UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Thiago da Fonte Bastos

Incorporação de Rollback no Game Loop de Jogos em Tempo Real Multijogadores Online

RECIFE

2023

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Thiago da Fonte Bastos

Incorporação de Rollback no Game Loop de Jogos em Tempo Real Multijogadores Online

Monografia apresentada ao Centro de Informática (CIn) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para conclusão do Curso de Ciência da Computação, orientada pelo(a) professor(a) Geber Lisboa Ramalho.

RECIFE
2023
UNIVERSIDADE FEDERAL DE PERNAMBUCO

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Bastos, Thiago da Fonte.
Incorporação de Rollback no Game Loop de Jogos em Tempo Real Multijogadores Online / Thiago da Fonte Bastos. - Recife, 2023.
51 p. : il.

Orientador(a): Geber Lisboa Ramalho
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2023.

1. C++. 2. Determinismo. 3. Jogos digitais. 4. Networking. 5. Peer-to-peer. I. Ramalho, Geber Lisboa. (Orientação). II. Título.

000 CDD (22.ed.)

CENTRO DE INFORMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Thiago da Fonte Bastos

Incorporação de Rollback no Game Loop de Jogos em Tempo Real Multijogadores Online

Monografia submetida ao corpo docente da Universidade Federal de Pernambuco, defendida e aprovada em 27 de Abril de 2023.

Banca Examinadora:

_____ Orientador(a)

Geber Lisboa Ramalho

Doutor(a)

_____ Examinador(a)

Giordano Ribeiro Eulálio Cabral

Doutor(a)

A minha mãe, Sandra, por sempre me encorajar a seguir meus sonhos e vocações, e me motivar até nos momentos que eu não acreditava em mim mesmo.

AGRADECIMENTOS

A Sílvia da Fonte e Dra. Ângela Polimeni, pelas revisões de texto.
A Pedro Peixoto e Lourinaldo Holanda, por jogar comigo o jogo do projeto e me dar a chance de vê-lo em ação num cenário real.

RESUMO

O rollback é um método de networking de jogos digitais que tem como objetivo eliminar o atraso de resposta entre jogadores causado pelo tempo que leva para dados serem transmitidos através da rede. Tal método tem sido bem-sucedido e levado a cenários competitivos de jogos em tempo real a florescerem online quando em outrora apenas partidas presenciais seriam consideradas aceitáveis. Entretanto, esta solução é adotada apenas dentro do gênero de jogos de luta e não muito buscada como uma possível melhoria da experiência online em outros tipos de jogos. Este trabalho busca compreender o que é intrínseco à aplicação do rollback e como isso pode vir a abranger os demais gêneros de jogo que podem se beneficiar de suas vantagens, e então implementá-lo em um jogo de um gênero diferente.

Palavras-chave: C++, Determinismo, Jogos digitais, Networking, Peer-to-peer

ABSTRACT

Rollback is a video game networking method that aims to eliminate the response delay between players caused the time data takes to travel through the net. Such method has been successful and led to competitive scenes to flourish online when otherwise only on-site matches would have been deemed acceptable. Nevertheless, this solution has only been adopted within the fighting game genre and not much sought after as a possible improvement of the online experience in other kinds of games. This work seeks to comprehend what is inherent to the application of rollback and how this can come to encompass the other game genres that can benefit from its advantages, and then implement it in a game of a different genre.

Keywords: C++, Determinism, Games, Networking, Peer-to-peer

Sumário

Lista de Figuras

Tabela de Siglas

Sigla	Significado
fpm	Fixed-point math (matemática de ponto fixo)
FPS	Frames por segundo
GGPO	Good Game, Peace Out
GLSL	OpenGL Shading Language
GPU	Graphics processing unit (unidade de processamento gráfico)
HP	Hit points
MMO	Massively multiplayer online (jogo massivamente multijogador online)
MMORPG	Massively multiplayer online role-playing game
NPC	Non-playable character (personagem não-jogável)
P2P	Peer-to-peer
PC	Personal computer (computador pessoal)
RNG	(Pseudo)-random number generator (gerador de número (pseudo)-aleatório)
RTS	Real-time strategy (jogo de estratégia em tempo real)
RTT	Round trip time
STL	Standard Template Library
TOML	Tom's Obvious Minimal Language
UDP	User Datagram Protocol

1. Introdução

O início da pandemia da COVID-19 foi marcado pelo impedimento da realização de eventos presenciais sem nenhuma previsão de quando poderiam voltar a acontecer. Isso teve grande impacto na realização de campeonatos profissionais de esportes eletrônicos, os esports, que precisaram passar a serem realizados remotamente, através de partidas online. [4]

Um aspecto inevitável de partidas online em jogos digitais é a presença da latência entre computadores, comumente chamada de lag. A métrica mais comum, se não universal, para medir a intensidade da latência é o *round trip time* ou ping, o tempo que uma informação leva para atravessar a rede e chegar no outro computador, para então retornar com uma resposta. [10]

Ao menos que a partida se dê através de uma rede local, uma ação realizada por um jogador nunca é recebida instantaneamente pelos outros. Isso gera uma disparidade entre os estados de jogo vistos por cada jogador e pode levar de experiências aquém do esperado a métodos de explorar a disparidade para obter vantagens injustas no jogo. [10]

Cada gênero de jogo online emprega uma técnica de networking própria para prover a melhor experiência possível naquele contexto, o que envolve, entre outros aspectos, a topologia dos computadores conectados pela rede e a replicação dos dados entre eles. Por exemplo, enquanto jogos de luta dispõem os computadores numa topologia peer-to-peer, em que todos comunicam entre si, MMOs e *battle royales* que envolvem dezenas a centenas de jogadores em uma mesma sessão de jogo adotam a topologia cliente-servidor, em que todos se conectam com um servidor. [10][15]

Jogos digitais de luta corpo-a-corpo envolvem o confronto direto de dois ou mais personagens em variadas artes marciais em tempo real, e, devido a isso, requerem reações imediatas de ambas partes. Logo, organizadores e jogadores de partidas a nível competitivo se unem na busca de toda e qualquer opção que remova atrasos nas ações dos jogadores e no feedback visual. Notoriamente, jogadores profissionais de *Super Smash Bros. Melee*[1] jogam

[1] Nintendo, 2001

em TVs de tubo para tal fim, dado que não sofrem da mesma demora de 17 a 18 milissegundos para mostrar imagens que TVs digitais. [1] Tal gênero poderia ter sido o mais afetado pelo novo cenário decorrente da pandemia e perdido espaço em um mercado de jogos que está em constante fluxo, mas, ainda assim, manteve sua continuada relevância e comunidades dedicadas, e uma das explicações mais populares é que isso se deu pelo desenvolvimento anos antes de uma técnica de networking chamada rollback. [4]

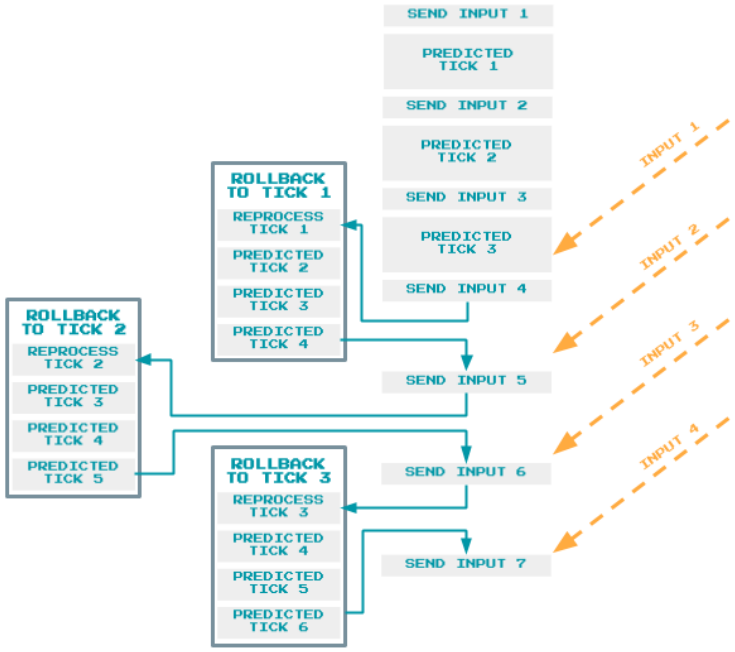

Figura 1: Ilustração do funcionamento do rollback. Ao receber de um computador remoto a ação (input) referente a um estado de jogo (tick, ou frame) anterior, esse estado de jogo é restaurado e reprocessado levando em conta as novas informações até chegar de novo na atual. Em processamentos que a ação remota ainda é desconhecida, realiza-se uma predição de qual pode ter sido a ação tomada a partir da extrapolação das anteriores. Fonte: [11]

Rollback é uma engenhosa técnica de networking peer-to-peer que busca criar a ilusão da completa ausência de latência. Seu nome é inspirado em sistemas de gerenciamento de base de dados, e se dá por restaurar um estado de jogo anterior para então reaplicar as ações realizadas localmente junto com as recebidas de outros computadores. (Fig. 1) [11] Assim, cada jogador pode reagir no momento devido em relação ao instante em que o adversário realizou as ações, sem precisar levar em consideração quando a informação chegou e, portanto, a latência entre eles. O modo que rollback sucede em mitigar o problema da latência o torna mais tolerante a falhas e eficaz que outra técnica de networking P2P chamada *delay-based*, como Huynh e Valerino (2019) demonstram em um estudo de caso. [15]

Entretanto a presença de tal método, conhecido por melhorar drasticamente a experiência de jogos online de ação, se mantém limitada a jogos de luta. Outros gêneros em tempo real como jogos de tiro e corrida implementam outras soluções que podem levar a modos diferentes de tratar a latência entre jogadores, mas são ou, de modo geral, heurísticas, ou não criam tão bem quanto o rollback a ilusão da partida acontecer localmente.

Por exemplo, jogos que usam a topologia cliente-servidor levam em conta que há uma disparidade entre a simulação no cliente (a qual o jogador pode reagir) e a no servidor (o *ground truth* da simulação compartilhada) devido à latência, que, caso seja tamanha pode resultar numa péssima experiência para o jogador. Devido a isso, alguns jogos como os feitos na game engine *Source* utilizam de um método chamado *lag compensation*[6], que tenta resgatar um estado passado da simulação para recriar o ponto de vista do jogador no próprio servidor e agir de acordo, no lugar do estado atual. Tal método é heurístico e atrai controvérsia, onde se acusa os jogos que o implementam de injustamente favorecerem jogadores com pior conexão. [17]

Este trabalho levanta a pergunta de pesquisa que um jogo que não é de luta, como, por exemplo, um jogo de tiro ou corrida, pode ter rollback como técnica de networking, como sugerido por Huynh e Valerino. [15] Entretanto, a investigação para respondê-la não se dará por tentar implementar a técnica em um jogo existente e já considerado *feature-complete*, o que é considerado custoso e demorado, [26] mas demonstrando que o rollback em si é generalizável seguindo uma linha teórica e dedutiva de raciocínio levando à incorporação do rollback ao próprio modelo de game loop, [33] que, sendo este aproximadamente o mesmo em todos estilos de jogo, abriria as portas para ser viável aplicar o rollback em vários jogos multijogador que possam se beneficiar dessas vantagens, justamente os de ação em tempo real, independente de que gênero específico sejam.

A linha a ser seguida será a de comparar o rollback com outras técnicas de networking existentes, assim elucidando as ressalvas intrínsecas à sua implementação e a que ponto podem possivelmente limitar as possibilidades de design de um jogo planejado desde o início

para comportar rollback. Sendo uma dessas ressalvas o determinismo da simulação do jogo, [4][7][11][15][26][29] também serão pesquisados fatores que impedem uma simulação de ser determinística e métodos para mitigá-los.

