delay-based* e apresentados seus efeitos. Em seguida, uma solução que faz uso de *rollbacks* é discutida. A solução utilizada (uma combinação das soluções anteriores) e que está sob licença MIT desde 2019 se trata de uma biblioteca chamada Good Game, Peace Out. A mesma impõe requisitos para o design e implementação com o intuito de amenizar o impacto das dificuldades de rede descritas. Com a pandemia de COVID-19, muitas soluções de *rollback* estão sendo consolidadas justamente por causa do distanciamento (LAWSON, 2021).

Toda a motivação e argumentação por trás de decisões tomadas são apresentadas e incluem a implementação de um protótipo da movimentação de um personagem na *engine* escolhida, os *trade-offs* por usar tal *engine*, a criação de *sprite sheets* (arquivo com sequências de imagens que contém os quadros de uma animação) em um estilo *pixel art* utilizando uma ferramenta de edição de imagens, a criação dos efeitos sonoros e músicas utilizando uma ferramenta *tracker* de produção de música.

O restante do texto está organizado da seguinte forma. O **Capítulo 1** corresponde ao capítulo atual e apresenta uma breve contextualização e motivação que guiaram o desenvolvimento. O **Capítulo 2** descreve em mais detalhes as tecnologias usadas e suas características, além de apresentar alguns trabalhos relacionados. O **Capítulo 3** trata de uma explicação sobre *netcodes* (termo genérico que geralmente é associado aos problemas de redes e sincronização) e de uma análise sobre GGPO. O **Capítulo 4** descreve todo o desenvolvimento do projeto com as tecnologias e ferramentas adotadas partindo de um pequeno protótipo. O **Capítulo 5** apresenta os testes realizados após desenvolvimento que ajudaram a avaliar a qualidade esperada da aplicação e uma pequena discussão sobre a adaptação do código fonte realizada para uma loja digital de jogos. O **Capítulo 6** apresenta as conclusões do trabalho e alguns pontos para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo descreve as tecnologias usadas: o que são, suas principais características e como foram utilizadas no contexto do trabalho proposto. Ainda são apresentados de maneira breve algumas aplicações, tecnologias e trabalhos relacionados.

2.1 Tecnologias usadas

2.1.1 Good Game, Peace Out

Good Game, Peace Out ou GGPO é uma biblioteca desenvolvida por Tony Cannon lançada em 2006 para testar o conceito de *rollback* com jogos executando em emuladores. Por volta de 2019, Cannon lançou uma versão da biblioteca que está sob licença MIT e que ajuda a incorporar a estratégia em muitos jogos em desenvolvimento. A biblioteca foi desenvolvida com código C e C++ e apresenta mecanismos para mascarar a latência por meio de previsão de inputs e execução especulativa (GGPO. . . , 2022). É de extrema relevância para um jogo online e de ação onde a experiência do usuário é prioridade (como o proposto pelo trabalho).

2.1.2 Godot

Para o desenvolvimento em geral do jogo foi escolhida a *engine* Godot. Trata-se de uma *engine open source* com recursos ideais para jogos 2D, apresentando um sistema de cenas flexível e hierarquia de *nodes*. A possibilidade de incorporar módulos de código C++ permitiu adicionar a tecnologia GGPO já citada (arquivos auxiliares estão no repositório Godot GGPO[1]). Por ser leve se comparada a outras opções existentes, com um fluxo de trabalho sem muitos obstáculos (incluindo uma linguagem de script similar a Python que ajuda muito a implementar protótipos e testar ideias de forma rápida) e um *node* de animação poderoso que permite animar qualquer propriedade de outro *node* (AnimationPlayer), a *engine* foi escolhida (GODOT. . . , 2022).

[1] <https://github.com/FlutterTal/godot_ggpo>

2.1.3 GIMP

É um editor de imagens *open source* que fornece ferramentas para criar e manipular imagens que sejam *bitmaps* ou vetoriais (GIMP..., 2022). É frequentemente tomado como alternativa ao programa Photoshop que é pago. No contexto do trabalho, todo conteúdo visual foi desenvolvido com a ferramenta.

2.1.4 DefleMask

É um *tracker* de produção de música para vários chips de som antigos. É possível estabelecer instrumentos modificando parâmetros de cada operador e envelopes conforme síntese de modulação de frequência ou *frequency modulation* e inserir notas nos canais disponíveis (DEFLEMASK..., 2022). A ferramenta foi utilizada para a criação das músicas e todos os efeitos sonoros presentes no jogo proposto. Em especial, foi escolhida a opção que replica o chip de som de síntese FM Yamaha YM2612 que fazia parte de modelos do Mega Drive da Sega, por exemplo (WIKIPEDIA, 2022).

2.1.5 Clumsy

É uma ferramenta de rede que permite interferir diretamente nas portas do sistema sem outros requisitos adicionais, podendo simular alguns efeitos na rede como a latência, perda e duplicação de pacotes, entre outros (CLUMSY..., 2022). A ferramenta foi usada na etapa de testes do trabalho e serve como opção intermediária entre a funcionalidade de teste disponibilizada por GGPO e um cenário de rede real.

2.1.6 Steamworks

É um conjunto de ferramentas disponibilizado pela loja de jogos digitais Steam para desenvolvedores associados. Inclui uma aplicação para upload de *builds* de jogos e também uma *application programming interface* para interação com recursos da própria Steam como comunicação P2P (STEAMWORKS..., 2022). Para o trabalho, foi importante na etapa de testes em máquinas diferentes, pois ajuda a evitar configurações extras e ainda fornece segurança para o usuário.

2.2 Tecnologias relacionadas

2.2.1 Steam Remote Play

Uma alternativa ao uso de GGPO com a API de comunicação P2P da Steam. Funciona através do *streaming* de vídeo do jogo de uma máquina principal para até 4 ou mais jogadores que poderão jogar de qualquer dispositivo e sistema operacional sem precisar comprar ou ter uma cópia do jogo. No entanto, é necessário fazer uso da aplicação Steam Link (STEAM. . . , 2022). A tecnologia parece ser ideal para jogos com *multiplayer* local sem qualquer *feature* online. Os problemas usuais de latência são somados aos problemas adicionais associados ao *streaming* do jogo.

2.2.2 Parsec

Uma outra alternativa ao uso de GGPO com a API de comunicação P2P da Steam. Similar à alternativa anterior, porém faz uso de um protocolo proprietário e ainda se estende a uma aplicação *remote desktop* (CONNECT. . . , 2022).

2.2.3 Serviços na nuvem

Serviços na nuvem para jogos, como as tecnologias anteriores, fazem uso do *streaming* de vídeo. O diferencial é que os recursos para a execução do jogo (como servidores) são disponibilizados por demanda. Alguns exemplos são: Google Stadia, Xbox Cloud Gaming e PlayStation Now. Alguns desses serviços vendem ou oferecem uma biblioteca própria de jogos e isso implica que é necessário uma licença para publicar um jogo (similar a Steam). Outros desses serviços, como o Nvidia GeForce Now, permitem que o usuário se conecte com suas contas de lojas digitais de jogos (como a Steam) e jogue (se o jogo for compatível) usando os recursos na nuvem.

2.3 Trabalhos relacionados

Alguns jogos do mesmo gênero que fazem uso da GGPO e outros também do mesmo gênero que apresentam uma implementação própria de *rollback netcode* serão

apresentados. Skullgirls lançado em 2012 teve como referência para sua *engine* e jogabilidade o jogo de luta Marvel vs. Capcom 2. O jogo é famoso por sua arte visual com gráficos ilustrados e animados à mão (SKULLGIRLS..., 2022). Pocket Rumble lançado em 2016 apresenta gráficos no estilo *pixel art* e busca ser acessível fazendo uso de apenas dois botões de ação (POCKET..., 2022). Them's Fightin' Herds lançado em 2018 apresenta gráficos cartunizados e alguns anos atrás era um *fangame* da série animada My Little Pony (THEM'S..., 2022).

Alguns dos jogos que apresentam uma implementação própria de *rollback netcode* incluem Killer Instinct lançado em 2013 (KILLER..., 2022) e Mortal Kombat 11 lançado em 2019 (MORTAL..., 2022). Para traçar uma comparação com o jogo proposto pelo trabalho, a jogabilidade do mesmo tem como referência a trilogia dos primeiros jogos da série Mortal Kombat que inclui 5 botões de ação onde todos os personagens apresentam um conjunto básico e idêntico de movimentos junto de no mínimo 3 movimentos especiais únicos.

3 PROPOSTA DE INCORPORAÇÃO DE *ROLLBACK NETCODE* NUM JOGO DE LUTA 2D

O objetivo deste capítulo é apresentar alguns conceitos básicos sobre jogos e redes, detalhar duas das soluções de rede existentes, discutir os requisitos da biblioteca GGPO e explicar brevemente o seu funcionamento também.

