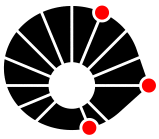

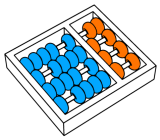

Compensação de atraso em um jogo atirador em terceira pessoa online

Júlio Moreira Blás de Barros *Luiz F. Bittencourt*

Relatório Técnico - IC-PFG-22-06
Projeto Final de Graduação
2022 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Compensação de atraso em um jogo atirador em terceira pessoa online

Júlio Moreira Blás de Barros Luiz Fernando Bittencourt*

Resumo

Com a crescente popularização de jogos competitivos, formas de melhorar a experiência de usuário são cruciais para atrair jogadores e criar produtos. Uma das principais formas de melhora é a compensação de atraso. Nesse trabalho, é descrita a implementação de um algoritmo de compensação de atraso em um jogo atirador em terceira pessoa, utilizando a plataforma *Unity*. Posteriormente, é analisado o impacto desse algoritmo na taxa de acertos de tiros, tanto em ambientes locais quanto remotos (*Cloud*), e é demonstrada a influência desses algortimos na experiência final.

1 Introdução

Nos últimos 20 anos, a internet sofreu enormes melhorias na qualidade de seus serviços. Um dos fatores determinantes dessas melhorias foi o avanço tecnológico de sua infraestrutura, que permitiu que as conexões fossem mais estáveis e rápidas [1].

Essas conexões de melhor qualidade permitiram que jogos multijogador pudessem ser jogados através da internet, atingindo um público muito maior e levando produtores a desenvolver e aprimorar esses jogos.

À medida que a competitividade desses jogos aumentou, percebeu-se que em uma rede instável como a internet, o tempo entre o envio e o recebimento de pacotes de dados poderia ser grande a ponto de prejudicar a qualidade da experiência dos jogadores. Por conta disso, mecanismos de compensação de atraso, também descrita como *lag compensation*, são desenvolvidos a fim de evitar que esse tempo, que é inevitável, afete a jogabilidade e responsividade desses sistemas.

Um exemplo indispensável desses jogos competitivos são os jogos atiradores em primeira pessoa (*First Person Shooter* ou *FPS*) e terceira pessoa (*Third Person Shooter* ou *TPS*). Nesses jogos, jogadores simulam batalhas armados com fuzis que podem disparar muitas balas por segundo, com cada uma delas impactando no diretamente estado do jogo. Além disso, para evitar que jogadores maliciosos manipulem os pacotes a fim de obter vantagens indevidas, é basal que os distribuidores desses jogos hospedem réplicas das simulações em servidores dedicados controlados pelo distribuidor, de forma que validações de estado e de entrada de jogadores possam ser efetuadas.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

Nesse contexto, o atraso introduzido para a atualização de estado entre jogadores pode ser muito elevado, por existirem informações que devem ser enviadas, recebidas, processadas e reenviadas entre um cliente, o servidor dedicado e outro(s) cliente(s). Com muitas atualizações de estado ocorrendo por segundo (até 128 vezes por segundo), mecanismos de compensação de atraso se tornam imprescindíveis [2].

Para a implementação desses jogos, é comum a utilização de motores pre-construídos, que já implementam física, renderização, compressão e tratamento de entrada dos jogadores. Além disso, esses motores disponibilizam frameworks que facilitam o desenvolvimento de jogos. O mais utilizado desses motores é a *Unity* engine [3].

Nesse trabalho, é descrita a implementação de um jogo atirador em terceira pessoa multijogador online, com a utilização do motor *Unity* e aplicação de um algoritmo de compensação de atraso. Descrevo, por fim, um experimento que busca medir o impacto que o algoritmo de compensação de atraso tem na qualidade de experiência, através da medição da taxa de acertos registrados (acertos de projéteis nos quais o servidor e o sistema cliente do jogador concordam) em comparação a acertos não registrados (nos quais o servidor não concorda com o cliente) [4].

2 Referencial teórico

Antes de descrever o funcionamento do jogo implementado, é importante que alguns conceitos comuns e padrões utilizados nesse desenvolvimento sejam devidamente apresentados e explicados.

2.1 Mecanismos de comunicação

Ao implementar um jogo multijogador, há duas formas fundamentais de estruturar os dados comunicados, a fim de atingir o equilíbrio requerido entre performance (banda utilizada, velocidade de comunicação) e consistência entre simulações.

