UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RUAN DA FONSECA RAMOS

DESENVOLVIMENTO DE UM JOGO MULTIPLAYER DE TEMPO REAL E DISTRIBUÍDO COM DESAFIOS DE PERGUNTAS E RESPOSTAS

RIO DE JANEIRO
2024

RUAN DA FONSECA RAMOS

DESENVOLVIMENTO DE UM JOGO MULTIPLAYER DE TEMPO REAL E DISTRIBUÍDO COM DESAFIOS DE PERGUNTAS E RESPOSTAS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Silvana Rossetto

RIO DE JANEIRO
2024

CIP - Catalogação na Publicação

R175d

Ramos, Ruan da Fonseca
Desenvolvimento de um jogo multiplayer de tempo real e distribuído com desafios de perguntas e respostas / Ruan da Fonseca Ramos. -- Rio de Janeiro, 2024.
98 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. Sistemas distribuídos. 2. Jogo multijogador. 3. Cliente-servidor. 4. Jogo educacional. I. Rossetto, Silvana, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

RUAN DA FONSECA RAMOS

DESENVOLVIMENTO DE UM JOGO MULTIPLAYER DE TEMPO REAL E DISTRIBUÍDO COM DESAFIOS DE PERGUNTAS E RESPOSTAS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 19 de agosto de 2024

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
SILVANA ROSSETTO
Data: 22/08/2024 13:35:56-0300
Verifique em https://validar.iti.gov.br

Silvana Rossetto
Instituto de Computação - UFRJ

gov.br Documento assinado digitalmente
GERALDO BONORINO XEXEO
Data: 22/08/2024 18:05:22-0300
Verifique em https://validar.iti.gov.br

Geraldo Bonorino Xexéo
Instituto Alberto Luiz Coimbra de Pós Graduação e Pesquisa de Engenharia - UFRJ

gov.br Documento assinado digitalmente
MARIA HELENA CAUTIERO HORTA JARDIM
Data: 22/08/2024 14:14:37-0300
Verifique em https://validar.iti.gov.br

Maria Helena Cautiero Horta Jardim
Instituto de Computação - UFRJ

AGRADECIMENTOS

Gostaria de agradecer primeiramente ao meu irmão Igor da Fonseca Ramos, sem o qual eu não teria conseguido chegar até aqui e que sempre foi meu maior apoiador.

Agradeço também aos meus pais Edineudes da Fonseca Ramos e Arildo Ramos, e minha tia Elieudes Lopes da Fonseca, por sempre terem me dado todo o apoio necessário em todos os sentidos para que eu pudesse chegar preparado até o momento da realização deste trabalho e tendo superado todos os percalços que só nós sabemos quantos foram.

Finalmente, agradeço a todos os professores que fizeram parte desta caminhada acadêmica, em especial minha orientadora Silvana Rossetto, por toda a paciência e apoio para que este trabalho fosse concluído.

RESUMO

Neste trabalho, foi desenvolvido um jogo multijogador distribuído em tempo real de arquitetura cliente-servidor com um servidor o mais autoritário possível, que mede o tempo de resposta dos jogadores ao mesmo tempo que provê um *framework* de customização de perguntas e respostas com intuito pedagógico. Para o desenvolvimento do jogo foram utilizadas a *game engine Unity* e a biblioteca de comunicação *Netcode for GameObjects*. Foi realizado teste remoto com até quatro jogadores simultâneos e com a aplicação servidor executando em uma máquina virtual hospedada na nuvem pública Azure. Realizou-se uma detalhada descrição da arquitetura lógica e de sistema projetada, dos objetos utilizados e da implementação dos comportamentos e interações entre eles, ao mesmo tempo que se discutiu as decisões de *design* tomadas e os desafios enfrentados. Os resultados se mostraram satisfatórios, gerando uma primeira versão que atendeu os objetivos propostos inicialmente, apesar de conter diversos pontos que poderiam ser incrementados. O jogo ainda precisa ser experimentado por educadores para que sua relevância pedagógica seja devidamente avaliada.

Palavras-chave: sistemas distribuídos; jogo multijogador; cliente-servidor; jogo educacional.

ABSTRACT

This work presents the development of a real-time distributed multiplayer game with a client-server architecture, aiming to maximize server authority and explore time measurements of player response times, while also featuring a customizable question-and-answer framework for educational purposes. The Unity game engine was used for development with the Netcode for GameObjects high-level networking library. Remote testing was conducted with up to four simultaneous players and the server application running on a virtual machine hosted in the Azure public cloud. A detailed description of the designed system and logical architecture, the objects used, and the implementation of behaviors and interactions between them is provided. Design decisions and challenges faced are also discussed. The results were satisfactory, generating a first version that met the initially proposed objectives, despite containing several points that could be improved. The game still needs to be tested by educators to evaluate its pedagogical relevance properly.