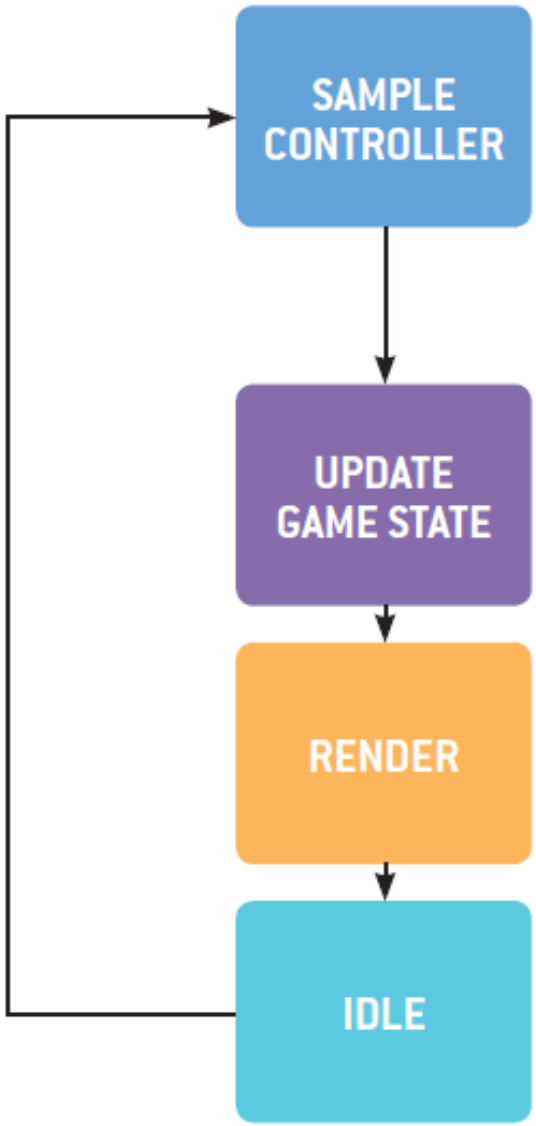
3.1 Visão geral e conceitos básicos

Para acrescentar mais detalhes aos problemas já enunciados e entrar em pontos mais específicos é preciso definir alguns termos. Jogos digitais possuem um laço ou loop de execução (Figura 3.1) que numa forma mais simples é composto das etapas de coleta de input do jogador (algum botão é pressionado, algum movimento direcional, entre outros), seguido da atualização do estado do jogo, seguido da etapa de *rendering* que dispõe todos os elementos visuais para o jogador ter noção do novo estado do jogo, interpretá-lo e seguir repetindo este mesmo loop.

Em jogos de luta, o tempo é geralmente medido por uma unidade chamada *frame* e é possível assumir que todos os jogos possuem uma atualização do estado 60 *frames* por segundo ou aproximadamente a cada 16 milissegundos em tempo real. Ao jogar localmente em um *arcade* ou console e quando um jogador gera algum input, este input é processado quase que imediatamente (desconsiderando pequenos atrasos de hardware). A situação é diferente com a internet, pois duas máquinas estão querendo se comunicar e o tempo necessário para uma mensagem chegar de uma máquina até a outra pode variar. Outro ponto importante é que agora ambas as máquinas estão executando uma cópia do jogo e querendo manter os estados das cópias idênticos. Isto implica que um determinismo deve existir, ou seja, dado o input de um jogador para cada cópia como mensagem, ambas cópias devem gerar estados idênticos ao atualizarem para um novo estado.

O determinismo aliado a um mecanismo de *lockstep* (uma cópia do jogo avança apenas quando possui os inputs necessários para o avanço) e também aliado a uma troca de mensagens contendo somente o input dos jogadores forma um modelo popular (SUN, 2019). Lembrando que a implementação do trabalho fez uso da GGPO, nas próximas seções são apresentados conceitos que, incrementalmente, se aproximam do conceito geral dessa tecnologia.

Figura 3.1: Loop típico de um jogo digital



Fonte: (CANNON, 2012)

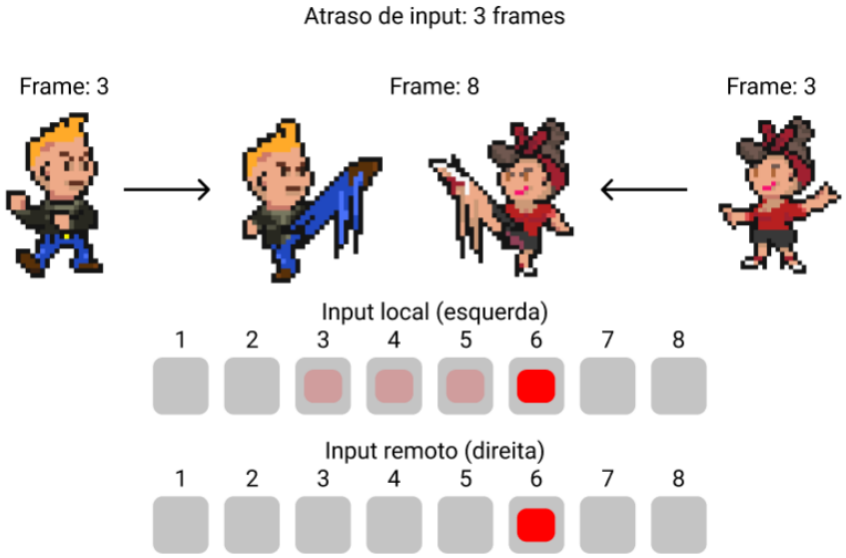
3.2 *Delay-based netcode*

Uma solução clássica que ainda é usada é chamada de *delay-based* ou *input delay*. Se o input de um jogador mostra um atraso para chegar à máquina remota então esta estratégia irá atrasar artificialmente o input do jogador local pelo mesmo intervalo de tempo.

Na Figura 3.2, assumindo que os mesmos inputs são gerados no *frame* 3 com animações idênticas, é possível ver como o atraso artificial permite que um input remoto seja executado junto do input local.

Um efeito desagradável da solução é que, quando a latência de rede variar de

Figura 3.2: Atraso artificial para input local



Fonte: autor

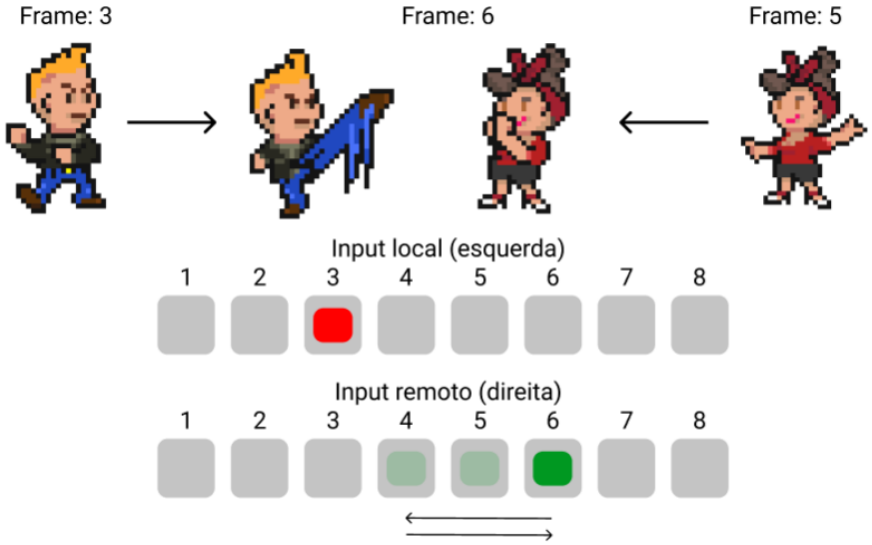
maneira que o tempo necessário para enviar uma mensagem aumente com relação ao estimado, a cópia do jogo não poderá avançar e irá simplesmente parar e esperar pelo input e inflar desnecessariamente a estimativa do intervalo de tempo em alguns casos (picos de latência). Outros efeitos incluem: a distância entre jogadores ainda pode afetar muito já que o atraso artificial deverá sempre ser maior e o fato de que ambas as cópias do jogo irão tratar as dificuldades de rede de maneira idêntica não permitindo maneiras de mascarar a latência (PUSCH, 2019). A vantagem da estratégia está muitas vezes apenas na simplicidade e no custo baixo de implementação.

3.3 *Rollback netcode*

Considerando a solução anterior, uma estratégia de *rollback* funciona da seguinte maneira (Figura 3.3): ao invés de parar o jogo quando um input remoto demora muito, a cópia do jogo continua executando normalmente. Quando o input finalmente chega *frames* depois, ocorre um *rollback* para o estado prévio divergente da simulação e há uma correção dos inputs. Ainda no tempo de um único *frame*, o jogo é atualizado o número de vezes necessário para um novo estado atual. Vale lembrar que o input local é processado imediatamente e isso permite que um jogador possua uma experiência consistente se comparado com a solução anterior.