2.1.1 Comunicação de *inputs* - *Lockstep*

A primeira e mais intuitiva maneira das instâncias de jogos multijogador se comunicarem é através da comunicação de *inputs* de jogadores, também chamada de *Lockstep*. Nessa situação, a instância do jogador capta os *inputs* do mesmo (em geral, botões apertados e posição de *joysticks*) e os envia para um servidor dedicado. Esse servidor dedicado valida e executa esses *inputs*, comunicando aos demais clientes eventuais mudanças no estado compartilhado. Há também a situação em que o servidor dedicado repassa os *inputs* com uma marcação de tempo, de forma que os demais clientes possam replicá-los em simulações locais [5].

Nesse contexto, simulações locais aumentam a responsividade do sistema, pois podem executar imediatamente *inputs* recebidos do usuário, mas a consistência é prejudicada, porque o momento em que os *inputs* são recebidos difere entre instâncias. Por outro lado, se a simulação só ocorre no servidor dedicado, a consistência do estado dos jogadores é

muito maior, mas um jogador pode ter que esperar por muito tempo até que veja seu *input* influenciando no jogo.

Em jogos competitivos, é comum que simulações ocorram tanto no cliente quanto no servidor, sendo o servidor somente uma ferramenta de validações. No exemplo do gênero *Shooter*, é comum que o servidor dedicado valide acertos de tiros, de forma a evitar que usuários maliciosos comuniquem acertos que não seguem as regras da simulação, e somente repasse tiros que não acertam [6].

2.1.2 Interpolação de *snapshots*

Outra forma de comunicar informações é através da propagação periódica do estado completo das entidades da simulação. Para isso, são extraídos *snapshots*, informações de uma ou mais entidades em um determinado momento. Essas informações são trafegadas e a simulação que as recebe deve estimar mudanças de estado a partir desses *snapshots* [5].

É importante notar que entre o envio de dois *snapshots*, há um intervalo de tempo que deve ser preenchido. Esse preenchimento pode ser feito através de um cálculo de *snapshots* intermediários, a partir das informações já recebidas. Esse cálculo é chamado de interpolação.

Um exemplo clássico, também utilizado no gênero *Shooter*, é da interpolação de movimento. Nesse caso, o servidor dedicado envia aos jogadores posições instantâneas dos demais, e através de um atraso na renderização ou mesmo de previsões, esses jogadores são renderizados em posições calculadas para momentos em que não existem *snapshots* [7]. Uma forma comum de calcular essas posições é através de uma regressão (linear, quadrática, etc) dos pontos 3d recebidos para uma função aproximada que relacione posições ao tempo. Com essa função, basta calcular um novo valor para o tempo atual e renderizar a entidade na posição obtida.

2.1.3 Modelos híbridos

Como exemplificado, os dois modelos de comunicação podem ser utilizados em um mesmo jogo. Um exemplo disso pode ser a própria movimentação de jogadores. É possível construir um sistema em que jogadores enviam seus *inputs* de movimento, mas recebem as posições de outros jogadores já consolidadas. Há também situações em que a comunicação é feita através de *Lockstep*, mas *snapshots* são eventualmente enviados para corrigir estados divergentes (estados podem divergir por conta de diferença entre relógios e tempo de processamento).

2.2 Métodos de compensação de atraso

Para compensar o atraso entre a comunicação de *inputs* ou *snapshots*, várias técnicas são propostas e implementadas, sendo essas técnicas intercambiáveis e não exclusivas. Os jogos multijogador, em geral, utilizam de múltiplas dessas técnicas para atingir um nível aceitável de responsividade e consistência de acordo com o gênero e objetivo do jogo.

2.2.1 Atraso remoto (*Remote lag*)

Uma forma de compensar atrasos de comunicação é assumir que o atraso vai ocorrer e introduzi-lo como parte da simulação. Nesse caso, entidades controladas remotamente são construídas em um estado passado, a uma distância fixa do momento atual. Em geral, essa distância é maior do que o atraso médio, fazendo com que a simulação na maioria das vezes conheça um estado mais recente do que o construído atualmente. Dessa forma, as mudanças de estado podem acontecer de forma mais fluida, melhorando a responsividade. Por outro lado, essa abordagem faz com que a simulação seja sempre inconsistente, porque o estado atual de um jogador somente será observado no momento correto por aquele jogador.