Keywords: distributed systems; multiplayer game; client-server; educational game.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

ECS	Entity Component System
MOBA	Multiplayer Online Battle Arena
MMORPG	Massive Multiplayer Online Rolling Playing Game
TLD	Top Level Domain
MDA	Mechanics, Dynamics and Aesthetics
GDD	Game Design Document
RPC	Remote Procedure Call
P2P	Peer-to-peer
API	Application Programming Interface
2D	Duas dimensões
3D	Três dimensões
RTT	Round Trip Time
FPS	Frames Per Second

SUMÁRIO

1 INTRODUÇÃO

Não são mais os tempos em que jogos eletrônicos são considerados apenas entretenimento e são consumidos apenas por um pequeno nicho específico de pessoas. Vivemos, nas últimas décadas, um enorme crescimento desta indústria, que hoje já ultrapassa o faturamento de grandes e consolidadas formas de entretenimento, como o cinema (WIJMAN, 2021).

O avanço da tecnologia e do desenvolvimento de jogos, aliado à miniaturização dos componentes eletrônicos, possibilitou a popularização destes jogos, principalmente pela expansão para além dos computadores pessoais. Hoje é possível jogar em aparelhos móveis, tablets, consoles dedicados e até mesmo na nuvem.

Por conta da inclusão de outros públicos, há também um fator cultural e social importante, com pessoas de classes sociais mais baixas ganhando acesso aos jogos eletrônicos. Houve uma profunda transformação no perfil dos jogadores, abrindo-se espaço para um público mais heterogêneo, incluindo indivíduos de diferentes faixas etárias e gêneros. Além disso, com a democratização do acesso à Internet, os jogos online focados em entretenimento e competitividade tornaram-se espaços de socialização e formadores de grandes comunidades que giram inteiramente em torno deles (RODRIGUES; MUSTARO, 2007) (MENDES, 2006).

São muitas as possibilidades, além do entretenimento, para o uso de jogos eletrônicos. Alguns muito comuns e já bastante difundidos são simuladores profissionais, que podem ter elementos de *gamificação*, seja no setor aéreo, embarcados, indústrias com maquinários que exijam treinamento especial, simuladores automotivos, exercícios militares, e tantos outros mais (FLETCHER; TOBIAS, 2006) (JAYAKANTHAN, 2002).

Surgiu também uma vertente de jogos digitais que geram um novo leque de possibilidades, sendo utilizados em reabilitação de pessoas com limitações físicas e mentais (HAJESMAEEL-GOHARI et al., 2023) (LOPES et al., 2021) (KAZMI et al., 2014).

Em ambientes corporativos e em praticamente todos os tipos de aplicativos disponíveis nas lojas de *apps* mais populares encontra-se algum tipo de *gamificação* para tentar tornar tarefas diversas algo mais leve, divertido e atrativo, com alguma sensação de progressão, desafios e recompensas. Tal dinamismo provê um ambiente muito fértil para que as tecnologias impulsionem cada vez mais os usos dos jogos digitais como forma de inovação. Uma das áreas onde essa inovação tem um futuro cada vez mais promissor e que também impacta a sociedade em diversos aspectos importantes são os jogos com intuito educativo. A ideia principal é utilizar destes jogos para incentivar o desenvolvimento de estudantes mediante métodos mais lúdicos e menos rígidos e tradicionais, com uso da tecnologia que já é parte do dia-a-dia destes jovens, mas não limitado apenas a eles (HSIAO, 2007) (WHITTON, 2014) (PRENSKY, 2005).

1.1 MOTIVAÇÃO

O autor deste trabalho é um ávido consumidor de jogos online de vários jogadores simultâneos e também tem muito gosto por desenvolver jogos eletrônicos. Este trabalho é a materialização de uma ideia inicialmente simples, que consistia em explorar dois aspectos interessantes dos jogos online em tempo real, que são a sincronização de movimentação em tempo real e a tentativa de conseguir aplicar um mecanismo de decisão sobre a ordem das ações de diferentes jogadores.

A sincronização de movimentação em tempo real é um dos principais desafios técnicos dos jogos online multijogador. Ela garante que todos os jogadores vejam a mesma cena e que as ações de cada um sejam refletidas no mundo do jogo de forma consistente, mesmo com latências e perdas de pacotes na rede. Ela é essencial para que os jogadores se sintam imersos no mundo do jogo e que suas ações tenham um impacto real no desenrolar da partida. Isso é particularmente importante em jogos de ação rápida, onde os jogadores precisam reagir rapidamente às ações uns dos outros.

Em jogos online multijogador, determinar a ordem em que as ações dos jogadores são processadas é crucial para garantir uma jogabilidade justa e consistente. Este mecanismo de decisão da ordem das ações garante que todos os jogadores tenham a oportunidade de agir de forma justa e que suas ações não sejam invalidadas pelas ações de outros jogadores. Isso é crucial para manter o equilíbrio do jogo e evitar frustrações.

A estética dos jogos *IO*, explorada com sucesso em jogos como *Agar.io*[1], *Slither.io*[2] e *Surviv.io*[3], é frequentemente caracterizada por um estilo simples e minimalista, com foco na jogabilidade e na acessibilidade. Os elementos visuais geralmente são básicos e abstratos, utilizando cores sólidas e formas geométricas. Os cenários podem ser simples e repetitivos, ou podem apresentar uma variedade de elementos, mas com detalhes mínimos. A música e os efeitos sonoros também são geralmente minimalistas, servindo para complementar a jogabilidade sem distrair o jogador.