Um erro visual da estratégia é que animações do jogador remoto podem pular e

Figura 3.3: Um simples *rollback*



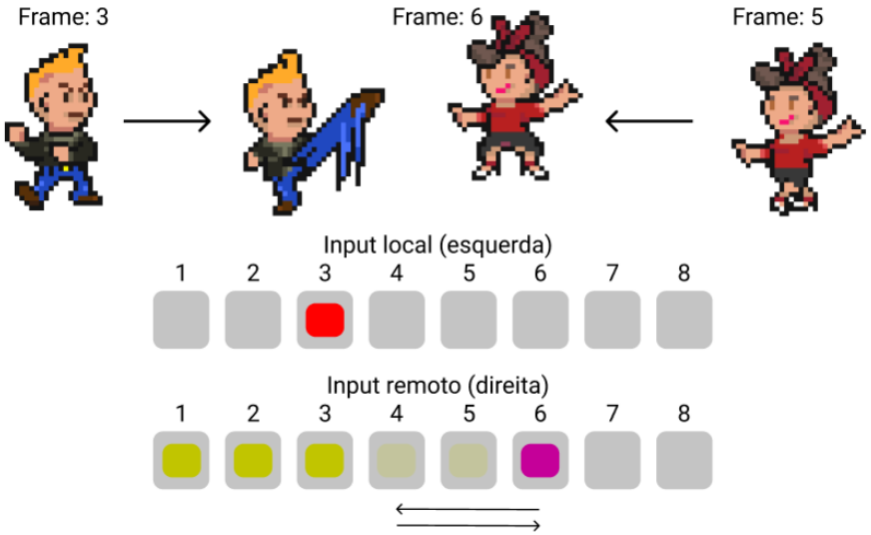
Fonte: autor

começar *frames* depois do seu ponto de início. *Rollbacks* quebram o modelo de *lockstep* temporariamente permitindo que uma cópia mostre um estado diferente para o jogador, porém, eventualmente com um *rollback*, a cópia é corrigida e assume um estado correto (PUSCH, 2019). A Figura 3.3 ilustra um simples *rollback* onde o jogador local inicia um golpe no *frame* 3 e o remoto inicia um bloqueio no *frame* 4. Com o atraso, apenas no *frame* 6 é possível verificar que o jogador remoto iniciou a ação no *frame* 4 e é nesse instante que ocorre um *rollback* e a animação de bloqueio é iniciada e sofre um pulo de 2 *frames* (junto da correção de todo o estado lógico do jogo).

A estratégia ainda pode ser aperfeiçoada com um sistema ingênuo de previsão dos inputs remotos. O jogo atualiza 60 *frames* por segundo, porém se assumirmos que um jogador dentro de um intervalo de 1 segundo pode gerar, num cenário extremo, até 5 inputs, então, a probabilidade da próxima ação a ser executada ser idêntica à última ação executada é relativamente alta.

A Figura 3.4 ilustra um simples *rollback* com o sistema de previsão onde o jogador remoto está realizando uma ação de caminhar e no *frame* 4 realiza uma ação de pulo. É possível ver que no *frame* 5 ele continua caminhando enquanto que no *frame* 6 o estado do jogo assume o estado correto por causa de um *rollback* devido a uma previsão errada. Assim, quando o input remoto chega e se a previsão estiver errada ocorre um *rollback* já descrito previamente. Quando a previsão estiver correta, a estratégia permite um jogo consistente mesmo na presença de problemas na rede, mascarando efeitos de uma comunicação ruim em muitos casos.

Figura 3.4: *Rollback* com o sistema de previsão



Fonte: autor

A estratégia, no entanto, pode ser aperfeiçoada novamente. É possível imaginar que pode ocorrer *rollbacks* constantemente já que o atraso de rede existe. Nesse sentido, é possível combinar a solução de *rollback* com a *delay-based* estabelecendo um *input delay* fixo (em número de *frames* e geralmente pequeno) e deixar para as situações em que as variações de rede são maiores para os *rollbacks*.

A vantagem desta estratégia é que frequentemente os pulos de animações, no contexto de experiência do usuário, são desejados ao invés de simplesmente adicionar um atraso e arruinar a responsividade de um input local. A latência muitas vezes pode ser tão pequena que o efeito não chega a ser percebido também. O sistema de previsão de inputs também pode ser aperfeiçoado mas, na prática, incorporar a possibilidade de uma previsão errada ao design do jogo para minimizar efeitos de *rollbacks* pode ser uma solução melhor (CANNON, 2012) e é a abordagem escolhida para o trabalho em alguns casos.

3.4 Requisitos para *rollback netcode* e GGPO

Alguns dos requisitos para uma implementação com *rollback*, além de uma simulação determinística, são a funcionalidade de salvar e carregar diferentes estados do jogo (serialização) e avançar com as atualizações por demanda idealmente de uma maneira otimizada (um número extra de 4 a 5 atualizações por *frame* já ajuda a mascarar 80 milissegundos de latência) (CANNON, 2012). Este último implica que a lógica do jogo deve ser isolada de tudo e qualquer outro sistema no loop e, dependendo da escolha da *engine*

e dos sistemas, isso pode não ser trivial.

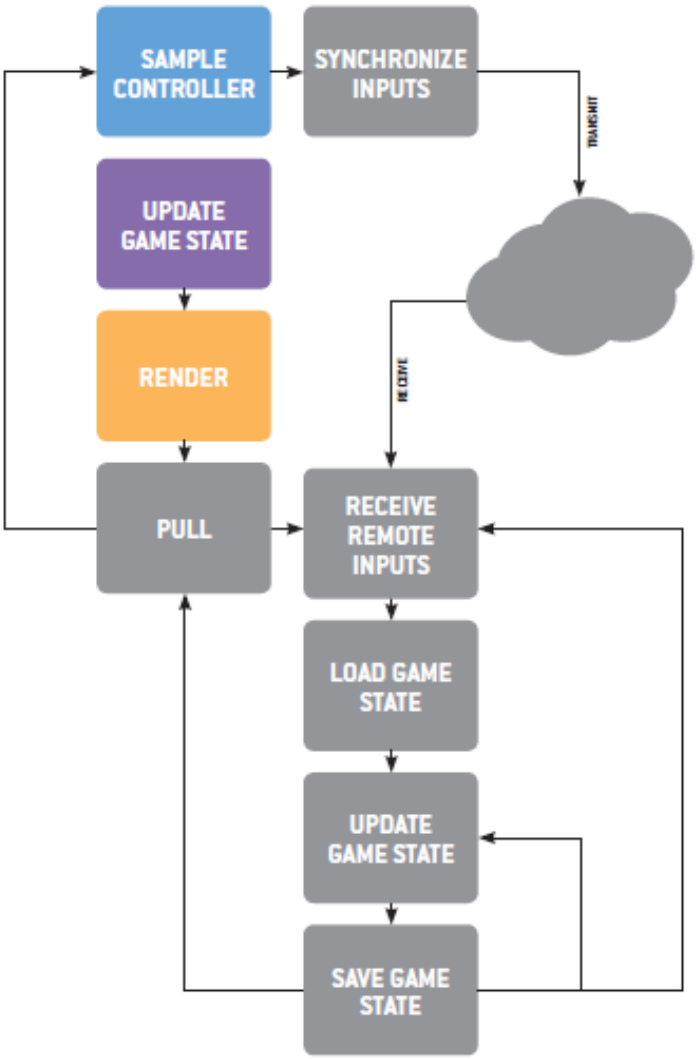
Outra questão é que a maioria das funcionalidades relevantes não chegam a ser executadas quando testando somente com inputs locais. Isto implica que testes em um cenário de rede são de muita importância para avaliar a aplicação. Com GGPO, inicialmente é possível usar um modo especial chamado de *sync test* com o propósito de tratar todo *frame* do jogo como uma previsão de input remoto errada quando apenas o input local é coletado (desconsiderando qualquer cenário de rede). Alguns dos esforços de testes podem ser reduzidos desta forma.

A biblioteca GGPO é uma solução robusta para a sincronização de máquinas e estados de jogo, mas é necessário cumprir com os requisitos e lidar com muitos casos especiais. Por exemplo, mudanças muito abruptas no estado do jogo durante *rollbacks* podem causar um erro visual de pulo. O mesmo se aplica para efeitos visuais, interfaces do usuário, efeitos sonoros, entre outros. Em alguns cenários pode ser necessário processamento e código adicional com o intuito de lidar com esses erros. Em algumas linguagens de programação a criação de objetos e todo o conceito de tempo de vida desses objetos também podem atrapalhar na performance durante *rollbacks* agressivos (geralmente envolve o conceito de projéteis no jogo). Outro caso de exemplo é decidir o fim de uma partida pois é necessário ter alguma certeza de que não ocorrerá mais *rollbacks* e que o evento que determinou o fim do jogo aconteceu de fato. A discussão até aqui acaba mostrando que é considerável estabelecer desde o início de um projeto que o desenvolvimento terá a estratégia de *rollback* para evitar esforços e investimentos adicionais ou até *retrofitting*.

3.5 Loop com GGPO

A Figura 3.5 ilustra como fica o loop de execução já descrito de um jogo quando modificado para incorporar GGPO. Depois da coleta de input, o input é passado para a função ggpo_synchronize_inputs. Esta função irá transmitir os inputs locais para o jogador remoto e também irá fazer um *merge* do input remoto previsto com todos os inputs remotos reais. O resultado é uma tupla de inputs que pode ser passada para a *engine* (com um valor para cada jogador). Ao invés do loop se repetir ao chegar no fim, a função ggpo_advance_frame deve ser chamada antes. Esta irá comparar os inputs recebidos para jogadores remotos com os valores previstos. Ao encontrar uma discrepância, ggpo_advance_frame carrega o último estado do jogo previsto corretamente e executa a função de atualizar o estado do jogo várias vezes, passando os inputs corrigidos com cada

Figura 3.5: Loop incorporando GGPO



Fonte: (CANNON, 2012)

chamada subsequente para avançar a simulação do jogo de volta ao *frame* atual de execução (CANNON, 2012). As funções de carregar e salvar o estado do jogo e de atualizar o mesmo são implementadas pelo desenvolvedor.