A pesquisa será realizada tanto na literatura formal quanto em sites de referência da indústria de desenvolvimento de jogos digitais como, por exemplo, o Game Developer (antigo Gamasutra), e em jogos que possam ser acessados dentro do período planejado. Por fim, seguindo uma abordagem empírica, será exemplificado o uso do rollback em um jogo concreto, no caso um jogo de tiro.

Este trabalho entrará mais a fundo no tópico de técnicas de networking e replicação, comparando vantagens e desvantagens entre topologias cliente-servidor e peer-to-peer, e em seguida, dentro de técnicas peer-to-peer relatará a evolução de técnicas que estabelecem lockstep determinístico entre jogadores até chegar no rollback.

Em seguida, abordará o tópico de game loop, [33] uma arquitetura que jogos em tempo real têm em comum, e o interpretará de uma forma que será argumentada como mais otimizada para rollback.

Adicionalmente, serão apresentados fatores que impedem um jogo de atingir determinismo, requisito vital para o rollback, como as variações em operações de ponto flutuante, o *streaming* de *assets* durante o decorrer do jogo, variáveis não-inicializadas e pontos de sequência.

Então, será relatado o processo de implementação de um jogo minimalista de tiro que seja jogado por dois jogadores através da rede, sobre o qual é aplicado rollback como técnica de networking, levando em consideração no desenvolvimento o que foi aprendido na pesquisa. Serão descritas as mecânicas de jogo implementadas e será relatado o processo de desenvolvimento e emprego das bibliotecas.

Finalmente, serão tiradas as conclusões do trabalho, e sugeridos temas de trabalho futuro.

2. Networking e replicação

A condição base para que jogadores conectados pela rede compartilhem um mesmo espaço virtual é que o estado de jogo esteja sincronizado de forma consistente entre eles. Tal é o objetivo de replicação, técnica adaptada de sistemas distribuídos (dentre os quais jogos em rede são um caso particular) que havia sido desenvolvida com objetivo de melhorar a tolerância a falha. [20]

Replicação vem em duas formas: ativa e passiva. Replicação ativa define que um pedido feito por um cliente seja recebido e processado por todos servidores do sistema. Entretanto, isso demanda que o processo sendo executado seja determinístico, ou seja, que dado um mesmo estado inicial e uma mesma sequência de pedidos a serem processados, o resultado seja sempre igual. [16]

Por outro lado, a replicação passiva emprega apenas um servidor chamado primário para receber e processar os pedidos do cliente e, então, repassar as atualizações pontuais feitas nos dados para os outros servidores. Determinismo não se torna mais necessário, mas, por sua vez, a resposta a possíveis falhas se torna mais lenta. [16]

Esses métodos são análogos respectivamente à replicação de jogos nas anteriormente introduzidas topologias peer-to-peer e cliente-servidor.

2.1. Cliente-servidor

Cliente-servidor se provou a solução predominante por diversas vantagens acima de P2P [20][29] como, por exemplo, comportar instâncias de jogo em que, virtualmente, não há limite de jogadores. A principal vantagem em relação a jogos peer-to-peer e à dispensa do determinismo é que o servidor pode enviar apenas frações do estado de jogo que são relevantes para o jogador, e assim não sobrecarregar nem a rede nem o poder de processamento do computador cliente.

No popular MMORPG *Final Fantasy XIV: A Realm Reborn*[2], áreas iniciais do jogo como a cidade portuária Limsa Lominsa costumam ser pontos de encontro e socialização e, portanto, constantemente populosas, mais do que é possível apresentar na tela todos os jogadores realmente presentes. A medida posta em prática é limitar os dados enviados pelo servidor para apenas um número arbitrário de jogadores que estejam mais próximos daquele cliente. Uma vez que se favorece constante performance, isso pode criar, particularmente em momentos de alta densidade populacional, a impressão que fora de um círculo ao redor do jogador, o mundo esteja vazio com exceção dos NPCs. (Fig. 2)

Figura 2: Congregação de jogadores em Final Fantasy XIV: A Realm Reborn, momentos antes do servidor sair do ar para uma manutenção de rotina. [27] Toda a área além desse círculo ao redor do cliente está cheia de jogadores na mesma instância, mas a partir de um limite alcançado não se replica jogadores mais distantes por questão de performance.

Essa oclusão de informação também é vital para garantir a performance em jogos onde combate ocorre esparsamente em áreas de jogo com quilômetros quadrados de extensão, por exemplo *battle royales* como *Fortnite Battle Royale*[3] e jogos de sobrevivência online como *DayZ*[4]. A posição de uma entidade que não esteja no campo de visão do jogador não será priorizada ou sequer enviada pelo servidor, assim como comandos para reproduzir efeitos sonoros podem ser omitidos dependendo do volume deles. O som de uma porta se abrindo não se replicaria depois de alguns metros, enquanto o som de um disparo de arma pode se

[2] Square Enix, 2013

[3] Epic Games, 2017

[4] Bohemia Interative, 2018

ouvir a quilômetros de distância, mesmo que nem o autor do disparo em si esteja sendo replicado. [32]

Entretanto, a demora na resposta a falhas da replicação passiva também se reflete na experiência de jogos cliente-servidor. O tempo de lag de um jogador é, em geral, o tempo que levará para que ele possa ver a consequência de suas ações. Formas de remediar isso incluem o já mencionado *lag compensation*, como também *client-side prediction*, que permite que o jogador consiga prever localmente como irá se comportar na simulação do servidor, e que na próxima leva de informações replicadas informações como sua posição se corrijam. [9] Eventos de divergência brusca, seja por alta latência ou perda de pacotes, são chamados de *warping* pelo modo que o jogador parece se teletransportar para a posição correta.

Outro problema encontrado principalmente em jogos de tiro é o *peeker's advantage* (vantagem do espreitador), em que, num impasse entre dois jogadores protegidos e escondidos por cobertura, o primeiro a sair e avançar sempre terá a vantagem de ser o primeiro a saber com precisão a posição do outro e agir em cima disso. [5] Por um tempo próximo à média aritmética de seus lags, o outro não terá como saber que o "*peeker*" avançou. Esse fenômeno é notado por influenciar estilos de jogo a serem mais agressivos, até em jogos que em seu design se planejam em torno de jogadas mais táticas e deliberadas.

2.2. Peer-to-peer

Duas consequências diretas do emprego da topologia P2P em relação à cliente-servidor são que a largura de banda da rede usada para comunicação de todos peers entre si cresce exponencialmente [32] à medida que mais peers são adicionados; e que devido à obrigatoriedade do determinismo, o estado de jogo deve ser armazenado e processado em sua totalidade por cada peer. Tais restrições certamente impediram que P2P fosse empregado em gêneros que evoluíram buscando acomodar mais e mais jogadores em sessões mais e mais extensas. [29] Em contrapartida, assim como a replicação ativa só necessita do envio de pedidos do cliente aos servidores, jogos P2P podem enviar entre si apenas as ações (inputs) dos jogadores. E dependendo do jogo, essa pode ser a única opção viável.

Dentre as técnicas de networking na topologia peer-to-peer está o rollback, que vem de uma "linhagem" de técnicas que evoluíram a partir do lockstep determinístico.

2.2.1. Lockstep

Jogos de estratégia em tempo real (RTS) geralmente exibem dezenas a centenas de unidades sendo constantemente atualizadas, e, portanto, a replicação de todas elas entre jogadores demandaria uma largura de banda que não poderia ser atendida [8][29] nos anos 90, quando os primeiros RTS em rede lançavam para o computador pessoal, como *Warcraft: Orcs & Humans*[5] [12] e *Command & Conquer*[6]. A solução para isso é que, a cada passo da simulação (chamado frame ou tick), ambos jogadores enviem apenas o input que realizaram naquele frame ou a mensagem informando que não agiram, e assim em ambos peers a simulação siga de acordo.

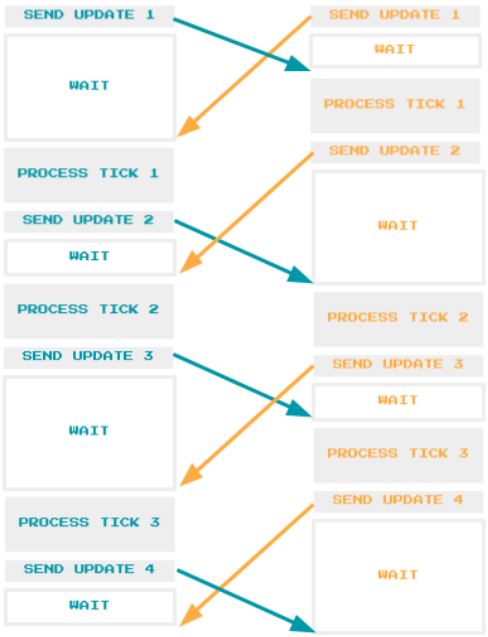

Figura 3: Ilustração do funcionamento do lockstep determinístico. Fonte: [11]

O jogo não procede em nenhum peer até que isso se confirme, e a simulação cobre sempre o mesmo tempo de jogo independente do tempo que passou parado. (Fig. 3) Este é o chamado método lockstep determinístico, [11] e, mesmo que seja considerado aceitável em

[5] Blizzard Entertainment, 1994

[6] Westwood Studios, 1995

RTS que promovem ações deliberadas em vez de reações rápidas, ou até mesmo jogos de luta em redes locais, que garantem medidas dispensáveis de lag, os constantes travamentos entre frames num cenário remoto causam uma grave degradação da experiência de jogo e da performance dos jogadores, que, em jogos de luta, dependem da familiaridade com a duração de cada golpe e habilidade.

2.2.2. Delay

A evolução direta do lockstep seria a de priorizar não travar o jogo, mas deixá-lo seguir normalmente enquanto os inputs são enviados ao outro peer para serem processados num frame posterior ao atual. Isso garante uma elasticidade ao diálogo entre peers, que não precisarão travar exceto em momentos de extremo lag. O número de frames atrasados é ajustado de acordo com o lag, logo, uma partida local não sofrerá com atrasos.

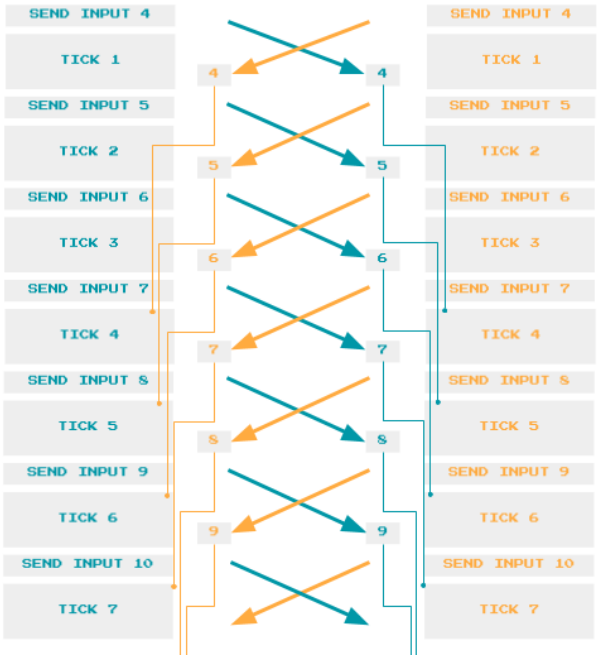

Figura 4: Ilustração do funcionamento do lockstep com atraso de input, ou delay-based. Fonte: [11]

Entretanto, com o atraso no processamento do input remotamente, o lockstep é preservado aplicando este mesmo atraso localmente e com o mesmo número de frames. Isso significa que cada jogador sentirá que todas as suas reações são tardias, ao menos que leve o atraso em conta. (Fig. 4) Este método, chamado lockstep com atraso de input [11] ou *delay-*

based (baseado em atraso) ainda é amplamente utilizado em jogos de luta, apesar de críticas à experiência de jogo ser comprometida em partidas que não sejam locais. [4][15]

2.2.3. Rollback

O passo seguinte seria, então, preservar a sincronia determinística dos peers e o não travamento no decorrer do jogo, e, ainda assim, processar os inputs no mesmo frame em que são realizados. A inspiração para tal solução, assim como a replicação, veio de bases de dados que restauram estados anteriores em caso de falhas na transação ou a pedido do usuário. [31]

A implementação define, então, como frame de confirmação o último em que havia sincronia entre os peers (isto é, o último em que os inputs de todos peers estavam registrados em cada computador, em vez de serem incógnitas) como o estado a ser restaurado pelo rollback, que ocorre ao receber inputs de frames anteriores ao atual dos outros peers.