2.2.2 Atraso local (*Local lag*)

Em um sistema que utiliza *Lockstep*, é possível compensar o atraso introduzindo o atraso na aplicação do *input* do jogador local. Para que isso não seja absolutamente irresponsivo, é comum que animações longas ou barras de carregamento de *input* sejam apresentadas, de forma que o usuário veja seu *input* sendo preparado. Para que o atraso seja compensado, esse *input* é enviado **antes** do início dessa animação ou carregamento, de forma que o servidor receba e aplique esse input com um atraso menor do que se ele fosse aplicado imediatamente na simulação local [6].

2.2.3 Extrapolação (*Dead reckoning*)

Em busca de melhorar a responsividade do sistema quando não há atraso sintético ou quando o atraso de rede é maior do que o sintético, é possível prever ações ou o estado de uma entidade. Por exemplo, se um jogador se move em uma direção, a chance de ele mudar de direção é muito menor do que a de ele continuar.

Portanto, alguns jogos que utilizam *Lockstep* prevêem que os jogadores irão repetir ações antes mesmo de receber essas ações. Da mesma forma, jogos que utilizam *snapshots* podem inferir a função do estado com snapshots passados e aplicá-la em um tempo para o qual não há *snapshot* futuro [6, 10].

Apesar de promissora, essa abordagem requer que além de sua implementação principal, seja implementada uma lógica de correção, para o caso em que as previsões estiverem erradas. Nessa sistuação, essa lógica deve transformar a entidade a partir de um estado previsto para um estado conhecido, de forma a evitar que a experiência seja prejudicada [7, 11].

2.3 Compensação de atraso no gênero *Shooter*

No gênero de jogos *Shooter*, há algumas compensações de atraso específicas que podem ser implementadas. A mais comum é a compensação no registro de acertos, também conhecido como *hit registration*. Dado que atrasos são inevitáveis, um jogador local sempre estará em um estado futuro em relação a outros jogadores. Quando ele atira em um jogador remoto, esse jogador remoto provavelmente estará em outro lugar no momento do tiro, mas

o atirador verá o tiro acertando. Se esse atraso não for compensado e a validação do tiro for feita no servidor, a responsividade dos tiros é muito prejudicada.

2.3.1 Visualização do atirador

A forma clássica de compensar o atraso ao registrar acertos é calcular o atraso total do atirador e simular uma volta no tempo a cada tiro. Sendo assim, o servidor vai validar que o atirador realmente estava observando a vítima no momento do tiro na sua simulação local, mesmo que a simulação do servidor esteja diferente no momento do tiro. Isso é chamado de *Collider Rollback* [14].

Essa abordagem tem dois problemas principais. O primeiro é que aceitando que o atraso existe (ou até mesmo o induzindo), jogadores que saem de coberturas têm uma vantagem: verão outros jogadores antes que possam ser vistos. Isso acontece porque o atraso na movimentação desses jogadores vai fazer com que eles se movam em momentos diferentes em cada simulação, sendo a simulação local a mais atualizada. Esse fenômeno é chamado de *Peeker's Advantage* [2].

O segundo problema é no caso em que jogadores entram em cobertura. Se um jogador atira em outro logo antes que a vítima entre em uma cobertura, pode ser que a vítima perceba que foi atingida mesmo depois de entrar na cobertura, como se o tiro atravessasse a parede. Esse fenômeno é descrito como *Shot Behind Covers*, cuja sigla é SBC [4].

2.3.2 Visualização da vítima

A fim de resolver o problema de acertos após a entrada em cobertura, é possível introduzir uma validação extra nos tiros. Para a situação em que o tiro é recebido pelo servidor momentos após a vítima entrar em cobertura ou se tornar invulnerável, mas antes que essa condição fosse atualizada na simulação do atirador, é possível que o servidor reavalie o tiro como se o alvo fosse a vítima no momento atual do servidor. Esse segundo tiro pode ser inválido, recusando o registro de acerto e melhorando a experiência da vítima [4, 10].