GGPO usa um protocolo simples e eficiente para a sincronização de inputs entre jogadores em uma sessão. Um *payload* (parte que contém os dados na mensagem sendo transmitida) é enviado em toda chamada à função ggpo_synchronize_inputs e os inputs sofrem uma compressão de maneira que quase todos os inputs são representados por um único bit. Isso faz com que os inputs de um último *frame* que foi confirmado ou *acknowledged* sejam codificados junto com todo pacote para lidar com pacotes perdidos e fora de ordem de maneira simples. GGPO ainda envia *reports* de qualidade para verificar se a comunicação está justa. O conceito de justiça ou *fairness* está baseado no

conceito de vantagem de *frame* que tenta medir quantos *frames* o jogador local pode estar a frente de um jogador remoto por causa do efeito de *clock skew*. Um *peer* em uma sessão está sempre ciente de sua vantagem local e recebe as atualizações dos outros *peers* com frequência. A comunicação fica justa, pois GGPO atrasa a execução de qualquer *peer* que apresente uma vantagem de *frame* maior para manter a diferença dentro de um valor tolerável (CANNON, 2012).

4 DESENVOLVIMENTO DO JOGO DE LUTA 2D

Considerando todos os pontos que foram propostos para o trabalho na introdução e deixando de lado os conceitos técnicos sobre redes, o texto deste capítulo descreve as tomadas de decisões feitas pelo autor que foram mantidas em um simples diário ao longo da etapa de desenvolvimento.

4.1 Recursos computacionais para implementação

Para a implementação do jogo proposto pelo trabalho foi usada a game *engine* Godot. Uma desvantagem encontrada foi o fato da engine de física ser não-determinística e ser difícil a atualização do estado por demanda. Isso impõe que todo código que reflita a física de algum objeto deve ser planejado pelo desenvolvedor. No contexto do jogo do trabalho, o número de entidades que precisam de comportamento físico é pequeno e o código trivial que lida com os casos específicos do jogo foi desenvolvido caso a caso.

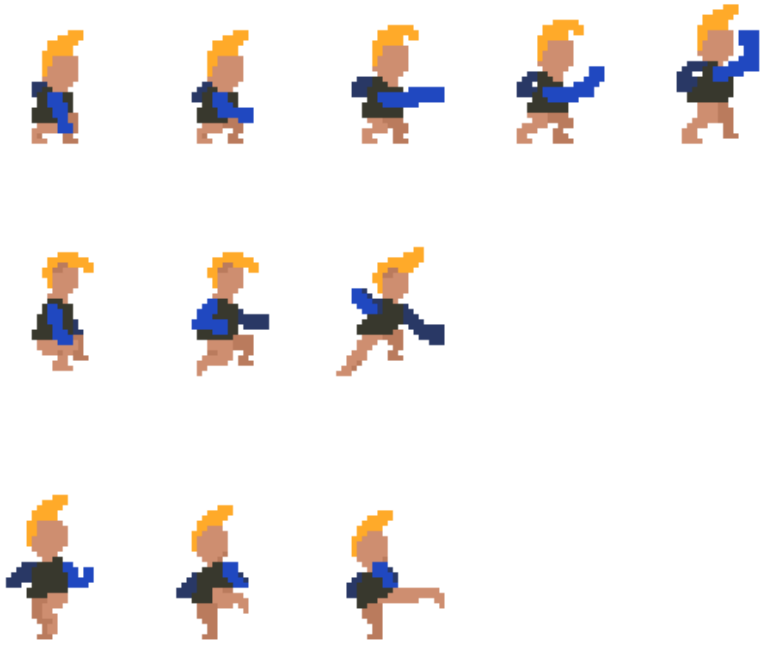
Em especial, foi preciso estabelecer um sistema de colisões que usa o conceito de *axis-aligned bounding box* (MDN, 2022) junto do *node* Position2D para *hitboxes* (caixas invisíveis que, quando ativadas, indicam que seu dono pode causar dano) e *hurtboxes* (caixas invisíveis que, quando ativadas, indicam que seu dono pode sofrer dano) de algumas entidades. As *hitboxes* e *hurtboxes* são definidas de acordo com o tipo da entidade e são ativadas e desativadas conforme ações no jogo. Por exemplo, ao executar um soco, a *hitbox* do soco é ativada e se a *hurtbox* do oponente (que quase sempre está ativa) estiver sobrepondo a área então toda uma sequência de efeitos como remover pontos de vida do oponente, sinalizar o *node* responsável pela interface do usuário, começar novas animações e efeitos sonoros é inicializada.

4.2 Implementação do protótipo

Ao começar o projeto, um *template* baseado em cores para as animações de movimentos para todos os personagens a serem criados foi desenvolvido (Figura 4.1). A ferramenta de edição de imagens utilizada foi GIMP e ela é primitiva no contexto de *pixel art*, ou seja, não há muitas automatizações ou *features* e tais automatizações não acrescentariam ao fluxo de trabalho pessoal de qualquer forma: os personagens apenas precisam

de animações em uma direção; o tamanho de uma *sprite* não passa de 48 pixels em largura e altura, ou seja, a área de trabalho é pequena; embora seja possível utilizar camadas, gerenciar as camadas ao posicionar os pixels se mostrou um trabalho adicional e então uma única camada é usada. Enquanto é importante seguir os muitos princípios de animação, a ideia do *template* foi contemplar quais *frames* são realmente necessários em uma animação sob contexto de um jogo de luta e sob um esforço de reduzir ao máximo o número de *frames* e ser econômico. É necessário ter noção de quanto tempo as animações básicas de um personagem podem precisar para serem finalizadas e em seguida aplicar para um número provável de personagens. O conjunto básico de animações acabou contendo mais de 25 animações variando de 2 a 6 *frames* cada.

Figura 4.1: Parte do *template* que guia a criação de personagens



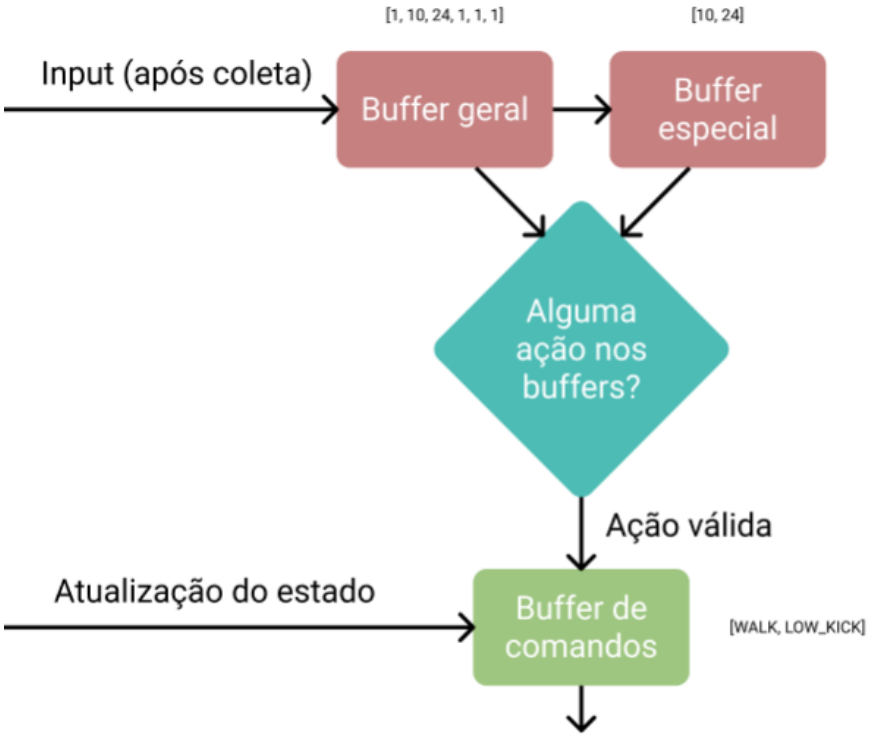
Fonte: autor

Ao finalizar a *sprite sheet* do primeiro personagem (muito similar ao do *template*), a criação de um protótipo de movimento e realização de ações na *engine* foi desenvolvido. Durante o loop do jogo é feita uma coleta de inputs e a mesma coleta é discretizada para um número inteiro com o propósito de cumprir com o requisito de determinismo e outros impostos por GGPO. Em um *buffer* que mantém inputs por um certo número de *frames* (por exemplo, 30 *frames* ou meio segundo), o input discretizado é inserido. Após passar esse tempo em *frames* de execução desde a inserção do input ou se uma ação é realizada, o input é removido. Há o constante *match* desses inputs inseridos (que são convertidos para uma única *string*) com uma lista de ações possíveis codificadas para o personagem

para checar se o jogador está querendo realizar uma ação. Em um momento inicial, este *match* era realizado com expressões regulares. A solução atual envolve usar o operador *in* disponível na linguagem de script da *engine* que checa se uma *string* (aqui no contexto é uma ação) é uma *substring* da *string* do *buffer* de inputs.

Existem prioridades de ações em um jogo de luta e, mesmo sendo possível aperfeiçoar este *match* com estruturas de dados mais sofisticadas, foi apenas criado um segundo *buffer* que mantém apenas os inputs específicos e usados para movimentos especiais. Quando uma ação é realizada e é válida no contexto dos inputs, um terceiro *buffer* que contém apenas comandos a serem realizados no jogo de forma ordenada é utilizado para armazenar a ação válida (convertida para um comando agora) e é acionado em toda atualização do estado. Durante a atualização, outra validação é feita para checar se o personagem realmente pode realizar o comando no momento e, por fim, ir removendo os elementos do *buffer* de comandos. A ideia fica ilustrada na Figura 4.2.