Figura 5: Parte da interface no modo prática de Guilty Gear Strive (ArcSystem Works, 2021) que mostra na tela os inputs sendo executados pelo jogador. A alavanca virtual, mesmo recebendo entrada de uma alavanca analógica, é discretizada para ter apenas 9 estados: as 8 direções e uma posição neutra.

Enquanto esses inputs remotos são enviados, os do jogador local são processados sem atraso e enviados para os outros peers. Nos frames em que um input remoto é uma incógnita, de forma inspirada em outro método peer-to-peer chamado dead reckoning, o próprio middleware de rollback realiza uma chamada "predição", em que os últimos inputs conhecidos são extrapolados pressupondo-se que o estado está sincronizado enquanto o input remoto permanecer constante. [4] Tal método demonstra sua eficácia em jogos de luta, em que inputs são discretos, (Fig. 5) e o reportório de ações de cada personagem contém

sequências pré-definidas de direções da alavanca e botões que resultam em ataques especiais, chamados *command inputs*, mas não com inputs analógicos como o do movimento de um mouse em um jogo de tiro. O exemplo mais famoso de *command input* é o Hadouken, golpe dos personagens Ryu e Ken da série *Street Fighter*. (Fig. 6)

Figura 6: Command input do Hadouken, se o jogador está no lado esquerdo da tela. Se estiver do lado direito, os inputs são horizontalmente espelhados. O botão P representa um input de soco.

Ao serem recebidos, os inputs remotos a partir do último frame sincronizado são comparados com os extrapolados pela predição, e, em caso de qualquer divergência, todos frames a partir do de confirmação, são reprocessados até voltar ao atual. [11] Também pode ser armazenado um frame especulativo entre o de confirmação e o atual que, caso o input remoto divergente seja posterior a esse, garante um rollback mais curto. [26] O requerimento de repetir a simulação múltiplas vezes é inevitável, e se torna necessário otimizar o jogo por completo para que múltiplos frames de rollback caibam em um único frame, sem comprometer performance constante. [26] Nota-se que jogos de luta comumente executam 60 frames por segundo, ou seja, se espera que rollback ocorra dentro de 16 ms sem afetar a performance.

A Figura 7 demonstra o processo de um rollback a partir do momento que o input remoto atravessa a camada de rede e é recebido pelo peer local. Antes do rollback, todos frames entre o de confirmação *state(C)* e o atual *state(C+3)* haviam sido simulados com o input local e uma predição do input remoto. Quando o input remoto a ser simulado em C+1 é recebido, o estado de jogo é restaurado para *state(C)* e ressimulado levando em conta este input, com o frame resultante *state(C+1')* se tornando o novo frame de confirmação. Em seguida, sucessivas simulações com novas predições do input remoto levam o estado de jogo

de volta ao novo atual *state(C+3')*, que enfim é avançado uma vez para *state(C+4)*. Note que, para que não ocorra queda de performance, o rollback inteiro deve ocorrer entre *render(C+3)* e *render(C+4)*.

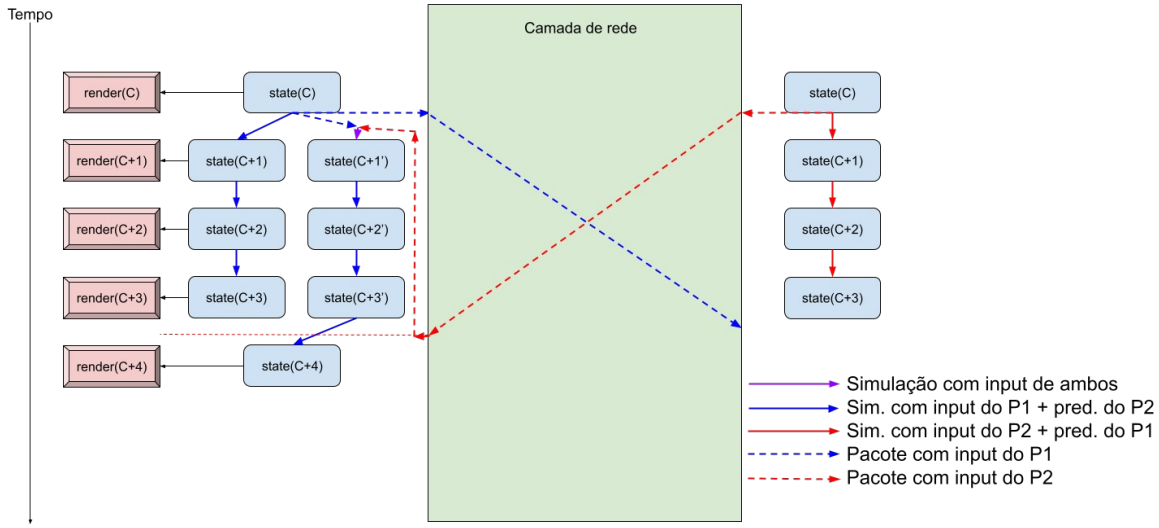

Figura 7: Ilustração mais detalhada do funcionamento de um rollback no peer P1. O mesmo processo acontece no peer P2, mas foi omitido por claridade.

3. Game loop e estado de jogo

Como já mencionado, a implementação de rollback ainda não é ubíqua em jogos de luta, com vários ainda sendo lançados com networking delay-based. Jogos já lançados comercialmente podem receber atualizações para passarem a comportar rollback, mas são tarefas custosas que envolvem reescrita de sistemas críticos do jogo. Seja por insuficiente compartilhamento de experiências na indústria [26] ou minúcias das circunstâncias de cada projeto, como game engine utilizada e particulares sistemas de jogo, implementar rollback em um jogo existente ainda é tratado como sendo caso a caso, sendo levadas em conta partes do código que não são determinísticas por não terem sido planejadas com esse requisito em mente, e partes que estejam muito acopladas ao código a ser repetido mesmo que não tenham efeito direto na simulação do estado de jogo.

O objetivo deste trabalho sendo o de demonstrar como rollback pode ser viável em gêneros além de jogos de luta, não será estudado um caso específico de implementação em um jogo existente, mas sim partindo de um ponto inicial comum. Esse ponto inicial é a própria arquitetura de game loop (Fig. 8), que está presente no núcleo de todos jogos digitais que apresentem imagem e/ou som constantemente atualizados em tempo real [33]. Neste capítulo, as funções que compõem um game loop serão reexpressadas de forma que, enquanto ainda podem comportar um jogo qualquer, podem também serem encaixadas em um modelo de rollback sem precisarem de qualquer alteração.

```
while true
{
    ProcessInput();
    Update();
    Render();
}
```

Figura 8: Pseudo-código do game loop, omitindo código que regula a frequência de atualização do jogo, na sintaxe da linguagem Rust.

Segundo Nystrom (2014), *ProcessInput()* representa o código que recebe do sistema operacional da máquina as ações realizadas através de periféricos como mouse, teclado e controles de jogo, e define os valores que ditam a influência do usuário no próximo passo da simulação. *Update()* representa um passo discreto da simulação, que atualiza o estado de jogo levando em conta o input processado. Em game engines que permitem que o jogo seja executado a uma frequência variável para acomodar computadores de variados poderes de processamento, também é levado em conta o tempo desde a última atualização, chamado tempo decorrido (elapsed) ou delta tempo [19] por ser a diferença entre os tempos dados pelo relógio do computador no frame atual e no anterior. Comandos para reprodução de faixas de áudio e feedback háptico também são comumente emitidos daqui. [33]

E, ainda segundo Nystrom, *Render()* transforma o estado de jogo em uma imagem a ser mostrada pelo monitor ligado ao computador. O estado de jogo não tem uma imagem inerente e depende de conter uma entidade câmera com posição própria, que "pintará" o que estiver em seu campo de visão em uma porção da tela, chamada *viewport*. Em jogos que comportam mais de um ponto de vista, geralmente jogos de multi-jogador local, há mais de um *viewport*, cada um atrelado a uma entidade câmera. Senão, espera-se que o único *viewport* da única câmera ocupe toda a tela.

Nota-se que as funções nesse pseudo-código não recebem argumentos nem retornam valores, suas chamadas acarretam apenas em efeitos colaterais. Isso vai de encontro à estratégia que será adotada neste trabalho em busca de determinismo e otimização do game loop, que será a de empregar funções puras[7] para representar a execução do jogo.

Se levadas em conta todas as variáveis que levam uma sessão de jogo para o mesmo fim, tais variáveis podem ser aplicadas novamente para alcançar o mesmo resultado, assim podendo afirmar que o jogo é determinístico. Um jogo pode permitir ao usuário configurar preferências de opções audiovisuais ou que afetam jogabilidade, que serão aplicadas em todas sessões futuras. O que também pode causar variações entre sessões são as regras e nível de jogo escolhidos para cada sessão. Tais fatores podem ser dados como argumentos de uma função pura (Fig. 9), dado que sendo os mesmos, darão no mesmo resultado.

[7] Funções puras são aquelas que retornam sempre os mesmos valores dado os mesmos argumentos recebidos, sem serem influenciadas nem resultarem em efeitos colaterais além de seu escopo local.

```
fn GameSession(
    gs: GameSettings,
    gr: GameRules,
    gl: GameLevel,
    inputs: Vec<Input>,
    deltas: Vec<DeltaTime>,
    cs: ComputerSpecs,
    seed: RNGSeed
) -> (
    FinalGameState,
    FramesOverTime,
    AudioOverTime,
    HapticsOverTime
);
```

Figura 9: Função pura que representa os fatores que podem afetar o resultado final de uma sessão de jogo finita. Pseudo-código na sintaxe da linguagem Rust.

Os seguintes argumentos são recebidos como entrada:

- *GameSettings* contém as preferências de usuário de áudio, vídeo, jogabilidade, idioma, etc.
- *GameRules* contém qualquer permutação das regras do jogo, como as escolhas de personagem de cada jogador, ou modificações de valores como dano ou gravidade.
- *GameLevel* é o nível escolhido para ser a arena da partida, moldando o estado de jogo com uma disposição própria de elementos e obstáculos.
- *Vec<Input>* é o vetor de inputs executados a cada frame pelos jogadores ao longo da sessão.
- *Vec<DeltaTime>* são os delta tempos aplicados em cada frame, que, segundo Fiedler (2014), junto ao input de usuário asseguram o determinismo da simulação quando levados em conta ao reproduzi-la. [7][23] Todavia, jogos em rede (especialmente em lockstep determinístico) demandam uma frequência constante, dado que o delta tempo não é replicado. [29]
- *ComputerSpecs* é um pseudo-argumento que representa as especificações da máquina, tanto em questão de peças de hardware quanto versões de sistema operacional e

drivers instalados. Essa variável não pode ser expressa em código real, entretanto influencia o resultado final.