2.3.3 Híbridos

Em geral, jogos preferem a utilização de compensação de atraso com base na visualização do atirador, sendo a validação de visualização da vítima uma melhoria aplicada somente em situações em que a vítima seria prejudicada. Nesses casos, a simulação de jogo da vítima pode enviar uma contestação de acerto a fim de melhorar a jogabilidade [4].

3 Objetivos

Os trabalhos acadêmicos que tratam de compensação de atraso em sua maioria trazem foco em atiradores em primeira pessoa, por ter sido o gênero de atiradores de maior sucesso na última década. Entretanto, nos últimos anos, houve um crescimento do gênero de atiradores em terceira pessoa, por conta da popularização de novos modos de jogo e a possibilidade de

customizar personagens advinda das melhorias da capacidade de computação dos dispositivos. Jogos como *Fortnite*, *Free Fire* e *PlayerUnknown's Battlegrounds* são exemplos de jogos atiradores em terceira pessoa que compõem o grupo de jogos mais jogados mundialmente.

Apesar desse crescimento, a quantidade de pesquisas envolvendo esse gênero de jogos ainda não é relevante. Este trabalho visa aumentar esse repertório. Neste trabalho, é descrita a implementação de um modelo de jogo atirador em terceira pessoa com um mecanismo de compensação de atraso aplicando atraso remoto e compensação de atraso no registro de acertos através de um *collider rollback* alterado. Além disso, nesse jogo há a opção de desligar esses algoritmos e há a captação de métricas relevantes para eventuais experimentos, como taxa de acerto e taxa de atraso.

Com isso, busco avaliar os impactos dessas implementações na jogabilidade, especialmente na taxa de acerto, para diferentes configurações de rede, variando o atraso na rede e a posição geográfica de seus nós.

4 Metodologia

Para desenvolver o jogo mencionado, foi utilizada a plataforma *Unity*. Essa ferramenta já fornece tratamentos físicos e de renderização. A fim de replicar o comportamento de *TPS*s típicos, tanto o código do servidor quanto o código do cliente foram implementados no mesmo projeto, de forma a reutilizar o motor de física e as regras do jogo. A fim de agilizar o desenvolvimento, foram utilizados recursos (áudio, texturas, modelos 3d, fontes) de domínio público. Por fim, o código de rede foi desenvolvido utilizando a biblioteca de código aberto *LiteNetLib*, que implementa comunicação confiável e não confiável através do protocolo *UDP* [13].

4.1 Movimentação

Um dos mecanismos principais do protótipo é a movimentação. A movimentação do personagem local segue os *inputs* do jogador, que são executados localmente. Para replicar a movimentação dos jogadores, foi utilizada a comunicação por *snapshots* com interpolação linear entre *snapshots* [5].

Nesse contexto, uma subrotina monitora a posição do jogador local e envia ao servidor sua posição instantânea, juntamente com um número de sequência em intervalos fixos definidos pelo desenvolvedor. Para os experimentos, foi definido um intervalo de envio padrão de 30 envios por segundo.

Ao receber a posição de um jogador remoto, uma simulação a enfileira em uma fila de interpolação. Após esperar um tempo equivalente ao atraso remoto definido pelo desenvolvedor, o jogador remoto é movido até a próxima posição dessa fila seguindo uma interpolação linear entre a penúltima e última posições desenfileiradas. Ao chegar na posição destino, uma nova posição é desenfileirada. Note que a diferença entre números de sequência é proporcional à diferença de tempo entre posições, dado que as atualizações são enviadas em intervalos fixos. Essa implementação segue a lógica de **Atraso remoto** para compensação de atraso, porque a renderização dos jogadores remotos está mais atrasada do que a do jogador local [11].

4.2 Tiros

Ao atirar, o cliente calcula se na simulação local o tiro acertou através de um mecanismo físico de *raycast*, que funciona como um laser. O primeiro colisor (obstáculo ou jogador são colisores) que o laser atinge é calculado como o alvo do tiro. Se o tiro atinge um jogador, é enviado um pacote com vetores de direção e posição do tiro, o id numérico do jogador atingido e a sequência do *snapshot* da vítima vista pelo cliente do atirador ao atirar.