Figura 4.2: *Buffers* de input e de comandos para cada jogador

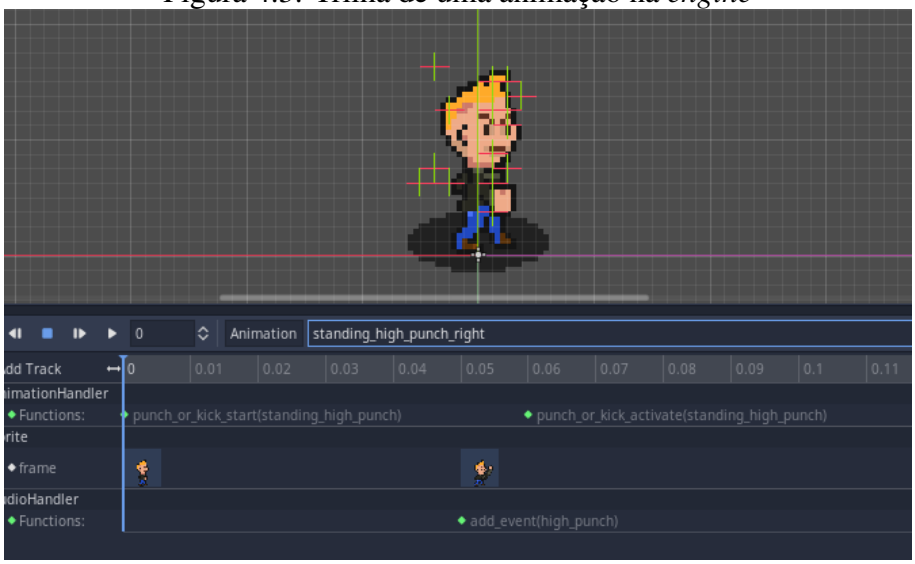


Fonte: autor

Outro aspecto importante do protótipo foi o tratamento de animações e prepará-lo para o cenário de *rollbacks*. Faz parte do estado do jogo o estado de um personagem. Faz parte do estado de um personagem que animação ele está executando e em que posição na trilha da animação. O *node* AnimationPlayer da *engine* permite um processamento ma-

nual da animação através do modo ANIMATION_PROCESS_MANUAL. Para avançar todo *frame* é possível usar o método *advance* e usar como argumento o tempo equivalente a 1 *frame*. Outro método importante é o *seek* que é especialmente usado ao carregar um estado do jogo. A diferença entre os dois métodos é que o primeiro executa eventos na trilha de animação ao mudar de posição. Para funções que ativam *hitboxes* e *hurtboxes* na trilha da animação, o modo ANIMATION_METHOD_CALL_IMMEDIATE também é importante para evitar qualquer atraso de chamada de método e eventualmente quebrar a sincronização. A Figura 4.3 ilustra a trilha de animação de um golpe. Um segundo aspecto importante foi o escalamento aplicado a todos os recursos de *sprites*. Como a movimentação ficou baseada em valores inteiros apenas, ao usar uma nova escala foi possível estabelecer, por exemplo, 10 pixels como unidade básica para movimentação ao invés de 1 pixel e garantir uma movimentação mais suave para os visuais.

Figura 4.3: Trilha de uma animação na *engine*



Fonte: autor

4.3 Adicionando elementos secundários e *polish* ao protótipo

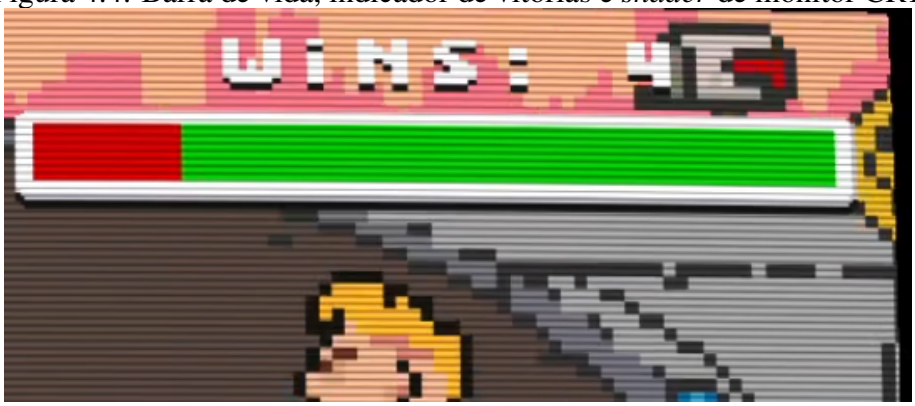
A maior inspiração para o jogo é a série Mortal Kombat. Grande maioria de movimentos e movimentos especiais tomam como base a série. Isso ainda inclui a mecânica de uma partida ter no máximo 3 *rounds* e que o jogador vencedor é o que obtiver vitória primeiro em 2 *rounds*. Outra mecânica é o fato dos personagens apresentarem no mínimo 3 movimentos especiais junto ao conjunto básico de socos, chutes e suas variações.

Após consolidar o protótipo, outros aspectos mais visuais e também relacionados

ao *game feel* (SWINK, 2008) podem ser abordados. O indicador de vida e o de vitórias do jogador são criados (Figura 4.4), o indicador de tempo restante da partida é acrescentado, a movimentação do protótipo e animações são reajustadas, alguns efeitos visuais são criados (para indicar acerto de golpe, bloqueio, pouso após pulo, entre outros). Transições de interface, ajustes de câmera, *camera shake*, *shaders* (monitor CRT), menus e toda a lógica relacionada aos menus são adicionados. Vale citar que os efeitos visuais e sonoros, por exemplo, não fazem parte do estado lógico do jogo e são processados de uma maneira diferente em relação ao mesmo. Em paralelo, outros personagens e seus cenários também já podem ser explorados. É considerado um número de 3 personagens para manter o escopo pequeno do projeto.

De acordo com análise do diário, desenvolver as animações básicas de um personagem leva em média 10 dias e, juntando com animações de movimentos especiais e seu cenário, a média é 2 semanas. Como as animações básicas são comuns a todos os personagens, um *node* Sprite na *engine* fica encarregado de manter a *sprite sheet* e apenas mudar a textura do *node* de acordo com o personagem a ser exibido na tela. Para as animações de movimentos especiais são utilizados *nodes* Sprite para cada animação pois cada uma é única nesse contexto. Para os cenários, alguns *nodes* da *engine* são usados como o de *parallax scrolling* que permitiu criar um efeito de movimento às imagens que até então eram estáticas.

Figura 4.4: Barra de vida, indicador de vitórias e *shader* de monitor CRT



Fonte: autor

Em sequência foram desenvolvidos os efeitos sonoros do jogo. Como explicado anteriormente, a ferramenta utilizada foi o *tracker* DefleMask. O sistema escolhido no software foi o Sega Genesis ou Mega Drive (Figura 4.5). Aliado ao estilo *pixel art*, um som sintetizado e um pouco de nostalgia influenciaram a escolha. Comparando com músicas, para a criação de efeitos é preciso uma abstração e experimentação maior, além

de que cada tipo de efeito acabou sendo um instrumento único. Tendo ideia do som desejado e ao alcançar algo parecido, algumas iterações explorando outros parâmetros ainda são realizadas. Dentre os sons estão os de navegação em menus, socos e variações, chutes e variações, dano sofrido e variações, movimentos especiais e golpe bloqueado.

Conforme os sons são inseridos, é possível perceber na prática o problema do *rollback*. Para ajudar a reduzir o efeito repetitivo dos erros sonoros (e para erros visuais também) durante *rollbacks* foi adicionado um pequeno teste. Para transições e elementos de interface é completamente ignorado o input do jogador ou nenhum tratamento é realizado. Por exemplo, a barra de vida de um personagem é lentamente esvaziada ou enchida de maneira que mudanças abruptas não causam efeitos visíveis. O teste funciona da seguinte maneira: um *buffer* de eventos é mantido e, ao adicionar um novo evento, é verificado se ele é similar a algum já presente (mesmo tipo, mesma fonte emissora, entre outros atributos). Se sim, é verificado a diferença em *frames* do *frame* atual de execução com o de inserção do evento. Se a diferença estiver dentro de uma janela (que corresponde a uma latência desejada) então remove-se o novo evento. Caso o novo evento não seja similar a um presente, simplesmente adicionar ao *buffer*. É ainda adicionado um atraso de alguns *frames* antes de iniciar qualquer evento de fato. Nesse mesmo contexto, eventos que são acionados em trilhas de animações podem ser afastados do início da trilha para ajudar reduzir alguns dos efeitos indesejáveis.

Figura 4.5: Interface do *tracker* DefleMask

Fonte: autor

Outra parte importante é a de projéteis que são gerados por certos movimentos especiais (Figura 4.6). Em uma primeira implementação, os projéteis faziam uso do *node* AnimationPlayer e de trilhas de animações, porém, ao usar o atributo de posição, os valores sofrem uma interpolação e isso afeta a necessidade de evitar ao máximo valores em ponto flutuante. Eventualmente os projéteis passaram a ter um comportamento similar mas definido por script. Há o problema da criação e destruição desses projéteis também. Para alguns jogos e linguagens de programação pode ser preciso definir *pools* de objetos que são constantemente criados, por exemplo, para melhorar a performance (NYSTROM, 2009). Mas, a técnica não se mostrou necessária com Godot e nenhum tratamento para *rollbacks* foi acrescentado nesse sentido.