- *RNGSeed* é o valor inicial do RNG, mais especificamente uma instância contida no estado do jogo que não é chamada por nenhum outro ponto do código senão a simulação. [23] Em algumas implementações, diversas fontes de aleatoriedade, como o tempo do relógio do computador, podem ser definidos como um novo seed durante a execução do jogo, mas neste caso isso mataria qualquer chance de alcançar determinismo, além de ser um efeito colateral que impediria a função de ser pura.

E os seguintes elementos são o resultado produzido pela função:

- *FinalGameState* é o estado do jogo quando a condição de fim de jogo é alcançada. Este é o único valor da saída que precisa ser sempre o mesmo, independente de ComputerSpecs, para alcançar determinismo *cross-platform* (através de múltiplas plataformas).
- *FramesOverTime*, *AudioOverTime* e *HapticsOverTime* são as saídas sensoriais do jogo, partes integrais da interação com o usuário, mas que não necessitam ser exatamente as mesmas em toda execução da mesma sessão em múltiplas plataformas, já que, como será desenvolvido a seguir, são apenas representações de cada passo da simulação. Essas são inevitavelmente influenciadas por *ComputerSpecs*.

3.1. Início de jogo e processamento de entrada

O primeiro passo da sessão é a transformação do nível, das regras de jogo e das preferências de jogabilidade no estado de jogo a ser simulado (Fig. 10). Segundo Stallone (2018), é vital para a otimização do jogo limitar o máximo possível a porção mutável do estado de jogo, para que apenas esta seja salva em frames de confirmação restaurados pelo rollback, em vez de junto à porção imutável. [26] As regras também definem a condição para o fim do jogo.

```
fn StartGame(
    gr: GameRules,
    gl: GameLevel,
    gameplay: GameSettings::GameplaySettings,
) -> (
    MutableGameState,
    ImmutableGameState,
    GameEndCondition
);
```

Figura 10: Função de início de jogo. Pseudo-código na sintaxe da linguagem Rust.

```
while !GameEndCondition()
{
    //game loop
}
```

Figura 11: Game loop envolto em uma condição de fim de jogo definida pelas regras de jogo. Pseudo-código na sintaxe da linguagem Rust.

A seguir, se inicia o game loop da sessão, envolto numa checagem do cumprimento da condição de fim de jogo (Fig. 11), significando que será repetidamente executado enquanto a condição de fim de jogo não for alcançada.

```
fn ProcessInput(
    raw: RawInput,
    control: GameSettings::ControlSettings
) -> Input;
```

Figura 12: Função de processamento de input. Pseudo-código na sintaxe da linguagem Rust.

O processamento de input (Fig. 12) deve levar em conta que, ao enviar o input para o outro peer, deve-se considerar que cada usuário pode configurar quais teclas são associadas a quais ações, seja por preferência, conforto ou acessibilidade. Portanto, uma boa prática é a de processar dados crus vindos dos periféricos em representações específicas do jogo (por exemplo “Jump” em vez de “Space”) para serem processadas na simulação local, enviadas para peers e armazenadas em arquivos de replay. [7]

3.2. Simulação

As diferenças entre um jogo digital ao qual se aplica um game loop e uma simulação de computador são, essencialmente, que, enquanto uma simulação é executada do começo ao fim tomando como entrada apenas o estado inicial e um predeterminado número de passos, um jogo também leva em conta o input do usuário a cada passo; e que, enquanto uma simulação pode apresentar apenas seu estado final, seja por texto ou gráficos, um jogo deve apresentar o estado de jogo a cada passo. [33] Uma simulação que envolve passagem do

tempo pode, assim como um jogo, receber como argumento o delta tempo desde o passo anterior, e pode reproduzir o mesmo resultado desde que seja determinística e armazene o delta tempo de cada passo.

```
fn Simulate(
    previousMutState: MutableGameState,
    immutState: ImmutableGameState,
    input: Input,
    delta: DeltaTime
) -> MutableGameState;
```

Figura 13: Função de simulação. Pseudo-código na sintaxe da linguagem Rust. Como já mencionado, o delta tempo deve se manter constante em lockstep determinístico, dado que não é replicado.

O passo de simulação (Fig. 13) é o que é executado múltiplas vezes no evento de um rollback, [26] portanto, o mais importante no processo de otimização para acomodar a técnica, e também é o passo do game loop em que determinismo é obrigatório. [29] É ingênuo pensar que apenas pelo código ser organizado assim o torna determinístico, portanto, no capítulo seguinte serão vistos os fatores que levam as diferenças de hardware e software a impedirem determinismo durante a execução de um jogo.

3.3. Apresentação

Por fim, o passo de apresentação (Fig. 14) deve conter todo tipo de processamento que não serve no passo de simulação por não ter influência no valor do estado mutável ao fim do frame e, portanto, ser redundante em um rollback. Isso inclui a renderização do frame como também a reprodução de áudio e feedback háptico.

```
fn Present(
    mutState: MutableGameState,
    immutState: ImmutableGameState,
    viewports: Vec<Viewport>,
    video: GameSettings::VideoSettings,
    audio: GameSettings::AudioSettings,
    language: GameSettings::LanguageSettings
) -> (
    Frame,
    Audio,
    Haptics
);
```

Figura 14: Função de apresentação. Pseudo-código na sintaxe da linguagem Rust.

Figura 15 mostra a reorganização desses passos em um game loop de rollback. Nota-se que um frame de confirmação pode ser restaurado antes de um ser salvo, isso se dá pois o estado inicial é também o primeiro frame de confirmação, já que não há nenhum input anterior não-confirmado. O passo de receber inputs para o frame a ser simulado pode, junto ao input local que foi registrado para aquele frame, ou retornar os verdadeiros inputs remotos recebidos, ou a predição destes por extrapolação dos anteriores. Nota-se, também, que este modelo assume o armazenamento de apenas um frame para ser restaurado, o frame de confirmação. Outras implementações, como a do GGPO, [14] podem salvar mais de um frame a fim de restaurar o anterior mais próximo ao input remoto recebido em vez de um mais distante.

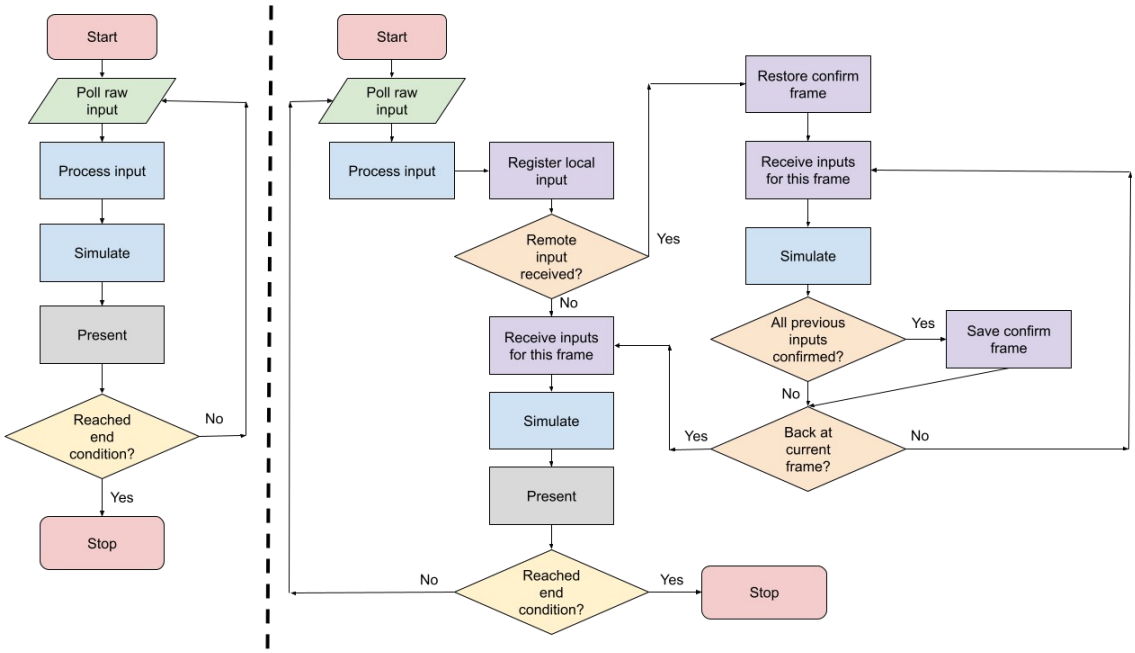

Figura 15: À esquerda, fluxograma de um game loop normal. À direita, fluxograma de um game loop de rollback.

3.4. Simulação primária e secundária

No processo de otimização da simulação, deve-se também considerar quais partes do processamento influenciam ou não no resultado final do jogo. [29] As que não influenciam, mas ainda processam parte do estado mantido entre frames, geralmente se referem a elementos da apresentação, como sistemas de partículas. Stallone (2018) dá o exemplo de, enquanto parte da equipe encarregada de implementar rollback em Mortal Kombat X[8],

[8] Warner Bros. Interactive Entertainment, 2015

desabilitar simulação de tecidos (um de vários subsistemas presentes em engines de física) entre outros, justamente por, no caso específico do jogo em questão, não afetar jogabilidade. [26]

Dado que os métodos para alcançar determinismo podem vir em troca de performance, [8] é interessante que tais simulações não sejam abrangidas pelo determinismo, mas que estejam propriamente desacopladas das mais "sensíveis", que afetam jogabilidade e, portanto, devem se manter determinísticas. [8] Essas simulações sensíveis serão aqui chamadas de simulações primárias, e as demais, secundárias.

Entretanto, por não serem determinísticas, as simulações secundárias e a porção do estado de jogo regido por elas, ao terem um estado anterior restaurado e reprocessado, podem resultar em descontinuidades visuais. [26] Stallone também descreve a solução aplicada em Mortal Kombat X (e posteriormente em Injustice 2[9]) em que caches dos estados de partículas eram salvos e comparados com o cache calculado após o rollback, e, caso fossem similares o suficiente, seriam mantidos em vez de ressimulados.

Especialmente partículas computadas pela GPU, assim como reprodução e mixagem de áudio pelos drivers e hardware, são não-determinísticas por natureza e exteriores ao estado de jogo mutável, portanto, não podem nem devem ser restauradas pelo rollback. Pelo contrário, para evitar que uma emissão de partículas fique constantemente "piscando" ou que o áudio tenha "estouros", também foram abrangidos pela solução descrita por Stallone.

[9] Warner Bros. Interactive Entertainment, 2017

4. Demais fatores que impedem determinismo

Segundo Chance et al (2022), “enquanto game engines são planejadas primariamente para performance a fim de alcançar uma boa experiência de usuário, os requerimentos para verificação de veículos autônomos vão além disso e incluem determinismo (tradução nossa)”. [3]

Se, numa tentativa de implementar rollback ou lockstep, não há determinismo *cross-platform* na simulação, mesmo que dois peers partam de um mesmo estado de jogo inicial e compartilhem entre si os mesmos inputs, à medida que o jogo progrida a menor divergência entre os estados desencadeará divergências ainda mais graves que invalidam a sessão de jogo a partir daquele ponto, chamadas de dessincronização. Uma prática comum para detectar essas divergências é comparando *checksums* do estado de jogo em intervalos constantes, mas encontrar a causa é consideravelmente mais difícil. Entretanto, há caminhos que podem ser percorridos para preemptivamente impedir dessincronização. [29]

4.1. Ponto flutuante

Fiedler (2010, 2014) descreve o emprego de números de ponto flutuante[10] em cálculos de simulação de física como obstáculo para o determinismo da operação. Diferenças entre compiladores, arquiteturas com diferentes conjuntos de instruções, ou até mesmo as diferenças entre otimizações em builds debug e release do mesmo código podem levar a divergências nos resultados de cálculos entre números de ponto flutuante. [7][8]

Ainda de acordo com Fiedler (2010), um cálculo de ponto flutuante ainda é determinístico se o mesmo executável, vindo de um mesmo compilador, é rodado em computadores com a exata mesma arquitetura e aplicando “truques” específicos à plataforma, [8][29] sendo que isso ainda não consta como cross-platform, mas pode ser o suficiente se o jogo for exclusivo a um único console de videogame. [23]

[10] Valores não-inteiros em que o ponto decimal pode mudar de posição (“flutuar”) dependendo da ordem de magnitude do valor, para garantir maior precisão a valores menores.