Com essas informações, o servidor pode calcular a diferença de posição da vítima em sua simulação com a simulação do atirador utilizando o número de sequência como uma marcação de tempo. Para que isso seja possível, o servidor armazena um *buffer* de snapshots recebidos anteriormente. Com essas duas informações, o servidor deve realizar o seguinte cálculo:

$$\Delta p = B(s_s) - B(s_a)$$

Onde Δp é a diferença de posição da vítima entre simulações, $B(s_x)$ é o snapshot na sequência x, s_s a sequência executada na simulação do servidor e s_a a sequência executada na simulação do atirador.

Com isso, o servidor desloca o vetor de posição de origem do tiro na mesma quantidade em que as posições da vítima diferem (Δp). O deslocamento se dá na origem do tiro porque o *raycast* é somente um teste que não altera o estado físico do sistema, enquanto mover a vítima do tiro poderia alterar esse estado, prejudicando a performance.

Após esse deslocamento, o servidor testa o *raycast*, e caso esse retorne um resultado positivo, o tiro é então validado. Como a origem do tiro é deslocada, espera-se que não ocorram tiros que atravessam obstáculos, porque a posição de origem seria deslocada para atrás desses obstáculos, nesse caso. Por fim, no caso em que o servidor valida o tiro, um pacote é enviado para todos os clientes (inclusive o atirador) que então podem calcular resultados do tiro (dano, animações).

Esse algoritmo tem o mesmo objetivo de um *Collider Rollback* [14], com a diferença de que a volta ao passado acontece na origem do tiro, ao invés de ocorrer no colisor da vítima.

4.3 Implementação

A implementação completa utilizada nesse trabalho pode ser encontrada em `https://github.com/juliombb/lag-warfare`

5 Experimentos

Para realizar experimentos, foi adicionada uma lógica que permite que um jogador se mova de um lado para o outro automaticamente. Além disso, foi adicionada uma métrica de taxa de acertos e *round trip time*. Nesse contexto, duas sessões de jogo e um servidor são iniciadas, sendo uma sessão a vítima, com o movimento automático aplicado e outra sessão o atirador. Durante 2 minutos, o atirador segue a vítima com a mira e atira sem interrupções até que o tempo se esgote.

Nos testes, os dois jogadores se posicionam a uma distância de 8 metros, sendo o metro a unidade de medida padrão da *Unity*. O jogador que simula a vítima se move com velocidade

senoidal, com módulo máximo variado entre testes. Os personagens do jogo tem formato aproximado por uma cápsula de 2m de altura e 1m de diâmetro. Com essas predefinições, quatro variações base foram aplicadas:

1. Sessões e servidor locais, com atraso compensado nos acertos
2. Sessões e servidor locais, sem atraso compensado nos acertos
3. Sessões locais e servidor em *cloud*, com atraso compensado nos acertos
4. Sessões locais e servidor em *cloud*, sem atraso compensado nos acertos

Analisando a taxa de acertos em cada variação, espera-se definir se em situações que refletem jogos em *LAN* e jogos *online* hospedados em *cloud* a compensação de atraso é efetiva e pode melhorar a experiência de jogo.

Para as sessões locais, foi utilizado um *laptop* com processador Intel core i7-3630QM (2.4 Ghz, 8 CPUs), memória RAM de 8GB, placa de vídeo *Nvidia Geforce GTX 660M*, com *Windows 10*, localizado em Campinas, no estado de São Paulo, no Brasil.

Para executar o servidor em *cloud*, foi utilizado o serviço de computação EC2 da AWS. Nesse serviço, foram provisionadas instâncias do tipo *t2.micro*, com 1 vCPU e 1GB de memória RAM, executando *Amazon Linux*. Foram executadas variações em múltiplas localizações.

5.1 Resultados

A Figura 1 apresenta os resultados obtidos, discutidos a seguir.

Simulações locais: Através dos resultados obtidos, é possível concluir que em um jogo atirador em terceira pessoa local, o uso de compensação de atraso pode melhorar muito a experiência dos jogadores, em situações nas quais as entidades remotas se movem muito rápido. Nessas situações, é possível observar melhorias de até 400% na taxa de acertos (velocidade remota de 0.5m/s).

Por outro lado, se as entidades remotas se movem em velocidades baixas, como o atraso é quase nulo, não há ganhos em utilizar algoritmos de compensação de atraso. (velocidade remota de 0.1m/s).

Uma possível explicação para a melhoria observada nos servidores locais é o atraso existente na propagação das informações na rede local e principalmente no processamento dessas informações, que pode onerar o processamento do computador, causando atrasos perceptíveis, dado que esse está executando 3 simulações e renderizando duas.