Por estes motivos, ela foi a escolhida, servindo como base para a implementação dos mecanismos de sincronização e decisão de ordem de ações. Este trabalho se utilizará de um *lobby* onde todos os jogadores se conectam e enfrentam todos os outros simultaneamente, podendo conectar-se e desconectar-se a qualquer momento.

1.2 OBJETIVOS

Tendo em vista o sucesso não tão recente dos jogos competitivos online, assim como dos jogos educativos, este trabalho tem por objetivo o desenvolvimento de um jogo online multijogador em tempo real que explore as duas principais motivações citadas na seção

[1] Disponível em https://agar.io/
[2] Disponível em https://slithergame.io/
[3] Disponível na plataforma Steam em https://store.steampowered.com/app/1375740

anterior. O resultado esperado é um jogo competitivo e divertido e, ao mesmo tempo, uma ferramenta de apoio para educadores.

Para alcançar o objetivo colocado, um desafio central reside em superar as limitações das soluções existentes, principalmente na questão da flexibilidade do lado do educador, proporcionando uma experiência de aprendizado mais eficaz e envolvente para os jogadores. Busca-se criar uma ferramenta que permita ao educador ter liberdade sobre os assuntos e temas a serem abordados mediante um *framework* interno de personalização do jogo que seja cativante e explore de forma inovadora os aspectos de sincronização e tempos de repostas.

1.3 METODOLOGIA

Para realizar o projeto, implementação e avaliação do jogo, as seguintes etapas foram seguidas:

a) Projeto: a etapa de projeto foi conduzida de maneira personalizada, com a criação de soluções e processos exclusivos. Nenhum *framework* de *game design* ou documentação foi desenvolvida, permitindo uma maior flexibilidade e adaptação às necessidades específicas do projeto.

b) Implementação: Para simplificar a implementação do jogo, foram escolhidas a *game engine Unity (Unity Technologies)* e a sua nova biblioteca *Netcode for GameObjects* que tem o papel de abstrair a lógica de rede para ser possível focar no desenvolvimento do jogo em si.

c) Avaliação: Para a avaliação do jogo, foram realizados testes locais ao longo do desenvolvimento e um teste remoto ao final.

1.4 ORGANIZAÇÃO DO TRABALHO

No capítulo 2, serão introduzidos conceitos básicos necessários para o entendimento deste trabalho. No capítulo 3 explora-se o projeto do jogo, seus objetivos, dinâmicas e mecânicas. No capítulo 4 discute-se a implementação do jogo, as tecnologias e arquiteturas utilizadas e os desafios encontrados com as soluções escolhidas. No capítulo 5 será feita uma análise do produto final obtido e uma avaliação do processo de desenvolvimento. Finalmente, no capítulo 6 concluímos o trabalho.

2 CONCEITOS BÁSICOS E JOGOS DE REFERÊNCIA

Este capítulo tem por objetivo discutir os conceitos básicos necessários para embasar o restante do trabalho. Na seção 2.1 serão explorados os jogos na educação, seus possíveis usos e benefícios. Na seção 2.2 são definidos alguns conceitos do que são jogos multijogador em tempo real e alguns dos desafios envolvidos. Na seção 2.3 jogos similares que serviram de base para este trabalho são apresentados.

2.1 JOGOS NA EDUCAÇÃO

Os jogos sempre foram parte da vida na terra, sendo observados até mesmo no mundo animal (TYLOR, 1879). Os humanos há milênios cultivam a prática de jogos esportivos e lúdicos como algo cotidiano. Uma perspectiva pela qual se pode analisar os jogos diz respeito ao quão regrados eles são.

Jogos lúdicos e livres, como são os jogos praticados por crianças mais jovens e que ainda não interagem com outras, não usam regras e auxiliam muito no desenvolvimento sensorial e motor. Observa-se naturalmente no decorrer do crescimento das crianças o aumento da complexidade de suas brincadeiras. Ao mesmo tempo que os jogos vão ganhando regras mais rígidas, eles vão também tornando-se menos individuais e mais sociais.

Isso é interessante, pois podemos pensar na vida de uma pessoa como tarefas que precisam ser realizadas seguindo um conjunto de regras sociais. O percurso de uma pessoa durante sua existência pode facilmente se assemelhar ao que são estes jogos com regras e com participação dos outros indivíduos da sociedade. Castro e Tredezini afirmam que os jogos têm um papel de desenvolvimento social e comunitário muito relevante, além de diversos outros benefícios.

> O jogo pode ser considerado como um importante meio educacional, pois propicia um desenvolvimento integral e dinâmico nas áreas cognitiva, afetiva, lingüística, social, moral e motora, além de contribuir para a construção da autonomia, criticidade, criatividade, responsabilidade e cooperação das crianças e adolescentes (CASTRO; TREDEZINI, 2014).

Olhando por esse ponto de vista, fica mais fácil entender empiricamente como os jogos podem ser utilizados na educação como uma ferramenta que agrega no aprendizado, seja ajudando nos conhecimentos necessários para a vida quanto auxiliando no desenvolvimento intelectual e absorção de informações relevantes. Fernandes defende que estes jogos podem construir autoconfiança e praticar habilidades.