> Contudo, com bastante trabalho pode ser que você consiga persuadir diferentes compiladores ou diferentes arquiteturas de máquina a terem exatamente os mesmos resultados de ponto flutuante usando o modo “estritamente” compatível com IEEE 754 de seu compilador e restringindo o conjunto de operações de ponto flutuante que você pode usar. Isto tipicamente resulta numa performance consideravelmente pior com ponto flutuante. (FIEDLER, 2010, tradução nossa)

Outra opção seria substituir pelo emprego de matemática de ponto fixo. [18][29]

4.2. Streaming de assets

Mundos bastante extensos, assim como vários tipos de dados como modelos 3D, materiais, efeitos sonoros e animações, podem não caber por inteiro na memória e, portanto, áreas distantes o suficiente do jogador e os demais dados aos quais o jogador não poderá reagir ou presenciar em certo momento durante a simulação ficam fora do runtime até serem requisitados pela simulação para serem carregados, enquanto outros liberam seu espaço na memória quando deixam de ser necessários. [22] O tempo deste carregamento depende da velocidade de acesso da unidade de armazenamento de dados, seja um disco rígido, um SSD ou outro, e espera-se que seja imediatamente adicionado ao estado de jogo, o que pode variar de um computador para outro, mesmo com arquiteturas iguais. Jogos de luta predominantemente estão contidos em arenas fechadas que são carregadas por completo antes do início de jogo e, portanto, não lidam com isso.

4.3. Variáveis não-inicializadas

Por questões de segurança, quando uma região da memória do computador é alocada para a execução de um programa, ela é inicializada por completo com valores padrão pelo sistema operacional, a fim de evitar que dados do programa a ocupar anteriormente aquela região sejam acessados. Entretanto, isso determina o valor de uma variável não-inicializada apenas quando ela aponta para uma posição que não foi previamente escrita durante a execução do mesmo programa, que dinamicamente aloca regiões no heap e stack à medida que forem requeridas durante a execução do programa. [21]

Nem toda linguagem de programação permite a leitura de variáveis não-inicializadas, mas dentre as que permitem está C++, que é predominante no desenvolvimento de jogos.

Deve-se notar também que a linguagem não define padrões para valores pré-inicializados, o que fica por conta apenas do sistema operacional, tal fator impedindo o determinismo cross-platform até mesmo das primeiras variáveis a apontarem para posições da memória. [21] A solução para prevenir qualquer quebra de determinismo em decorrência disso é tão simples quanto inicializar todas variáveis ao serem declaradas, principalmente as que fazem parte do estado de jogo. [23][29] Tal qual pôr o compilador em um alto nível de aviso, ferramentas como Valgrind [28] podem apontar variáveis não-inicializadas.

4.4. Pontos de sequência

Ao escrever um programa de computador, espera-se que o código seja executado estritamente na ordem em que foi escrito, com a exceção de operadores de maior precedência em uma mesma expressão. Entretanto em C e C++, esse não é precisamente o caso, dado que apenas certos operadores são considerados pontos de sequência, isso é, pontos do código que asseguram que todas expressões anteriores e seus efeitos colaterais sejam avaliadas por completo. [2] Naturalmente, o operador de fim de *statement* (;) é um ponto de sequência, assim como os operadores lógicos (&& e ||), dado que, dependendo do resultado do operando esquerdo, o direito não precisará ser avaliado.

```
int RNG_GetNext(int* seed);
int Function(int a, int b);

int main()
{
    int seed = 0;
    int output = 0;

    //deterministic
    int randA = RNG_GetNext(&seed);
    int randB = RNG_GetNext(&seed);
    int value = Function(randA, randB);

    //non-deterministic
    int value = Function(RNG_GetNext(&seed), RNG_GetNext(&seed));
}
```

Figura 16: Exemplo de como pontos de sequência podem afetar determinismo. Adaptado de [29].

Entretanto, o operador vírgula que separa os argumentos de uma função não é especificado como ponto de sequência, e portanto diferentes compiladores podem

possivelmente avaliar a expressões dos argumentos em ordens diferentes, e dependendo dos efeitos colaterais resultantes de cada expressão, o determinismo é perdido. (Fig. 16) [29]

5. Projeto de Implementação

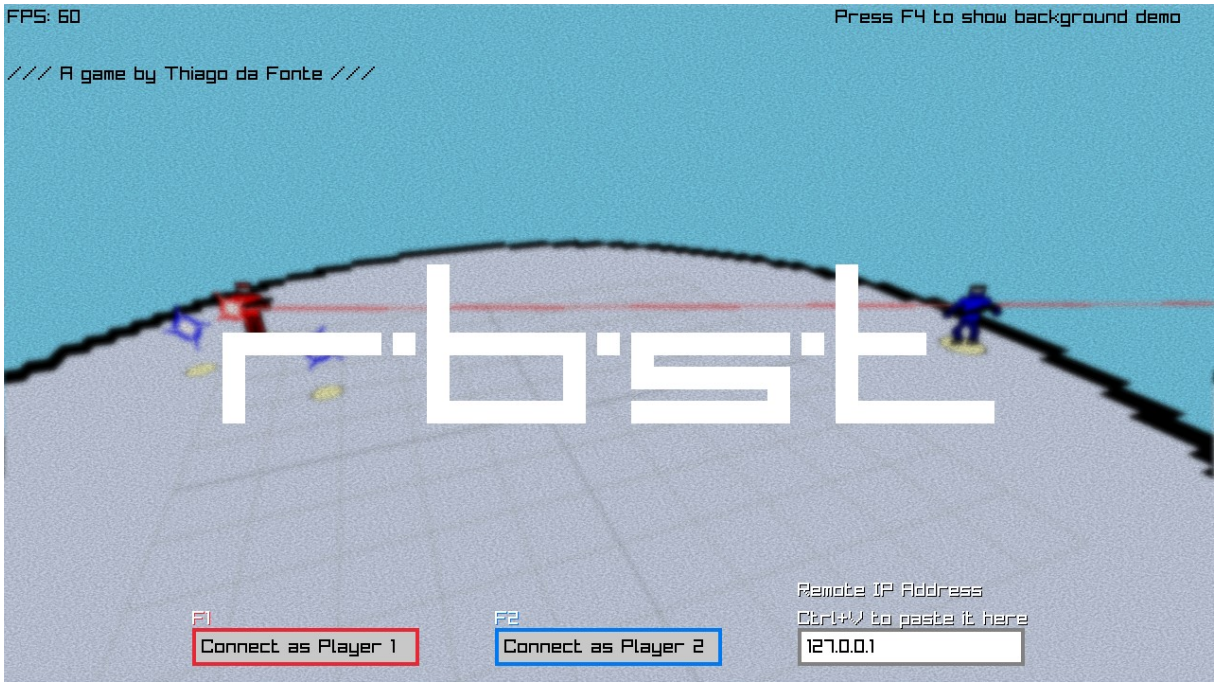

Figura 17: Tela inicial do jogo do projeto de implementação, RBST (iniciais das sílabas de "Rollback Shooter"). No pano de fundo, o replay de uma partida anterior é reproduzido a partir de condições iniciais e inputs dos jogadores a cada frame.

O gênero de jogo a ser abordado é o do jogo de tiro, que é definido por combate à longa distância em grande parte com armas de fogo reais ou fictícias. Durante o desenvolvimento de partidas em rede do influente *Doom*[11], o lockstep determinístico na topologia peer-to-peer foi inicialmente considerado, mas por fim descartado, dado que os travamentos para sincronização dos peers iam de encontro ao ritmo frenético, e em seu lugar foi adotada a topologia cliente-servidor. Essa abordagem continuaria em seu sucessor *Quake*[12], que foi vital para a popularização de jogos de tiro em rede estreando avanços tecnólogicos como o *client-side prediction*. [9]

[11] id Software, 1993

[12] id Software, 1996

Hoje em dia, jogos de tiro são predominantemente cliente-servidor, dentre eles os mais jogados competitivamente, como *Counter-Strike: Global Offensive*[13], *Valorant*[14], *Fortnite Battle Royale* e *Apex Legends*[15]. Esses dois últimos, sendo *battle royales*, desfrutam da capacidade de cliente-servidor de comportar altos números de jogadores. Por outro lado, as cenas competitivas mais fortes a surgir da série *Quake* e de jogos similares, denominados *arena shooters*, são as de duelos entre apenas dois jogadores. [24] Entretanto, novos *arena shooters* ainda são desenvolvidos como cliente-servidor, sem levar em consideração que o estado da arte de jogos P2P é bem diferente de quando o primeiro *Quake* foi lançado.

Partindo disso, foi desenvolvido um *arena shooter* com rollback como técnica de networking. Tem escopo minimalista para que possa ser concluído dentro dos prazos da disciplina sem perder os traços que definem um jogo de tiro de PC, como mirar com o mouse e se locomover em qualquer direção.

5.1. Documento de design do jogo

Seguindo o preceito de minimalismo, a área de jogo se limita a uma arena circular em um único plano, sem obstáculos, e, similar a *Wolfenstein 3D*[16], os jogadores podem olhar para os lados, mas não para cima ou para baixo. Em vez de modelos 3D, os personagens e demais entidades de jogo são representados por imagens 2D, ou sprites, e suas áreas de colisão são círculos no chão que representam cilindros de mesmo raio que envolvem as entidades. A perspectiva escolhida é a em terceira pessoa (em que a câmera é exterior ao personagem) em vez de primeira pessoa (em que a câmera são os olhos do personagem) para que o jogador tenha uma visão clara dos círculos no chão. (Fig. 18)

Já em um aspecto mais similar a jogos de luta, partidas são divididas em rodadas, em que cada jogador inicia em um ponto fixo da arena com um número pré-definido de pontos de vida (HP) e que deve deduzir o HP do outro jogador a zero para vencer a rodada, ou que, ao

[13] Valve Corporation, 2012

[14] Riot Games, 2020

[15] Electronic Arts, 2019

[16] id Software, 1992

chegar no tempo limite da rodada, ser o jogador com mais HP. (Fig. 19) As condições para empate são que, no tempo limite, ambos tenham o mesmo número de HP; ou que percam seu último HP no mesmo frame.