Simulações remotas: Ao analisar a execução remota desse tipo de jogo, torna-se imperativo o uso de algoritmos de compensação de atraso para que a experiência seja verossímil, pois uma precisão inferior a 5%, observada em todos os testes remotos sem tal algoritmo, impossibilita qualquer tipo de competitividade.

É possível observar, também, que a taxa de acerto aumenta de forma inversamente proporcional à velocidade do jogador remoto. No caso desse experimento, isso pode ser explicado pela interpolação de *snapshots*. O algoritmo de compensação de atraso não leva em

Taxa de acertos por localização

Geolocalização \ Velocidade	0.1 m/s	0.25 m/s	0.5 m/s
Local (Campinas) RTT 0ms	**0,703703700**	**0,845559800**	**0,690909100**
	0,749490900	**0,288321200**	**0,173708900**
AWS São Paulo RTT [4,400]ms	**0,929824600**	**0,844720500**	**0,730769200**
	0,041358407	**0,042296070**	**0,009259259**
AWS Ohio RTT [100,400]ms	**0,917226000**	**0,877193000**	**0,765822800**
	0,034334760	**0,006600660**	**0,005405406**
AWS Cingapura RTT [250,600]ms	**0,914942500**	**0,893835600**	**0,724550900**
	0,041214750	**0,007326007**	**0,005555556**

Atraso compensado
Sem compensação

Figura 1: Cada coluna representa uma velocidade máxima (variando entre 0.1 metro por segundo, 0.25 metro por segundo e 0.5 metro por segundo). As linhas representam a localização do servidor de jogo. Cada célula contém a taxa de acerto (fração de acertos validados sobre acertos contabilizados) com compensação, em verde e sem compensação, em vermelho.

conta interpolações em andamento, somente reverte o estado para um *snapshot* conhecido. Se a velocidade das entidades remotas é muito alta, pode ser que uma posição intermediária da interpolação seja atingida por um tiro, mas o *rollback* seja feito para o início dessa interpolação, que pode estar em uma posição muito diferente.

Por fim, nota-se que a taxa de acerto sem compensação de atraso cai à medida que o atraso aumenta, enquanto a taxa de acerto com compensação de atraso se mantém constante, independente do atraso induzido. Esse é o objetivo principal que se busca alcançar com esses algoritmos, uma vez que o atraso passa a ser despercebido pelos usuários.

5.2 Anomalias e riscos

Alguns dos resultados não seguem a lógica geral concluída anteriormente em todas as medições, como a taxa de acerto sem compensação na AWS São Paulo com velocidade 0.1m/s e os testes locais.

A primeira dessas anomalias ocorreu no teste da AWS São Paulo com velocidade 0.1m/s. Para atingir o valor encontrado, foi feita uma média entre 3 medições, sendo uma delas com taxa mais baixa de acertos do que o esperado. Essa anomalia pode ter sido causada pela instabilidade da rede dessa localização, que apresentou grandes variações de atraso, ao contrário das demais. Sendo essa taxa muito baixa sem compensação, qualquer tiro validado a mais ou a menos pode influenciar muito no valor final da taxa de acertos.

A anomalia nos testes locais, que apresentaram menor taxa de acerto com compensação de atraso, pode ter sido causada por sobrecarga de processador. Como o mesmo computador executa as 3 simulações, pode existir maior lentidão no processamento de mensagens e renderização. Essa lentidão pode causar atrasos em momentos não contabilizáveis, como entre um tiro acertar e a sequência (tempo) ser capturada. Esse tipo de atraso causaria uma invalidação do tiro, porque o cliente enviaria uma mensagem afirmando que viu uma vítima em uma posição inválida para o próprio tiro. Essa anomalia não foi removida mesmo repetindo o experimento.

6 Conclusão

Através das observações, é reforçada a necessidade do uso de algoritmos de compensação de atraso em jogos *Shooter online*. Apesar de não atingir a perfeição no registro de acertos, esses algoritmos melhoram significativamente a experiência na maioria das situações.