> Os jogos podem ser empregados em uma variedade de propósitos dentro do contexto de aprendizado. Um dos usos básicos muito importante é a possibilidade de construir-se a autoconfiança. Outro é o incremento da

> motivação. (...) um método eficaz que possibilita uma prática significativa daquilo que está sendo aprendido. Até mesmo o mais simplório dos jogos pode ser empregado para proporcionar informações factuais e praticar habilidades, conferindo destreza e competência (FERNANDES et al., 1995).

Quando pensamos em jogos digitais, consideramos os mesmos princípios, mas atualizados conforme o avanço tecnológico e da sociedade e suas formas de socialização. Quando usados na educação, acabam tendo enfoques mais específicos como desenvolvimento de algum tipo de habilidade motora ou obtenção de conhecimentos de uma área específica. Ademais, não precisam ser jogos especificamente. A natureza interativa de programas educacionais, como aplicativos de estudo e plataformas de ensino à distância, os torna candidatos naturais para a aplicação de mecânicas e elementos de jogos, ou seja, *gamificação*. Um exemplo prático disso é o famoso aplicativo de estudo de línguas, Duolingo, que faz uso destes artifícios para tornar o aprendizado algo mais interativo, divertido e recompensador (MUNDAY, 2017). Arthur Naiman diz que os jogos não podem ser distinguidos facilmente de programas educativos, o que reforça o uso da *gamificação*.

> *I don't think it makes sense to distinguish too strictly between teaching programs and games. If a teaching program is good, it feels like a game; if a game is good, it teaches you things (EMMENS, 1984).*

2.2 JOGOS MULTIJOGADOR EM TEMPO REAL

Estes jogos estão vastamente difundidos e há muita variedade, a ponto de serem caracterizados em diversos gêneros. Alguns dos mais famosos são os jogos de tiro em primeira e terceira pessoa, os de corrida, de esportes, de luta, de estratégia e de sobrevivência.

Todos eles têm a característica em comum de admitirem mais de um jogador ao mesmo tempo. Em alguns gêneros, poucos por vez, como os jogos de luta em que participam apenas dois jogadores simultaneamente. Outros são jogados por muitas pessoas ao mesmo tempo, como os famosos MOBAs (*Multiplayer Online Battle Arena*) em que, em geral, dez jogadores são divididos em dois times de 5 componentes. Existem ainda aqueles jogados por uma quantidade muito grande de participantes simultâneos, os conhecidos como MMORPGs (*Massive Multiplayer Online Roling Playing Game*). Estes chegam à casa dos milhares de jogadores em momentos de pico.

Os desafios principais no desenvolvimento desse tipo de *software* aumentam principalmente com a quantidade de jogadores simultâneos que se quer suportar e se devem principalmente pelo fato da comunicação entre os elementos do jogo não ocorrer de forma instantânea. Isso gera demandas para se lidar com diferentes tipos de atrasos de comunicação e a consequente necessidade de sincronização do estado do jogo.

A topologia utilizada para viabilizar essa comunicação pode ser variada e depende também dos objetivos que se deseja alcançar. Na maioria das vezes observa-se o uso de

uma topologia padrão de cliente e servidor, mas é possível também encontrar exemplos de jogos que fazem uso da topologia *peer-to-peer* em que um dos clientes é o *host* e controla o estado do jogo para os outros clientes. Assim como existem exemplos de *relay*, em que um dos clientes também é o host do jogo, mas clientes comunicam-se com um servidor *relay* e não diretamente entre si.

Os jogos multijogador em tempo real são hoje também uma grande forma de pessoas interagirem e sentirem-se parte de uma comunidade. Muitos deles contam com diversas funções de socialização e comunicação, além de incentivarem criadores de conteúdo, comunicação via fóruns na web e a criação de laços e relações que rodeiam o universo dos games.

No caso deste trabalho, o objetivo é desenvolver um jogo que admita uma quantidade grande de jogadores simultâneos, mas em uma escala bem menor da que aquela que um MMORPG suportaria.

2.3 JOGOS SIMILARES

Os jogos utilizados como inspiração na construção deste trabalho são os hoje conhecidos jogos *IO*. São jogos que funcionam em um navegador web, caracterizados por um estilo de arte minimalista e foco na competição entre os jogadores em uma partida ou *lobby*. É importante salientar que o termo "*IO*"neste contexto se refere a um TLD (*top level domain*) que é a sigla utilizada para identificar o tipo de domínio de um site na internet. No caso dos jogos *IO*, o TLD *"IO"* é frequentemente utilizado, mas não é uma característica obrigatória. Dentre os motivos deste TLD ter se popularizado dentre os jogos desta categoria, pode-se citar a facilidade de memorização, o fato de ele ter se associado à ideia de jogos online e a alta disponibilidade que facilitou sua proliferação. As figuras 1 e 2 mostram exemplos destes jogos de referência.

O jogo Agar.io será a principal referência utilizada nesse trabalho. Nele, os jogadores coletam itens pelo espaço de jogo e aumentam sua massa e tamanho de acordo com sua pontuação. A velocidade de movimento varia de forma inversamente proporcional ao tamanho do personagem. Ao colidirem com outros jogadores, o maior deles absorve a massa do adversário e cresce de tamanho. O objetivo do jogo é que você acumule o máximo de massa possível e se mantenha o maior personagem do jogo.