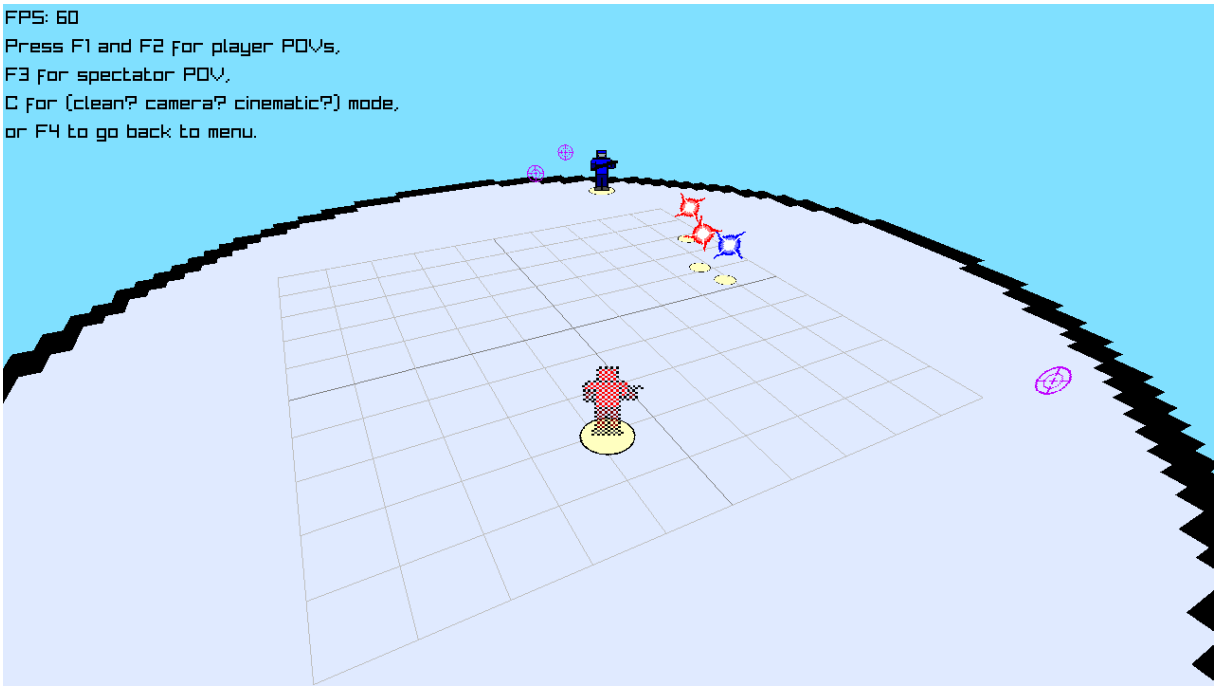

Figura 18: Replay de partida passada na perspectiva do jogador 1.

Figura 19: Interface gráfica no início de uma partida de RBST, em que ambos jogadores começam a rodada com 5 HP, e devem vencer 2 rodadas para vencer a partida.

Cada jogador tem acesso a dois recursos que iniciam em seu valor máximo, e recarregam constantemente entre usos: *ammo* (munição) e *stamina* (energia, vigor). *Ammo* é gasto com *shots* (tiros) e *rails*, e *stamina* é gasto com *dashes*.

Shots disparam projéteis que se deslocam numa velocidade linear e constante, tendo a mesma cor de quem disparou, e podendo colidir apenas com o adversário. *Rails*, por outro lado disparam raios que atingem instantaneamente tudo que fizer interseção com uma linha reta partindo de quem disparou. Entretanto, ao executar um input de *rail* tendo *ammo* suficiente para tal, o jogador fica imóvel enquanto o disparo carrega, revelando a trajetória no

chão com faixas de aviso, que convergem para explicitar o instante em que o rail será disparado. (Fig. 20)

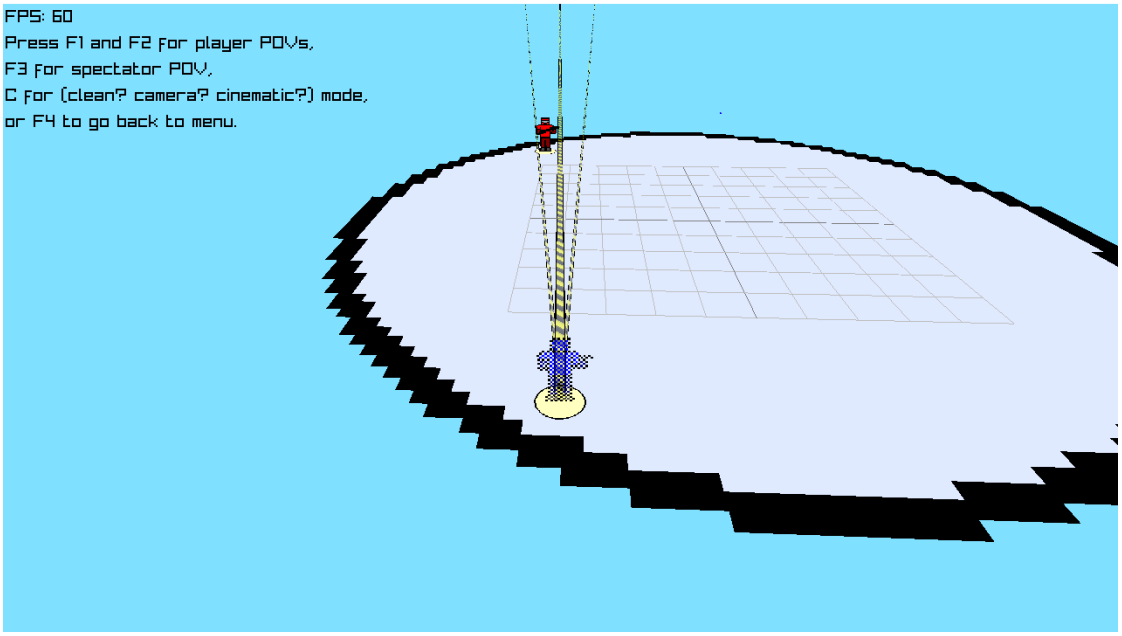

Figura 20: Replay na perspectiva do jogador 2, que está carregando um rail.

Inspirado no *arena shooter Unreal Tournament*[17], cuja arma *ASMD Shock Rifle* funciona de maneira similar, quando um *rail* atinge um projétil, independente do "dono" de cada, causa uma grande explosão em torno do projétil, chamada combo, que pode infligir dano em ambos jogadores se estiverem dentro da área de impacto. (Fig. 21) Para compensar o tempo de carga, uma projeção da posição futura de cada projétil é mostrada na tela desde que ainda esteja dentro da arena (os alvos roxos na Figura 18), e se quando há *ammo* o suficiente para um *rail*, também é mostrada a área de impacto do possível combo. (Fig. 22)

Como forma de risco e recompensa, estar perto o suficiente de um *rail* adversário sem ser atingido (como o jogador 1 na Figura 20) recarrega por completo *ammo* e *stamina*. Essa mecânica é chamada *graze*, e o feedback visual que o jogador realizou um *graze* com sucesso é a emissão de várias partículas de cruzes verdes a partir dele.

[17] Epic Games, 1999

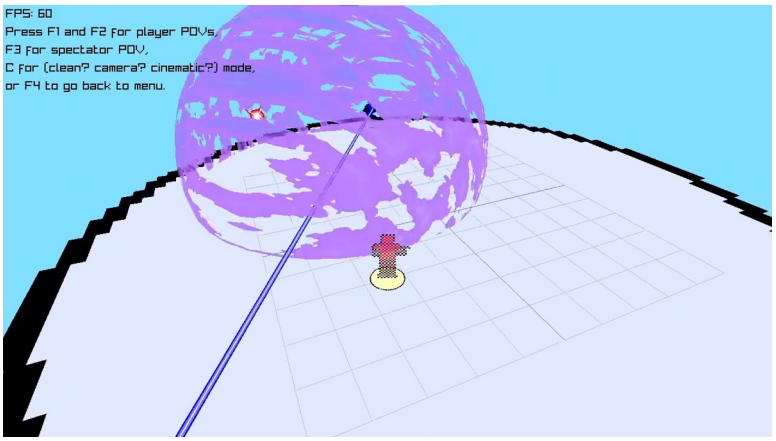

Figura 21: Combo a partir de um rail do player 2.

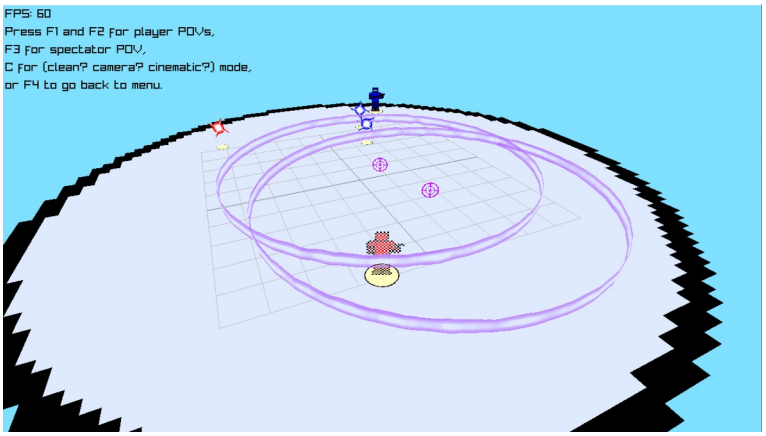

Figura 22: Projeção de combo em cada posição futura que ainda esteja dentro da arena.

A locomoção normal do jogador é deliberadamente escorregadia e tem baixa velocidade máxima. Em contrapartida, o *dash* é um deslocamento rápido e curto em uma direção, e é tanto uma ação ofensiva quanto defensiva. Quando é ofensiva, em que o jogador executa um *dash* em direção ao adversário, é um *tackle*; quando é defensiva, em que o jogador esquiva de ataques, é um *evade* (esquiva). Dependendo do quão perto o jogador estava de ser atingido, o *evade* pode rebater de volta qualquer ataque: projéteis (que trocam de "dono" e ficam mais rápidos), *rails* (Fig. 23) e *tackles*.

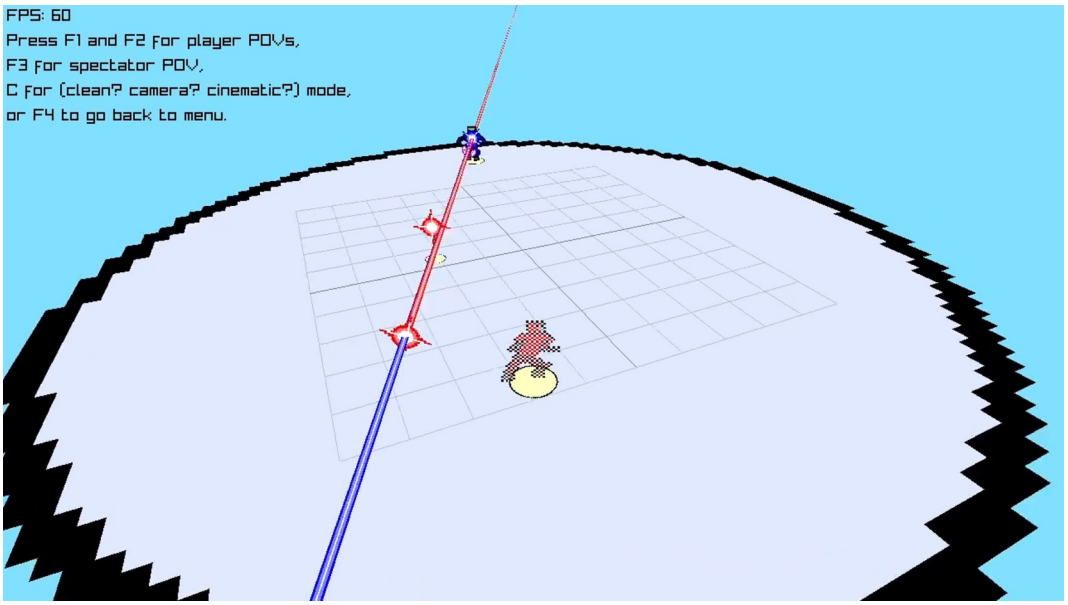

Figura 23: Jogador 1 rebatendo um rail com um evade.