Analisando os experimentos e resultados, foi possível encontrar fatores que influenciam na eficiência e ineficiência de algortimos de compensação de atraso. O principal fator que causa imprecisões nesses algoritmos é a medição do atraso. Como a percepção de tempo e o tempo de processamento de dois nós de computação difere, um mesmo intervalo observado por um humano será calculado de forma diferente entre os dois nós. Como corrigir essa diferença de cálculo de tempo é impraticável no tempo exigido por jogos em tempo real, a compensação de atraso nunca será perfeita, pois o atraso calculado pelo servidor será sempre uma aproximação do atraso real.

É surpreendente observar que o atraso é influenciado não só pela rede, mas também pela taxa de processamento dos nós, o que pode se tornar um risco quando se utiliza esse tipo de

algoritmo. Sendo assim, ao escolher o *hardware* que executará as simulações, equipamentos poderosos são importantes para garantir o bom funcionamento desses algoritmos.

Outro fator de grande influência na eficiência desses algoritmos é a velocidade das entidades. Quando uma entidade remota se move muito rapidamente, mesmo diferenças pequenas no tempo podem causar grandes diferenças em sua posição. Por conta disso, é mais difícil reconstruir um estado passado, dado que uma divergência entre atraso compensado e atraso real causa uma divergência maior entre a posição aproximada e a posição real da entidade. Definitivamente, levar esse fator em consideração ao idealizar um jogo *online* pode levar a experiências melhores.

Por fim, essa pesquisa demonstrou que a necessidade do uso de algoritmos de compensação de atraso introduz muitos fatores, requisitos e limitações a serem considerados na implementação de um jogo *TPS online*.

Como trabalhos futuros, esta pesquisa pode ser expandida de diferentes formas:

- Teste em rede LAN local, com cada computador executando somente uma simulação;
- Teste com múltiplos jogadores
- Teste com múltiplos algoritmos de compensação de atraso

Referências

[1] Artel Communications Corporation, *Fiber Optics in RGB Color Computer Graphics Communications*, Application Note CG-1, disponível em `https://web.mit.edu/course/21/21.guide/pd-cxc.htm`

[2] VALORANT, *Código de rede e servidores de 128 ticks*, Diários dos Devs, disponível em `https://www.youtube.com/watch?v=_Cu97mr7zcM`

[3] Unity, disponível em `https://unity.com/`

[4] Steven W. K. Lee and Rocky K. C. Chang. 2018. *Enhancing the experience of multiplayer shooter games via advanced lag compensation.* In Proceedings of the 9th ACM Multimedia Systems Conference (MMSys '18). Association for Computing Machinery, New York, NY, USA, 284–293.

[5] Glenn Fiedler, *Networked Physics*, disponível em `https://gafferongames.com/categories/networked-physics/`

[6] Tony Cannon. *Fight the Lag! The Trick Behind GGPO's Low Latency Netcode* Game Developer Magazine (2012)

[7] Savery, C., Graham, T.C.N. *Timelines: simplifying the programming of lag compensation for the next generation of networked games.* Multimedia Systems 19, 271–287 (2013).

[8] Battle(non)Sense, *Netcode Delay: Battlefield vs CS:GO*, disponível em `https://www.youtube.com/watch?v=B40u12etY_U`

[9] Battle(non)Sense, *Battlefield 4 Netcode demystified*, disponível em `https://www.youtube.com/watch?v=KJoKn42j6W8`

[10] Timothy Ford, *Overwatch Gameplay Architecture and Netcode*, GDC 2017, disponível em `https://www.youtube.com/watch?v=zrIY0eIyqmI`

[11] Savery, Cheryl & Graham, T.C. & Gutwin, Carl & Brown, Michelle. (2014). *The effects of consistency maintenance methods on player experience and performance in networked games.* Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW. 1344-1355.

[12] Li, Z., Melvin, H., Bruzgiene, R., Pocta, P., Skorin-Kapov, L., Zgank, A. (2018). *Lag Compensation for First-Person Shooter Games in Cloud Gaming.* In: Ganchev, I., van der Mei, R., van den Berg, H. (eds) Autonomous Control for a Reliable Internet of Services. Lecture Notes in Computer Science(), vol 10768. Springer, Cham.

[13] Ruslan Pyrch, *LiteNetLib*, disponível em `https://github.com/RevenantX/LiteNetLib/`

[14] Zach Ross, *Why You Get Bad Hit Registration*, disponível em `https://www.youtube.com/watch?v=1fZFhzCsHYI`