Existem algumas customizações do jogador e alguns recursos no espaço de jogo e de mecânica de jogo que não serão aplicados. Um exemplo é o círculo verde à direita da imagem, de borda dentada, chamado de vírus. Ele tem o intuito de atrapalhar a movimentação dos jogadores, pois divide o personagem em vários menores quando se colide com ele.

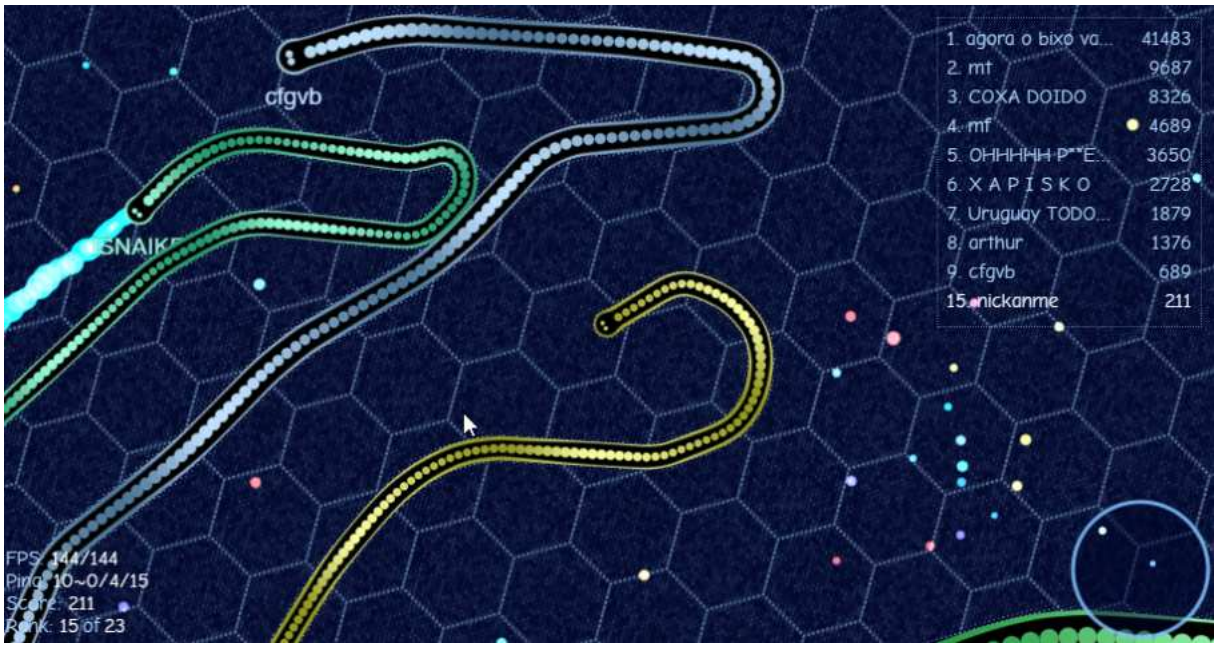
A divisão de massa pode também ocorrer de forma voluntária. Ao apertar a barra de espaço, o jogador divide sua massa pela metade e dobra a quantidade de personagens que

Figura 1 – Tela de jogo do Agar.io

Disponível em: https://agar.io/ Acessado em 06/05/2024 às 23:32

ele controla. Ao apertar a tecla W, pode-se também jogar para frente pequenas partes de massa em outros jogadores. Estas são todas mecânicas bem específicas deste jogo que não farão parte do projeto.

Figura 2 – Tela de jogo do Gulper.io



Disponível em: https://gulper.io/ Acessado em 06/05/2024 às 23:32

O jogo Gulper.io funciona de forma bastante similar, mas com uma mecânica levemente diferente e mais simples. Os itens coletados aumentam o comprimento do personagem

e o jogador pode usar a barra de espaço ou o botão esquerdo do mouse para aumentar sua velocidade de movimento. O objetivo é se manter o maior possível sem colidir com o corpo dos outros adversários.

2.4 TRABALHOS RELACIONADOS

Embora a literatura apresente uma vasta gama de trabalhos relacionados a jogos educativos, a maioria se concentra em modalidades que não se encaixam na proposta multijogador em tempo real, sendo de outros estilos e com propostas diferentes. Em geral, estes jogos têm um foco em alguma área de conhecimento específica, ou buscam ensinar e desenvolver um tema individual, se afastando da ideia principal deste trabalho.

3 PROJETO DO JOGO

Este capítulo tem por objetivo detalhar o projeto do jogo e todas as questões que permeiam esta proposta.

Existem *frameworks* e métodos mais formais de se projetar um jogo, o mais conhecido deles sendo o *MDA* (*Mechanics, Dynamics and Aesthetics*) (KUSUMA et al., 2018). Porém, por entender que o uso do *framework* pode de certa forma limitar a liberdade criativa e por este jogo ter necessidades e objetivos bem específicos, o autor optou por fazer um game design através do seu conhecimento tácito acerca do tema, acumulado ao longo de sua experiência.