Quando um jogador leva dano, ele fica imóvel por um número de frames relativo à força do impacto (fraca para projéteis, média para *rails* e *tackles*, e forte para combos), e então passa alguns instantes *stunned* (atordoado) sem poder reagir, e em risco de receber ataques consecutivos. A única coisa que pode tirar o jogador do *stun* previamente é um *dash*, desde que tenha *stamina* o suficiente, mas em troca a *stamina* é reduzida a zero independente do valor anterior. Essa mecânica é o *alert*, e é representado por partículas de pontos de exclamação vermelhos emitidos do jogador.

5.2. Relatório de desenvolvimento e emprego das bibliotecas

O projeto foi desenvolvido em C++ no Visual Studio, e dispôs de bibliotecas *open source* nas licenças MIT e zlib.

5.2.1. Raylib

Game engine que opera gráficos, áudio, input, intervalos de tempo entre frames e o tempo de vida da janela do programa. [25] O maior benefício do uso dessa engine em relação a outras é que a renderização é completamente imediata em vez de retida, ou seja, a renderização de sprites e modelos se dá por sequências de comandos executados a cada frame, e não pela instanciação de objetos que são posicionados a cada frame e correm o risco de ficarem sem referência que aponte para eles, o que os tornaria em ruído na cena de jogo.

A engine também provê *shaders* de vértice e fragmento padrões, para que apenas um ou outro seja substituído por um criado pelo desenvolvedor em GLSL. Os *shaders* personalizados criados para este projeto estão presentes no efeito aplicado no combo e na projeção dele (que usam o mesmo *shader* com diferentes valores), no borrão e ruído aplicados no replay que é reproduzido de fundo na tela inicial, no descarte de pixels transparentes de sprites, e na rolagem de textura da faixa de aviso do *rail*.

Reprodução de áudio não foi implementada neste jogo.

5.2.2. GGPO

Biblioteca desenvolvida em 2009 pelos pioneiros do rollback, que se manteve proprietária e fechada até 2019, quando foi lançada como *open source* sob a licença MIT para que fosse livremente usada por desenvolvedores independentes. [14] Apesar da licença MIT permitir modificação do código fonte, a biblioteca foi aplicada *as-is* neste projeto por conta do prazo da disciplina, e por ser a maior validação da hipótese do trabalho que, mesmo não modificado, um middleware planejado para jogos de luta pôde ser aplicado em um jogo de tiro que atendesse as necessidades certas.

A “predição” de inputs remotos inevitavelmente causa divergências e resulta em rollbacks em todo frame com o mínimo movimento de mouse. Porém, assim como no desenvolvimento dos jogos apresentados por Stallone, basta otimizar visando sempre o pior caso, pois assim, os casos mais comuns, que em geral estão mais perto do ideal, terão uma experiência substancialmente melhor. [26]

Uma peculiaridade da extrapolação dos inputs remotos foi a de que, por esperar que o input de cada ação seja representado por um stream de dois estados (botão pressionado ou não), foi de encontro ao processamento de input, que trata as 3 ações dos jogadores como um único stream de eventos pontuais, mas de maneira imperceptível. A extrapolação repete um evento nos frames seguintes, executando *dashes* que são instantaneamente interrompidos pelos seguintes até a *stamina* ser gasta, ou atirando projéteis que estão próximos demais para

serem distinguidos até a *ammo* ser gasta. Este problema é sempre resolvido pela chegada de outros inputs antes que fosse sequer notado durante partidas, e foi apenas deduzido que acontecia, entretanto torna-se, então, desejável uma maneira de manipular a extrapolação além de apenas "repetir dados anteriores".

Dentre os eventos que o objeto *GGPOSession* pode lançar está a notificação que um peer está *n* frames adiantado em relação ao outro, o que pode criar uma vantagem injusta. A forma de tratamento escolhida foi, em vez de chamar o método *Sleep* da API do Windows para congelar o jogo por *n**16 milissegundos, (o que poderia gerar conflito com o controle de taxa de frames do Raylib) é enviado ao Raylib um comando de diminuir a taxa de frames para 50 FPS por n*5 frames, o que é consideravelmente menos perceptível. [26]

GGPO mantém salvos os estados de jogo de múltiplos frames, mas curiosamente pode restaurá-los fora de ordem (restaurando um frame mais velho que o do rollback anterior), ou ainda executar mais de um rollback no intervalo de um mesmo frame, ao que deve, afinal de contas, depender da ordem de chegada dos inputs remotos, já que são enviados por UDP. A verdade é que GGPO não deixa transparente qual o frame salvo mais velho, e portanto o frame de confirmação. Isso gerou complicações na implementação de replays e correções de partículas da simulação secundária, e, por fim, foi definido que manteriam uma janela de 15 frames anteriores ao atual, o que a 60 FPS, corresponde a um lag de 500 ms[18].

Indo mais a fundo na relação entre simulações primária e secundária, são armazenadas "consequências" da primária se associam cada uma a emissões de partícula da secundária. Após um rollback, as consequências de frames ressimulados sobrescrevem as anteriores daqueles mesmos frames, e as partículas que estão associadas a consequências sobrescritas, isso é, "em fluxo", checam se a consequência ainda existe e se teve algum valor mudado, sendo então apagadas se a consequência não existe mais, ou tendo a posição ajustada. Finalmente, consequências que ainda não haviam partículas associadas a elas resultam em novas partículas emitidas no frame atual. (Fig. 24) Tal solução é cabível também para reprodução de efeitos sonoros sem o risco de estouros de áudio. Infelizmente, sem uma

[18] (1000/60) ms por frame * 15 frames * 2 (lag é o RTT, a ida e volta) = 500 ms

maneira de identificar as entidades que causaram a consequência, a associação só pode se dar por métodos heurísticos, como o implementado de checar posições próximas.

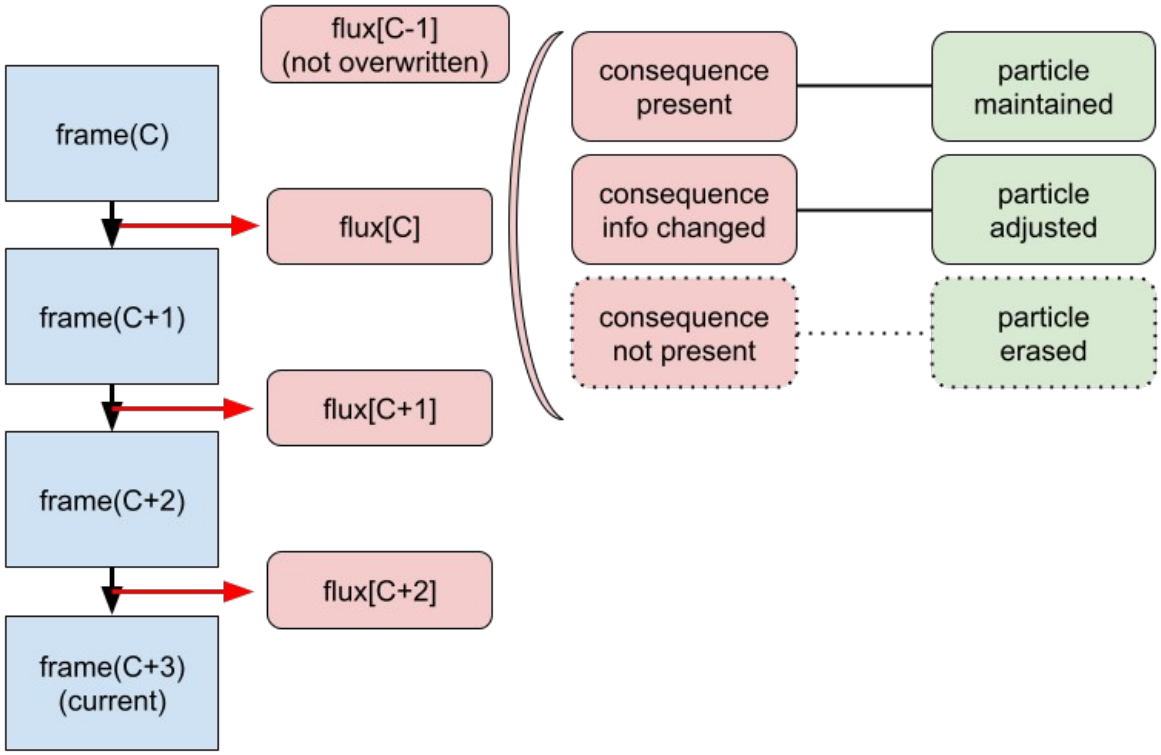

Figura 24: Ilustração do sistema de "consequências em fluxo" implementada no projeto

Vale notar que sessões de GGPO definem de antemão número de jogadores e até de espectadores (peers que recebem estado de jogo mas não enviam input), e têm tratamento para desconexões durante a sessão mas não para conexões novas. Logo, partidas em andamento não podem adicionar novos jogadores. Essa desvantagem em relação a jogos cliente-servidor, em que é simples um novo jogador se conectar ao servidor e começar a receber estado para enviar input, é conhecida, mas não é mitigada devido ao foco em jogos de um contra um.

E não menos importante, o método do GGPO (e de diversas outras implementações de rollback) de salvar e restaurar estados de jogo é por serialização, o que significa que o objeto dado como estado de jogo deve manter sua integridade mesmo quando transformado em uma sequência de bytes e de volta em um objeto. Isso implica, na prática, em não guardar partes do estado de jogo em ponteiros para objetos alocados na heap, já que isso aumentaria

desnecessariamente a complexidade do gerenciamento de estado de jogo, afetando performance, além de criar riscos de *memory leak*. Dado que, ao longo do desenvolvimento de um jogo não planejado para rollback, o estado de jogo pode acabar sendo armazenado em uma complexa estrutura de dados, é compreensível que seja um esforço hercúleo retroativamente implementar a técnica, juntamente a reescrever partes do código de simulação que não são determinísticos.

5.2.3. fpm

Biblioteca que provê tipos numéricos de ponto fixo e cálculos entre eles, em que resultados são salvos em valores intermediários de maior precisão antes de serem truncados de volta. [18] Todos valores decimais do estado de jogo foram armazenados neles, assim como o input de mouse, que, mesmo que seja recebido da engine em valores de ponto flutuante, se torna um dado próprio para simulação, replicação e armazenamento de replays ao ser convertido para ponto fixo.

Junto com a biblioteca abaixo e seguindo os critérios do capítulo 4, o passo de simulação, que foi o primeiro a ser escrito, não precisou ser refatorado em nenhum momento por ter tido falha de determinismo.

Entretanto, fazer cálculos consecutivos em que o valor intermediário é truncado múltiplas vezes causou imprecisões (não de determinismo, já que ainda teriam o mesmo resultado em cada peer, mas de desvios do verdadeiro valor matemático) que poderiam ser evitadas com maior entendimento do funcionamento da biblioteca em vez de se limitar ao uso básico. Em uma parte do código, o valor de um vetor 2D havia se tornado imprevisível ao menos que fosse normalizado a cada frame.

5.2.4. Embedded Template Library

Implementação da Standard Template Library que visa o *deployment* em sistemas embarcados, provendo versões com comportamento determinístico e tamanho fixo de containers. [30] Foram aplicados os containers *vector* para conter os projéteis em jogo, e

stack para implementar o autômato de pilha de cada jogador. O emprego desses containers em vez dos da STL garantiu que o tamanho do objeto GameState fosse estático.

5.2.5. TOML++

Parser de arquivos de configuração no formato TOML. [12] Enquanto *RBST_home.toml* e *RBST_controls.toml* agem como substitutos de menus de configuração para economizar tempo de desenvolvimento em troca de um leve deficit de usabilidade, *RBST_config.toml* foi criado para rápida iteração dos parâmetros de jogo sem precisar recompilar o jogo por completo.