Figura 4.6: Exemplo de projétil

Fonte: autor

Um dos aspectos finais na produção foram as músicas. Novamente foi utilizado DefleMask, porém um conjunto limitado de instrumentos foi criado. A ideia foi criar aproximadamente 30 minutos de músicas com tal conjunto de instrumentos onde cada faixa teria no mínimo 1 minuto e no máximo 3 minutos de duração. A grande maioria das músicas são destinadas para durante as partidas, enquanto que as demais servem para certos momentos como a tela de seleção de personagem, menu principal, modo de treinamento, entre outros.

Funcionalidades extras são implementadas e incluem o *remap* de inputs (teclado ou controle), um modo local para 2 jogadores e um modo de treinamento primitivo onde é possível acompanhar a lista dos movimentos especiais para o personagem escolhido e também visualizar as áreas das *hitboxes* e *hurtboxes* ativas.

4.4 Função de atualização do estado

Figura 4.7: Função (simplificada) de atualização do estado do jogo

```
6  func Update(delta, inputs):
7      var i = 0
8
9      for player in players:
10         player.input_buffer.handle_input(inputs[i], false)
11         player.command_buffer.handle_commands()
12         player.animation_handler.update(delta)
13         player.handle_physics(delta)
14         player.animation_handler.handle_animation()
15
16         i = i + 1
17
18     ProjectileHandler.update_all(delta)
19     CollisionSystem.handle_projectiles()
20
21     world.handle_timer()
22     world.incr_frame_count()
```

Fonte: autor

A Figura 4.7 ilustra a função que atualiza o estado do jogo. Para cada jogador são executadas algumas funções. Após GGPO lidar com os inputs, os mesmos são passados para a função handle_input que irá adicionar os inputs aos *buffers* e fazer o *match* já descrito. O segundo parâmetro indica se o jogo está no menu de seleção de personagem ou durante uma partida. Em seguida, a função handle_commands irá iniciar qualquer ação válida nos *buffers* que foi convertida para um comando, verificar se tal jogador pode executar tal comando e ir esvaziando o *buffer* de comandos. Na sequência, a função update do *handler* de animações irá atualizar a animação atual do personagem (avançar usando *delta* com um valor equivalente ao tempo de 1 *frame*).

A física é tratada na etapa seguinte. Esta etapa envolve aplicar um efeito falso de gravidade, ativar e desativar *hurtboxes* do personagem conforme seu estado lógico, calcular a posição do personagem usando a nova escala, orientar a *sprite* do personagem, ajustar as velocidades em cada eixo do personagem, manter o personagem nos limites da câmera e verificar se as *hitboxes* e *hurtboxes* ativas do personagem estão colidindo com a de outros personagens para ativar eventos de golpes ou impedir que o personagem fique em cima de outro, por exemplo. A etapa seguinte utiliza o *handler* de animação e o novo estado lógico do jogador para iniciar uma nova animação (se for o caso).

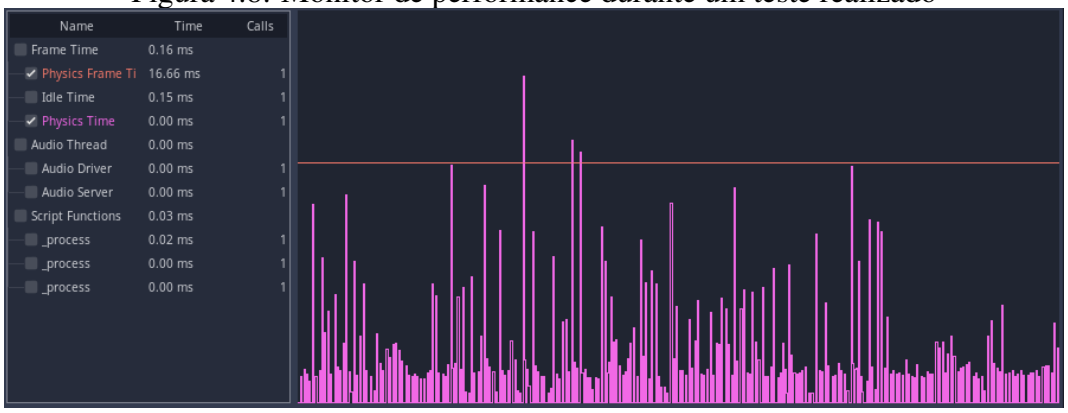
Após processar para cada jogador, é realizado o processamento de qualquer projétil ativo (atualizando sua posição e comportamento) com o respectivo *node singleton*.

Na sequência é realizado o teste de colisão (também com o respectivo *node singleton*) das *hurtboxes* dos personagens com os projéteis ativos para ativar eventos de impacto caso necessário. Finalmente, o *handler* que trata de todos os pontos relacionados ao tempo no jogo é chamado. Por exemplo, verificar se um *round* terminou, marcar início de *round*, sinalizar para o *node* lidando com o indicador de tempo da interface, entre outros. O número do *frame* atual é incrementado ainda.

4.5 Funções de salvar e carregar o estado

O estado do jogo é composto pelo estado dos personagens e do estado geral de controle do jogo. Como parte do estado de um personagem faz sua animação atual e posição na trilha, o estado dos *buffers*, sua posição no cenário e velocidade atuais, sua quantidade de pontos de vida e toda uma lista de variáveis do tipo *boolean* indicando estados menores como se o personagem pode se mover, se está pulando, se está caminhando, entre outros. Faz parte do estado de controle do jogo o modo do jogo, o *frame* em que começou um *round*, o número do *round* atual, o número de vitórias dos jogadores, quais *hurtboxes* e *hitboxes* estão ativas, o índice indicando qual cenário deve ser ativado, todos os dados com relação aos projéteis, entre outros. A *engine* disponibiliza maneiras de lidar com a serialização do estado usando a classe StreamPeerBuffer e os métodos put_var e get_var.

Figura 4.8: Monitor de performance durante um teste realizado



Fonte: autor

Para as funções de salvar e carregar o estado do jogo não foram realizadas otimizações. Porém, para checar a performance atual do jogo foi possível utilizar o monitor de performance da *engine*. A Figura 4.8 mostra a performance quando seguindo as condições do teste E descrito no próximo capítulo. Importante notar que Physics diz respeito

às funções executadas em todo *frame* de física da *engine* de acordo com uma frequência (60 *frames* por segundo) e, no contexto do projeto, inclui as 3 funcionalidades anteriores. É seguro confirmar que no mínimo 4 ou 5 atualizações extras por *frame* em casos de *rollback* estão sendo realizadas sem atrapalhar na performance de execução do jogo com as condições do teste E. É possível imaginar que a situação é diferente de acordo com o ambiente de execução e o hardware. Para um desenvolvedor é importante estabelecer a janela de latência que será adotada e o *budget* em poder de processamento para trabalhar e otimizar conforme.

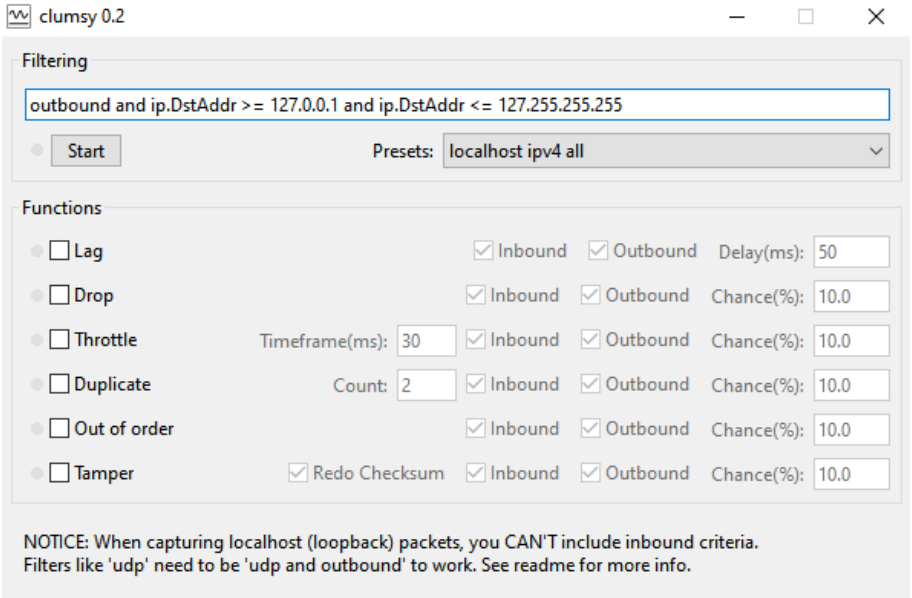
5 TESTES

O objetivo deste capítulo é avaliar o protótipo já em um estado mais polido em alguns cenários de rede. Para isso, testes em uma mesma máquina com o intuito de validar as mecânicas do jogo e a sincronização de estado são realizados. A latência é simulada por meio de um software com variações de parâmetros. Por fim, testes em máquinas diferentes utilizando a API de comunicação da loja digital Steam são realizados.

5.1 Metodologia dos testes

Para testar todas as funcionalidades do projeto e avaliar a qualidade esperada da aplicação, é importante lembrar de alguns conceitos. Por ser uma solução híbrida, GGPO faz uso de *input delay*. Embora seja possível modificar o valor durante o jogo, para a maioria dos testes foi fixado o valor de 2. Por se tratar de um jogo de luta, é crucial que o input local seja o mais responsivo possível sem que o usuário perceba atrasos. Um teste adicional irá fazer uso de um valor de 8. Para simular a latência na maioria dos casos é usada a ferramenta Clumsy cuja interface está ilustrada na Figura 5.1.