Pelo mesmo motivo não foi elaborado um *GDD* (*Game Design Document*), prática comum no design de games.

Na seção 3.1 serão tratados os objetivos do jogo. Na seção 3.2 serão descritas as dinâmicas utilizadas dentro de jogo. Na seção 3.3 serão exploradas as mecânicas de jogo. Na seção 3.4 se discutirá a proposta pedagógica por trás do trabalho. Por fim, na seção 3.5 serão discutidas questões de *design* adicionais que são relevantes para o jogo proposto.

3.1 OBJETIVOS DO JOGO

O objetivo principal é criar um jogo multijogador em tempo real que exija participar de desafios com outras pessoas, demandando velocidade de raciocínio, de reação e algum conhecimento específico.

Aliado ao entretenimento gerado pelos embates entre jogadores, busca-se desenvolver habilidade motora através da necessidade de rapidez de resposta e também a obtenção de conhecimento mediante um sistema de perguntas e respostas de conteúdo editável pelo educador.

Este jogo destina-se a qualquer tipo de público-alvo em idade suficiente para ler e escrever respostas para as perguntas, que por serem configuráveis, permitem ajustes para qualquer nível de dificuldade ou área de conhecimento.

3.2 DINÂMICAS DO JOGO

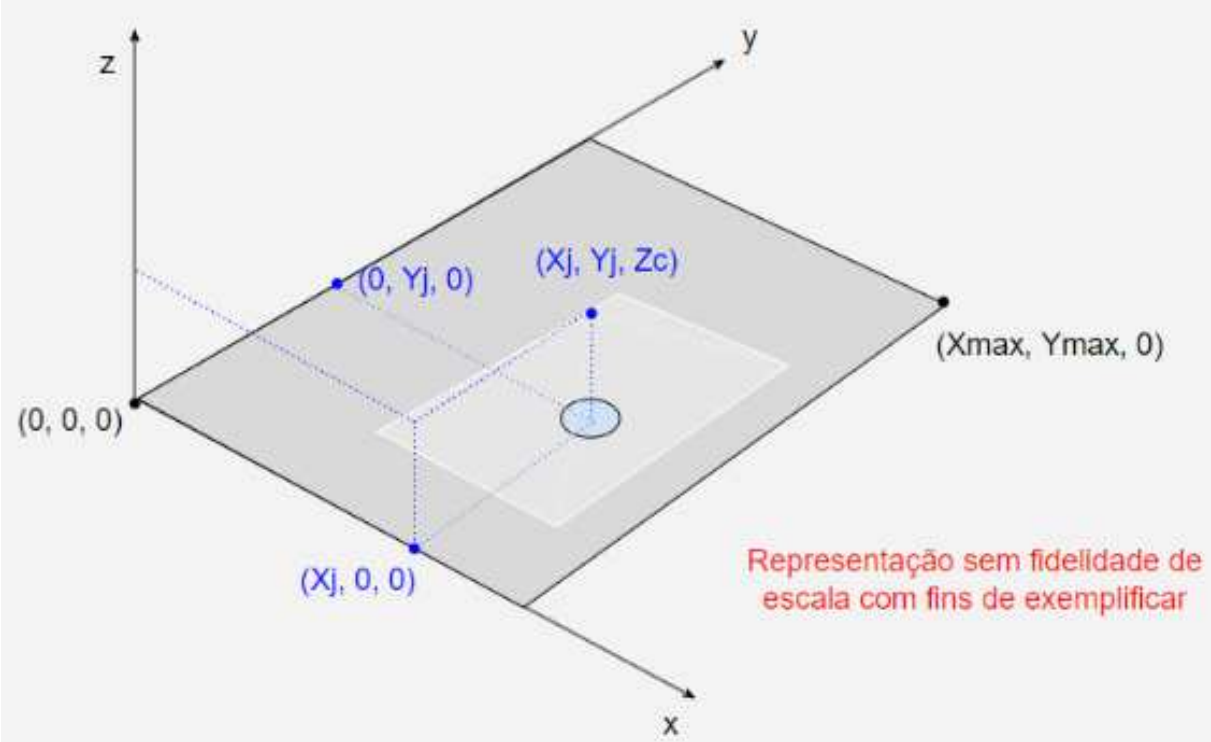
A seguir detalhamos as dinâmicas de jogo e as interações entre as entidades. Na subseção 3.2.1 descreve-se o espaço de jogo e visão do jogador. Na subseção 3.2.2 são detalhadas as características do jogador. Na subseção 3.2.3 será explicado como funcionará a pontuação. Na subseção 3.2.4 são detalhadas as características dos coletáveis. A subseção 3.2.5 explora o que compõe o desafio e como ele funciona. Por fim, a subseção 3.2.6 demonstra a tela de jogo e o que será apresentado na interface visual.

3.2.1 Espaço de Jogo

O jogo acontece em duas dimensões, em um plano. Tal plano tem um tamanho fixo, sendo definido no código como um valor imutável para cada instância do jogo executada.

A visão do jogador é dada por uma câmera centrada em seu personagem a uma altura fixa, criando uma visão de cima para baixo (conhecida popularmente como *top-down view*), perpendicular ao plano. Um esboço desta configuração pode ser vista na figura 3.