Quando replays foram implementados com os valores de configuração ainda em fluxo, invalidando replays passados, foi percebido que deveriam ser tratados como a parte imutável do estado de jogo, e assim foram salvos como cabeçalhos do arquivo de replay. Assim, *RBST_config.toml* continua como um arquivo modificável do jogo, mas que deve ser compartilhado manualmente entre jogadores para evitar dessincronizações.

6. Conclusões e Trabalhos Futuros

Este trabalho abordou os diferentes métodos de sincronização contínua de estado de jogo para viabilização de jogos em rede, e aos compará-los com o networking de rollback, foram levantados prós e contras de implementá-lo, sob a hipótese que, apesar de ser pensado para jogos de luta, é aplicável em outros gêneros sem deixar de colher os mesmos benefícios. Em vez de tentar encaixar esses outros gêneros no molde de jogos de luta, buscou-se elucidar o que é realmente intrínseco a jogos de luta que comporta rollback, e como aplicá-lo em outro gênero. Essa pesquisa se frutificou em um jogo que apesar de ser de tiro em terceira pessoa, tem rollback como método de networking.

6.1. Contribuições

Se determinou que o rollback, por ser aplicado em uma topologia peer-to-peer, não pode suportar um grande número de computadores conectados pois o consumo de banda cresce exponencialmente, e que requer que cada computador contenha o estado de jogo por inteiro. E que o rollback, como evolução do lockstep determinístico, requer não apenas determinismo, mas determinismo cross-platform para ser aplicado em um jogo de PC ou multi-plataforma. Isso implica diversos fatores:

- Frequência de simulação constante, ou seja, sem influência do tempo passado entre frames;
- Severo escrutínio ou completa ausência de operações de ponto flutuante em partes sensíveis à simulação determinística do jogo;
- Garantia que todas variáveis do estado de jogo ou declaradas para serem lidas durante a simulação sejam propriamente inicializadas;
- Garantia da ordem de avaliação de todas expressões da simulação;
- Que o estado de jogo seja carregado por completo, e que nenhuma parte seja acrescentada ou removida dinamicamente durante a sessão. Isso inclui entidades e valores carregados dos arquivos de jogo ou conexões de novos peers;

Além disso, o rollback requer que o estado de jogo seja serializável, ou seja, que possa existir como sequência de bytes, sem mudar sua natureza ao ser transformado de volta em um objeto, o que implica a dispensa de complexas estruturas de dados alocados na heap da memória de programa como representações do estado de jogo; e que a simulação possa ser executada múltiplas vezes no intervalo de um só frame sem queda de performance, o que requer o máximo desacoplamento possível das operações de simulação daquelas que não tem efeito no estado de jogo ao fim do frame. O que rollback não requer, entretanto, é que as entradas feitas por jogadores a cada frame sejam representadas exclusivamente de forma discreta, podendo comportar entradas analógicas em troca de menor taxa de acertos em técnicas de predição de entradas remotas através da extrapolação das anteriores.

6.2. Trabalhos Futuros

Sugere-se que trabalhos futuros incluam a flexibilização das restrições listadas acima, com foco nas que restringem possibilidades de design:

- Se o carregamento dinâmico de assets de jogo puder ser sincronizado, jogadores conectados por rollback podem coabitar níveis de jogo mais vastos, desde que a memória de cada peer comporte todas as zonas carregadas por cada um, ou então que por design os jogadores sejam forçados a não ficarem distantes um do outro;
- Se a conexão dinâmica de novos jogadores puder ser aceita pelos peers da sessão e então sincronizada sem longas pausas, serão possibilitadas sessões mais extensas;
- Ao que forem desenvolvidos métodos para minimizar o número de vértices no grafo de conexões entre peers sem afetar performance, o número de jogadores que podem estar conectados em uma mesma sessão poderá aumentar cada vez mais.

7. Bibliografia

7.1. Artigos da internet

[1] ANDRESS, Justin. Why 'Super Smash Bros. Melee' Pros Play on Ancient TVs. **Inverse**, New York City, 19 de Julho de 2016. Disponível em: https://www.inverse.com/article/18494-professional-gaming-super-smash-bros-melee-crt-lcd. Acesso em: 18 de Dezembro de 2022.

[2] C Sequence Points. Microsoft Learn. 3 de Agosto de 2021. Disponível em: https://learn.microsoft.com/en-us/cpp/c-language/c-sequence-points. Acesso em: 31 de Março de 2023.

[3] CHANCE, Greg et al. **On determinism of game engines used for simulation-based autonomous vehicle verification**. IEEE Transactions on Intelligent Transportation Systems, 10 de Junho de 2022. Disponível em: https://ieeexplore.ieee.org/abstract/document/9793395. Acesso em: 8 de Março de 2023.

[4] COHU, Cameron. **Rollback Networking in Peer-To-Peer Video Games**. 2021. Disponível em: https://cameroncohu.com/wp-content/uploads/2021/05/Rollback-Networking-in-Peer-to-Peer-Video-Games.pdf. Acesso em: 8 de Março de 2023.

[5] DEWET, Matt; STRAILY, David. **Peeking into VALORANT's Netcode**. Riot Games Technology. Disponível em: https://technology.riotgames.com/news/peeking-valorants-netcode. Acesso em 10 de Março de 2023.

[6] EDWARDS, Tom; KESTRELGUY. **Lag Compensation**. Valve Developer Community. Disponível em: https://developer.valvesoftware.com/wiki/Lag_Compensation. Acesso em: 20 de Dezembro de 2022.

[7] FIEDLER, Glenn. **Deterministic Lockstep**. Gaffer on Games. 29 de Novembro de 2014. Disponível em: https://gafferongames.com/post/deterministic_lockstep/. Acesso em: 3 de Março de 2023.

[8] _____. **Floating Point Determinism**. Gaffer on Games. 24 de Fevereiro de 2010. Disponível em: https://gafferongames.com/post/floating_point_determinism/. Acesso em: 3 de Março de 2023.

[9] _____. **What Every Programmer Needs To Know About Game Networking**. Gaffer on Games. 24 de Fevereiro de 2010. Disponível em: https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/. Acesso em: 10 de Março de 2023.

[10] FOR Honor Preliminary Network Analysis. [*S. l.: s. n.*] Publicado pelo canal Battle(non)sense. 15 de Fevereiro de 2017. 1 vídeo (15 min). Disponível em: https://www.youtube.com/watch?v=tAU5bIalbnc. Acesso em: 8 de Março de 2023.

[11] GAO, Yuan. **Netcode Concepts Part 3: Lockstep and Rollback**. Medium, 22 de Abril de 2018. Disponível em: https://meseta.medium.com/netcode-concepts-part-3-lockstep-and-rollback-f70e9297271. Acesso em: 19 de Dezembro de 2022.

[12] GERYK, Bruce. GameSpot Presents: A History of Real-Time Strategy Games – The First Wave. **GameSpot**, 4 de Maio de 2001. Disponível em: https://web.archive.org/web/20091120052650/http://www.gamespot.com/gamespot/features/all/real_time/p3_01.html. Acesso em: 22 de Fevereiro de 2023.

[13] GILLARD, Mark. **TOML++**. Disponível em: https://marzer.github.io/tomlplusplus/. Acesso em: 31 de Março de 2023.

[14] GGPO | Rollback Networking SDK for Peer-to-Peer Games. Disponível em: https://www.ggpo.net. Acesso em: 31 de Março de 2023.
[15] HUYNH, Martin; VALARINO, Fernando. **An analysis of continuous consistency models in real time peer-to-peer fighting games**. 2019. Disponível em: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1322881&dswid=-6101. Acesso em: 8 de Março de 2023.
[16] JAKSA. **Active and Passive Replication in Distributed Systems**. Jaksa's Blog, 1 de Maio de 2009. Disponível em: https://jaksa.wordpress.com/2009/05/01/active-and-passive-replication-in-distributed-systems/. Acesso em: 22 de Fevereiro de 2023.
[17] LAG Compensation will ruin CoD. Reddit, 26 de Março de 2017. Disponível em: https://www.reddit.com/r/Infinitewarfare/comments/61nuua/lag_compensation_will_ruin_cod/. Acesso em: 20 de Dezembro de 2022.
[18] LANKAMP, Mike. **fpm**. Disponível em: https://github.com/MikeLankamp/fpm. Acesso em: 4 de Março de 2023.
[19] LOVATO, Nathan et al. **Creating your first script**. Godot Engine (stable) Documentation in English. Disponível em: https://docs.godotengine.org/en/stable/getting_started/step_by_step/scripting_first_script.html. Acesso em: 3 de Março de 2023.
[20] LYSENKO, Mikola. **Replication in networked games: Overview (Part 1)**. 0 FPS, 10 de Fevereiro de 2014. Disponível em: https://0fps.net/2014/02/10/replication-in-networked-games-overview-part-1/. Acesso em: 19 de Dezembro de 2022.
[21] MORTORAY, Edaqa. **The uninitialized variable anathema: non-deterministic C++**. DEV Community, 17 de Dezembro de 2017. Disponível em: https://dev.to/mortoray/the-uninitialized-variable-anathema-non-deterministic-c-71a. Acesso em: 30 de Março de 2023.
[22] OPEN World Games and Asset Streaming. Meta Developers. Disponível em: https://developer.oculus.com/documentation/unity/po-assetstreaming/. Acesso em: 4 de Março de 2023.
[23] PROVINCIANO, Brian. **Automated Testing and Instant Replays in Retro City Rampage**. GDC Vault, 2015. Disponível em: https://www.gdcvault.com/play/1021825/Automated-Testing-and-Instant-Replays. Aceso em 30 de Março de 2023.
[24] QUAKE Pro League. Liquipedia Arena FPS Wiki. Disponível em: https://liquipedia.net/arenafps/Quake_Pro_League. Acesso em: 31 de Março de 2023.
[25] SANTAMARIA, Ramon. **Raylib | A simple and easy-to-use library to enjoy videogames programming**. Disponível em: https://www.raylib.com. Acesso em: 31 de Março de 2023.
[26] STALLONE, Michael. **8 Frames in 16ms: Rollback Networking in 'Mortal Kombat' and 'Injustice 2'**. GDC Vault, 2018. Disponível em: https://www.gdcvault.com/play/1025471/8-Frames-in-16ms-Rollback. Acesso em: 18 de Dezembro de 2022.
[27] TRICKLESS_MAGICIAN. **Limsa Lominsa 10 minutes before the maintenance**. Reddit, 2 de Dezembro de 2021. Disponível em: https://www.reddit.com/r/ffxiv/comments/r7360g/limsa_lominsa_10_minutes_before_the_maintenance/. Acesso em: 22 de Fevereiro de 2023.
[28] VALGRIND. Disponível em: https://valgrind.org. Acesso em 30 de Março de 2023.
[29] WEINKOVE, Andrew. **Minimizing the Pain of Lockstep Multiplayer**. Postagem de Daniel Collier. Game Developer, 24 de Novembro de 2015. Disponível em:

https://www.gamedeveloper.com/programming/minimizing-the-pain-of-lockstep-multiplayer. Acesso em: 10 de Março de 2023.
[30] WELLBELOVE, John. **Embedded Template Library**. Disponível em: https://www.etlcpp.com. Acesso em: 31 de Março de 2023.
[31] WHAT is a Rollback (in Databases)?. Techopedia. 31 de Agosto de 2012. Disponível em: https://www.techopedia.com/definition/9229/rollback. Acesso em: 22 de Fevereiro de 2023.

7.2. Livros

[32] GLAZER, Joshua; MADHAV, Sanjay. **Multiplayer game programming: Architecting networked games**. Addison-Wesley Professional, 2015.
[33] NYSTROM, Robert. **Game Programming Patterns**: Game Loop. Genever Benning, 2 de Novembro de 2014. Disponível em: https://gameprogrammingpatterns.com. Acesso em: 8 de Março de 2023.