Figura 5.1: Interface da ferramenta Clumsy



Fonte: autor

Inicialmente, será avaliado o funcionamento de duas instâncias do jogo interagindo em uma mesma máquina. Em seguida, será realizada uma outra avaliação com duas máquinas diferentes com o objetivo de avaliar o comportamento e impacto de possí-

veis variações na rede fora de um ambiente de simulação.

5.2 Testes na mesma máquina

Para realizar os testes são executadas duas instâncias do jogo na mesma máquina. A primeira fica executando ao fundo enquanto a segunda irá tratar de se comunicar com a primeira e ainda será responsável pelos inputs remotos. Com os parâmetros da ferramenta Clumsy será possível verificar os efeitos de *rollbacks* na primeira instância. Para os fins dos testes, as músicas são desativadas e apenas é possível escutar os efeitos sonoros da primeira instância. Junto a cada teste, um vídeo de demonstração do mesmo será gravado e usado como referência. Nos vídeos, a primeira instância estará no lado esquerdo enquanto que a segunda instância estará no lado direito, como ilustrado na Figura 5.2.

Figura 5.2: Configuração para testes na mesma máquina

Fonte: autor

O primeiro teste terá como parâmetro de *lag* o valor de 50 milissegundos apenas. Já que o *input delay* irá amortecer um atraso de tempo equivalente a 2 *frames*, com este valor já é possível ter alguma latência durante a execução. O segundo teste terá como *lag* 50 milissegundos e terá como valor para o parâmetro de perda a chance de 10%. O terceiro para o *lag* terá o valor de 50 milissegundos, para o de perda a chance de 20% e para o parâmetro de trocar a ordem uma chance de 10%. O quarto terá como *lag* o valor de 75 milissegundos, para o de perda a chance de 30% e para o parâmetro de trocar a ordem uma chance de 20%. O quinto terá como *lag* o valor de 100 milissegundos, para o de perda a chance de 30% e para o parâmetro de trocar a ordem uma chance de 20%. Um teste adicional terá apenas o parâmetro de *lag* com o valor de 150 milissegundos, porém o *input delay* será de 8 para verificar os *trade-offs* por ajustar o atraso artificial para o input

local.

5.2.1 Teste A - *lag*: 50 ms

Este teste busca explorar uma movimentação com um padrão relativamente avançado fazendo uso eventual de alguns movimentos especiais por parte do personagem com os inputs remotos. O cenário se aproxima muito do ideal e o número de erros visíveis por causa dos efeitos de *rollback* é quase mínimo (LENTZ, 2022a). Como primeiro teste serviu para verificar a sincronização no geral, para verificar se a aplicação está cumprindo com os requisitos impostos pela tecnologia GGPO e se as mecânicas básicas estão funcionando da maneira esperada.

5.2.2 Teste B - *lag*: 50 ms, perda: 10%

Este teste explora, por parte dos inputs remotos, um outro personagem cujos movimentos especiais permitem testar de uma forma melhor os efeitos quando no estado de agachado. O cenário deste teste ficou similar ao cenário do teste antecedente. No entanto, o teste conseguiu capturar em alguns momentos um erro visual principalmente quando o personagem sendo controlado se encontrava no estado de agachado e inputs direcionais eram pressionados (seja tentando realizar um dos seus movimentos especiais ou pressionando aleatoriamente) (LENTZ, 2022b). Uma ação de movimento é iniciada quando não deveria ser possível. Como GGPO está tratando da situação remete à questão da previsão de inputs e uma transição menos abrupta das mudanças de estado do personagem no código após os testes pôde resolver o problema (que não foi observado durante as etapas prévias de desenvolvimento).

5.2.3 Teste C - *lag*: 50 ms, perda: 20%, troca de ordem: 10%

Este teste buscou explorar a movimentação de forma mais intensiva por parte dos inputs direcionais com o personagem sendo controlado: alguns golpes são realizados, seguido de um movimento especial, seguido de uma sequência encadeada de movimentos especiais e outros golpes para finalizar com movimentos não tão padronizados que servem para testar previsões erradas e as suas consequências com a latência estabelecida (LENTZ,

2022c). Como resultado acabou cobrindo as questões já refletidas nos outros cenários. A consideração final foi de aumentar o atraso.

5.2.4 Teste D - *lag*: 75 ms, perda: 30%, troca de ordem: 20%

Este teste explora o terceiro personagem disponível e grande parte de seus movimentos. O padrão de movimentação por parte dos inputs remotos é mais natural, se aproximando dos primeiros testes A e B. Neste cenário, erros visuais de pulo ficam realçados para toda a movimentação. Em alguns instantes é possível verificar ainda uma repetição dos efeitos visuais e sonoros (LENTZ, 2022d). Após os testes, a janela de latência do teste ao inserir eventos de som ou de visual foi aumentada. Para o próximo teste, é considerado um aumento do atraso novamente para um caso que comece afetar muito a experiência do usuário se ficar notado que uma simples ação de soco tem uma janela de 200 milissegundos.

5.2.5 Teste E - *lag*: 100 ms, perda: 30%, troca de ordem: 20%

Este teste busca combinar uma movimentação que vai gerar previsões erradas com alguns golpes e eventuais movimentos especiais por parte do personagem sendo controlado. Novamente os erros visuais de pulo estão presentes. Na vez em que o segundo personagem é controlado é possível perceber que um movimento especial de *dash* foi executado em falso pelo fato de que GGPO teve como previsão uma ação que adicionada ao buffer de input deu *match* e gerou tal comando (LENTZ, 2022e). Como soluções é possível mudar a combinação de inputs necessária, simplesmente manter este efeito ou adicionar um atraso maior na execução da ação (especialmente para o evento de som).

5.2.6 Teste F - *lag*: 150 ms, *input delay*: 8 *frames*

Finalmente, para verificar que conexões com condições longe do ideal podem ainda ter uma experiência razoável, é realizado este teste adicional onde o *input delay* passa do valor de 2 para o valor de 8. Isso irá adicionar um atraso ao input local que não é visível no vídeo de referência e que pode atrapalhar a experiência também por aumentar o tempo de resposta entre o apertar de um botão e a ação sendo exibida na tela. Já que

8 *frames* equivalem aproximadamente a um tempo de 128 milissegundos é possível notar que grande parte da latência é amortecida pelo atraso artificial. Assim, o jogo se mantém relativamente suave em um cenário de latência alta (LENTZ, 2022f).

Os testes na mesma máquina conseguiram validar grande parte das mecânicas do jogo e principalmente verificar se a aplicação cumpre com os requisitos técnicos impostos por parte da solução de rede. Alguns comportamentos que não foram percebidos em etapas anteriores foram notados. Também foi possível perceber que um ajuste do *input delay* pode ser importante para trabalhos futuros e para contribuir na experiência do usuário em cenários específicos de rede. As mudanças propostas nos testes foram adicionadas para a realização dos testes em máquinas diferentes apresentados na próxima seção.

5.3 Testes em máquinas diferentes

Esta seção tem como objetivo testar a aplicação em um cenário que não seja simulado e que faça uso de duas máquinas. Para realizar testes em máquinas diferentes foi considerado a integração da aplicação com a loja de jogos digitais Steam e há uma motivação para isso. A tecnologia GGPO fornece apenas o essencial para o funcionamento de um jogo online e fica a critério do desenvolvedor em como resolver outros problemas relacionados como o *matchmaking* que realiza a função de parear dois usuários (ou mais) que queiram jogar. Antes do início de uma partida também é preciso todos os dados relacionados à comunicação. O código original da GGPO suporta *sockets* e é necessário *port forwarding* e configurações em roteadores para contornar algumas questões como *network address translation* ou NAT, *firewall*, entre outros.

Com isso em mente, a Steam disponibiliza uma API que fornece acesso a muitas funcionalidades, principalmente a de comunicação P2P que resolve as questões anteriores (a técnica de NAT *punch-through* é um exemplo) e as mensagens podem ainda serem transmitidas pelo *backbone* da Steam quando apropriado (*relay*). Existem algumas interfaces como opções para o desenvolvedor e a ISteamNetworkingMessages pareceu a mais simples para um pequena adaptação do código da aplicação. Essa interface busca ser similar a qualquer código que faça uso de *user datagram protocol* ou UDP (como GGPO). Isso quer dizer que *handles* de conexão não são utilizados e não é necessário gerenciar qualquer estado da comunicação pois, no momento em que dois usuários começam a trocar mensagens, implicitamente é estabelecida uma conexão que faz uso da interface ISteamNetworkingSockets (de nível mais baixo). Assim, um usuário precisa apenas for-

necer o identificador Steam de seu oponente e, se o oponente está querendo se comunicar com ele também, a conexão pela Steam é estabelecida.

As interfaces gráficas da aplicação foram modificadas para contemplar o uso dos recursos pela Steam e ainda foi adicionada a possibilidade do usuário poder modificar o valor do *input delay* com um número entre 2 e 8. O código fonte da GGPO foi modificado para usar os métodos disponibilizados pela API da Steam e, em seguida, foi feita uma nova *build*. O mesmo aconteceu para o código fonte da *engine* que precisou de uma nova *build* após atualização do código da GGPO.