Figura 3 – Esboço do espaço de jogo em visão isométrica



A câmera centrada no personagem do jogador mostra apenas uma fração do espaço total de jogo. Ela tem um campo de visão fixo e não configurável. Isso significa que o quanto se enxerga no entorno do jogador que ela segue é sempre uma área constante. Um esboço da visão do jogador está exemplificado na figura 4.

3.2.2 O Jogador

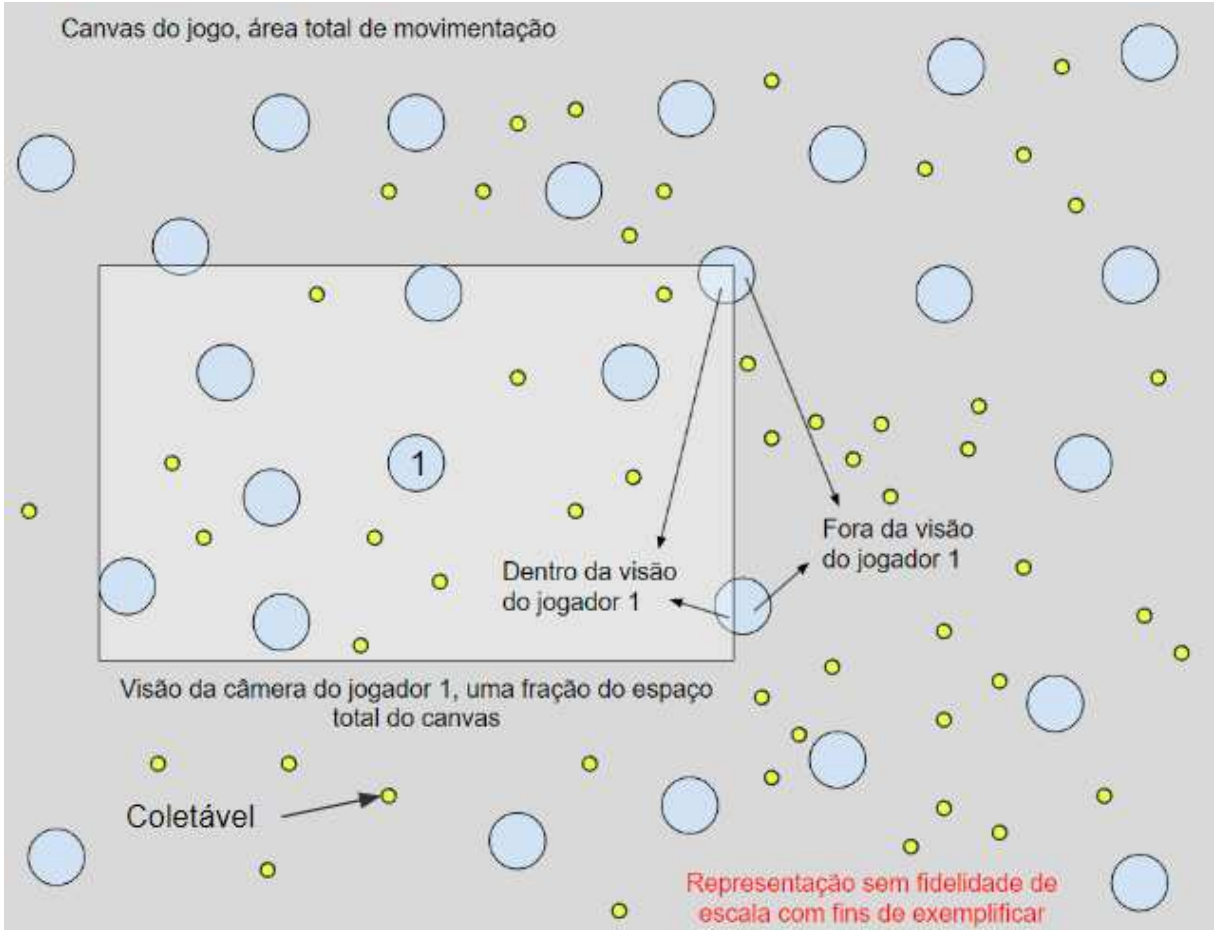
Uma instância de um jogador é definida por:

a) Um apelido escolhido pelo jogador;

b) Uma cor aleatória atribuída pelo jogo.

Durante a partida, cada jogador terá uma quantidade de vidas e o seu apelido mostrados durante todo o tempo acima de seu personagem, para deixar essas duas informações claras aos outros jogadores e para si.

Figura 4 – Esboço da visão do jogador



O número de vidas de um jogador pode chegar a um valor máximo definido como uma constante no código do jogo. A partir deste valor, mesmo que ele execute uma ação que deveria aumentar sua quantidade de vidas, ela permanecerá a mesma.

3.2.3 Pontuação

Cada jogador tem atrelada a si uma pontuação. O objetivo do jogo é alcançar uma determinada pontuação antes dos oponentes.

Ao conectar-se à partida, o jogador recebe uma pontuação inicial de zero pontos. A partir deste instante, o jogador pode pontuar de duas formas diferentes:

a) Vencendo desafios contra outros jogadores, o que renderá muitos pontos;

b) Chocando-se com os coletáveis espalhados pelo espaço de jogo, o que renderá poucos pontos.