Figura 5.3: Configuração para testes em máquinas diferentes

Fonte: autor

5.3.1 Teste U (único)

Este teste teve duas máquinas diferentes utilizando o serviço de comunicação P2P disponibilizado pela loja digital Steam. O *input delay* para ambas permaneceu no padrão de 2. O teste é diferente dos discutidos e apresenta apenas a perspectiva da mesma máquina dos testes anteriores (Figura 5.3). Apesar do vídeo ficar com um *frame rate* baixo em alguns momentos, é possível perceber as mudanças que foram realizadas, ouvir as músicas, além de ser possível ver uma estimativa do *ping* (LENTZ, 2022g). No geral, o teste não buscou explorar qualquer possibilidade de erros mas, apenas testar as funciona-

lidades de comunicação com as alterações propostas. Como resultado, conseguiu validar as funcionalidades e não apresentou algum comportamento diferente além do que já foi notado.

6 CONCLUSÃO

O trabalho teve como principal objetivo demonstrar todo o processo de desenvolvimento de um jogo que incorpora *rollback netcode*. Os conceitos importantes sobre o lado mais técnico foram abordados, além do lado criativo ser refletido em sequência. Alguns pontos importantes podem ser notados. Por mais que exista uma solução de baixo custo para tratar de problemas de redes, a experiência do usuário acaba sendo afetada no final. Mas, ao querer usar uma estratégia mais sofisticada, investimentos maiores são necessários por parte do desenvolvedor. Ao incorporar *rollback netcode* é interessante planejar todo o desenvolvimento desde o início pois, além da estratégia impor seus próprios requisitos, casos especiais e outras questões não triviais vão aparecer. Como exemplo, a *engine* de física que faz parte da *engine* escolhida para desenvolvimento não consegue cumprir com os requisitos impostos e foi necessário contornar o problema.

Outros pontos não foram explorados no trabalho, como a possibilidade de otimizar as funcionalidades de salvar e carregar o estado do jogo. É de muita importância ter essas funcionalidades e a atualização do estado otimizadas. Vale lembrar que é possível ajustar alguns parâmetros relacionados ao *netcode* adotado (como o *input delay*) com o propósito de tentar reduzir o número de *rollbacks*, a demanda imposta por tais funcionalidades e os erros ou efeitos consequentes. Entretanto, isso acaba dependendo do contexto e do tipo de jogo sendo criado. É possível adotar um mecanismo que automaticamente controle o *input delay* ou deixar que os usuários o configurem ou mantê-lo fixo também.

Ainda seria possível aperfeiçoar o tratamento dos sons e efeitos visuais considerando estes como parte do estado do jogo. Uma das possíveis maneiras seria quando a função de carregar o estado do jogo é chamada (por exemplo, em um *rollback*). Mantendo informação de quais efeitos visuais e sons estão sendo executados, ao carregar um estado é possível realizar uma comparação e obter um controle maior do que realmente deveria estar sendo executado ou não.

Do ponto de vista do lado criativo, uma liberdade maior foi garantida e foram explorados visuais no estilo *pixel art*, além de sons e músicas que fazem uso da técnica de síntese FM. A combinação acabou criando uma atmosfera retrô e até nostálgica por lembrar de jogos e dispositivos comuns da década de 1990.

6.1 Trabalhos futuros

Como todo o processo relatado buscou a conceitualização e produção até o fim seguindo um escopo definido, o resultado final foi um jogo digital que pode ser considerado um MVP. A continuação deste projeto tem como objetivo continuar a integração com a loja Steam e lançá-lo. Em seguida, verificar por meio de *feedbacks* e *reviews* se iniciar uma fase de produção de conteúdo para o jogo é interessante. No caso positivo, seguir com o ciclo de desenvolvimento que poderá contemplar a inserção de um sistema de *lobby* e *matchmaking* completo, um modo de treinamento mais sofisticado, um modo *singleplayer* e o desenvolvimento de um controlador que faça uso de inteligência artifical (para este último modo). No caso negativo, os usuários podem não ter mostrado interesse sendo necessário aperfeiçoar o projeto ou simplesmente abandoná-lo.

REFERÊNCIAS

CANNON, T. Fight the lag! the trick behind ggpo's low-latency netcode. **Game Developer**, v. 19, n. 9, p. 7–13, set. 2012.

CLUMSY, an utility for simulating broken network for Windows Vista / Windows 7 and above. 2022. Dísponível em: <http://jagt.github.io/clumsy/>. Acesso em: 29 abr 2022.

CONNECT to Work or Games from Anywhere | Parsec. 2022. Dísponível em: <https://parsec.app/>. Acesso em: 18 maio 2022.

DEFLEMASK Tracker. 2022. Dísponível em: <https://www.deflemask.com/>. Acesso em: 28 abr 2022.

GGPO | Rollback Networking SDK for Peer-to-Peer Games. 2022. Dísponível em: <https://www.ggpo.net/>. Acesso em: 29 abr 2022.

GIMP - GNU Image Manipulation Program. 2022. Dísponível em: <https://www.gimp.org/>. Acesso em: 28 abr 2022.

GODOT Engine - Free and open source 2D and 3D game engine. 2022. Dísponível em: <https://godotengine.org/>. Acesso em: 28 abr 2022.

KILLER Instinct. 2022. Dísponível em: <https://www.ultra-combo.com/>. Acesso em: 29 abr 2022.

LAWSON, A. **Online fighting games during COVID: How rollback helps us connect | Ars Technica**. 2021. Dísponível em: <https://arstechnica.com/gaming/2021/02/how-guilty-gear-saved-its-online-play-in-a-post-offline-world/>. Acesso em: 30 mar 2022.

LENTZ, H. S. **Demonstração do Teste A**. 2022. Dísponível em: <https://drive.google.com/file/d/1qmQu_K45EYk2wgDnuWnDEEY3T7BrQT1M/view>. Acesso em: 30 mar 2022.

LENTZ, H. S. **Demonstração do Teste B**. 2022. Dísponível em: <https://drive.google.com/file/d/1c_Kn6bAYYwSOhFaarSwMiGXTBKhxBXrf/view>. Acesso em: 30 mar 2022.

LENTZ, H. S. **Demonstração do Teste C**. 2022. Dísponível em: <https://drive.google.com/file/d/1BU_bFf0o5MDfthonjHVngxGlBukTgkxC/view>. Acesso em: 30 mar 2022.

LENTZ, H. S. **Demonstração do Teste D**. 2022. Dísponível em: <https://drive.google.com/file/d/1opMTmwTHqu6kWhwbaaBSXjzhXOePEagN/view>. Acesso em: 30 mar 2022.

LENTZ, H. S. **Demonstração do Teste E**. 2022. Dísponível em: <https://drive.google.com/file/d/1OdFu9I0SrqD26Wc0QARXJ1hCSWr3uhen/view>. Acesso em: 30 mar 2022.

LENTZ, H. S. **Demonstração do Teste F**. 2022. Dísponível em: <https://drive.google.com/file/d/190Im30UyYYros7uRWgyLWqcaB5Yu-4QF/view>. Acesso em: 30 mar 2022.

LENTZ, H. S. **Demonstração do Teste U**. 2022. Dísponível em: <https://drive.google.com/file/d/1MQsJbmmCNzlyVjrHZmpiK7ZSwNN2HY7q/view>. Acesso em: 23 abr 2022.

MDN. **2D collision detection - Game development | MDN**. 2022. Dísponível em: <https://developer.mozilla.org/en-US/docs/Games/Techniques/2D_collision_detection>. Acesso em: 29 abr 2022.

MORTAL Kombat 11 Ultimate. 2022. Dísponível em: <https://mortalkombat.com/>. Acesso em: 29 abr 2022.

NYSTROM, R. **Object Pool · Optimization Patterns · Game Programming Patterns**. 2009. Dísponível em: <https://gameprogrammingpatterns.com/object-pool.html>. Acesso em: 30 mar 2022.

POCKET Rumble. 2022. Dísponível em: <https://pocketrumble.com/>. Acesso em: 29 abr 2022.

PUSCH, R. **Explaining how fighting games use delay-based and rollback netcode | Ars Technica**. 2019. Dísponível em: <https://arstechnica.com/gaming/2019/10/explaining-how-fighting-games-use-delay-based-and-rollback-netcode/>. Acesso em: 30 mar 2022.

SKULLGIRLS 2nd Encore. 2022. Dísponível em: <https://skullgirls.com/>. Acesso em: 29 abr 2022.

STEAM Remote Play. 2022. Dísponível em: <https://store.steampowered.com/remoteplay>. Acesso em: 18 maio 2022.

STEAMWORKS SDK (Steamworks Documentation). 2022. Dísponível em: <https://partner.steamgames.com/doc/sdk>. Acesso em: 29 abr 2022.

SUN, R. **Game Networking Demystified, Part III: Lockstep**. 2019. Dísponível em: <https://ruoyusun.com/2019/04/06/game-networking-3.html>. Acesso em: 30 mar 2022.

SWINK, S. **Game Feel**: A game designer's guide to virtual sensations. [S.l.]: Taylor & Francis, 2008.

THEM'S Fightin' Herds – Mane6. 2022. Dísponível em: <https://www.mane6.com/>. Acesso em: 29 abr 2022.

WIKIPEDIA. **Yamaha YM2612 - Wikipedia**. 2022. Dísponível em: <https://en.wikipedia.org/wiki/Yamaha_YM2612>. Acesso em: 30 mar 2022.

WOOLUMS, K. **Influenciando jogos até hoje, 'Street Fighter II' finalmente entra para o 'Hall da Fama' dos games - ESPN**. 2017. Dísponível em: <http://www.espn.com.br/noticia/692246_influenciando-jogos-ate-hoje-street-fighter-ii-finalmente-entra-para-o-hall-da-fama-dos-games/>. Acesso em: 30 mar 2022.