Em nenhum momento é possível perder pontos. A única exceção é a eliminação do jogador que zera seus pontos caso ele queira começar novamente.

O intuito é que o jogador possa escolher como deseja ganhar seus pontos e alcançar a vitória. A primeira via é ganhar pontos por meio de coletáveis enquanto movimenta-se pelo espaço de jogo e evita-se confrontos com os oponentes. A segunda é garantir vitórias em desafios contra outros jogadores, o que exigirá ser mais habilidoso que eles, mas trará uma recompensa maior.

3.2.4 Coletáveis

A partida tem uma quantidade fixa de coletáveis. Estes são pequenos itens espalhados pelo espaço de jogo, que podem ser coletados pelos jogadores ao colidirem com eles. Ao coletar um desses itens, o jogador recebe uma pequena quantidade de pontos, definida como uma constante no código do jogo, e o coletável em questão é destruído. Neste momento, um novo item aparece em algum lugar aleatório do espaço de jogo, mantendo assim o número constante de coletáveis espalhados pela área jogável.

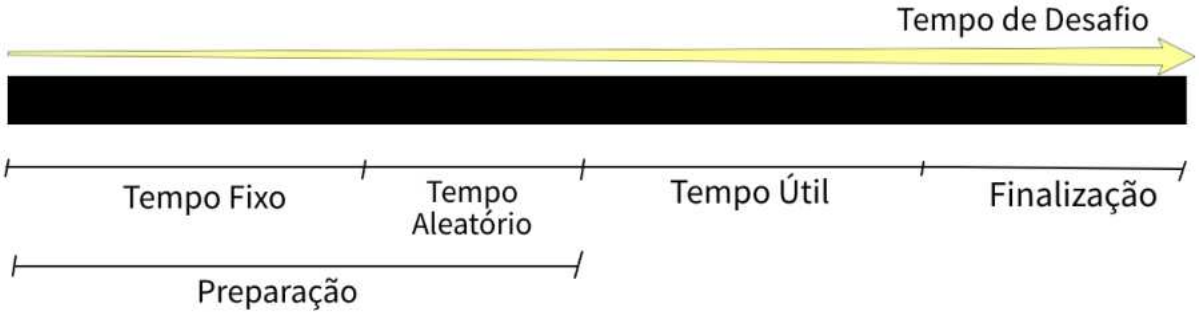
3.2.5 Desafio

Um desafio acontece toda vez que um jogador colide com outro (no caso de ambos estarem disponíveis). Esta é a parte principal do jogo. Esta mecânica acontece exclusivamente entre dois jogadores. Ao fim dele, o jogador vencedor recebe muitos pontos, quantidade essa definida como uma constante no código do jogo. Ele também recebe uma vida, desde que não tenha atingido o limite anteriormente. O perdedor não tem seus pontos alterados, mas perde uma vida. Caso chegue a zero vida após um desafio, ele é eliminado da partida e precisa entrar novamente, passando pelo mesmo processo de conexão da primeira vez. Existe também a possibilidade de um desafio terminar em um empate. Nesse caso, ambos os jogadores são penalizados em uma vida, sem alteração dos pontos.

Enquanto durar o desafio, ambos os participantes tornam-se imunes a novas colisões, ficando consequentemente impedidos de entrar em um novo desafio com um terceiro jogador. Esta imunidade tem um efeito visual de transparência para que os demais oponentes saibam que não se chocarão com eles. Após o término do desafio, os participantes continuam imunes a colisões por mais uma quantidade de tempo definida como uma constante no código do jogo e recebem um aumento de velocidade de movimento de igual duração, permitindo que se recoloquem no espaço de jogo e escolham uma posição favorável para continuar.

O ciclo de vida do desafio é caracterizada por três intervalos de tempo principais. Um tempo de preparação, um tempo de desafio útil e um tempo de finalização. Isto pode ser visto graficamente na figura 5.

Figura 5 – Detalhes do tempo de vida do desafio



O tempo de preparação é composto de um tempo fixo mais um tempo aleatório com o intuito de tornar a primeira ação do jogador mais reativa e menos dependente de memória muscular que seria criada após várias participações em desafios com um tempo fixo.

O tempo de desafio útil é o intervalo em que, de fato, os jogadores executam ações durante o duelo. A mecânica do duelo será descrita na próxima seção.

O tempo de finalização é a parte final do desafio, quando os resultados são apresentados aos jogadores.

3.2.6 Interface Visual

A tela de jogo mostra, no canto superior esquerdo, a maior pontuação alcançada por algum jogador desde a criação da partida, mesmo que ele já tenha se desconectado. No canto superior direito pode-se ver um *ranking* dos jogadores conectados. No código, como uma constante, é possível definir quantos participantes serão mostrados nesta lista. Em sua própria tela, o jogador sempre consta no ranking com sua posição atual. Em caso de estar pior colocado que os primeiros jogadores que aparecem, ele aparece em último lugar com sua colocação indicada. A figura 6 mostra um esboço desta tela.

3.3 MECÂNICAS DO JOGO

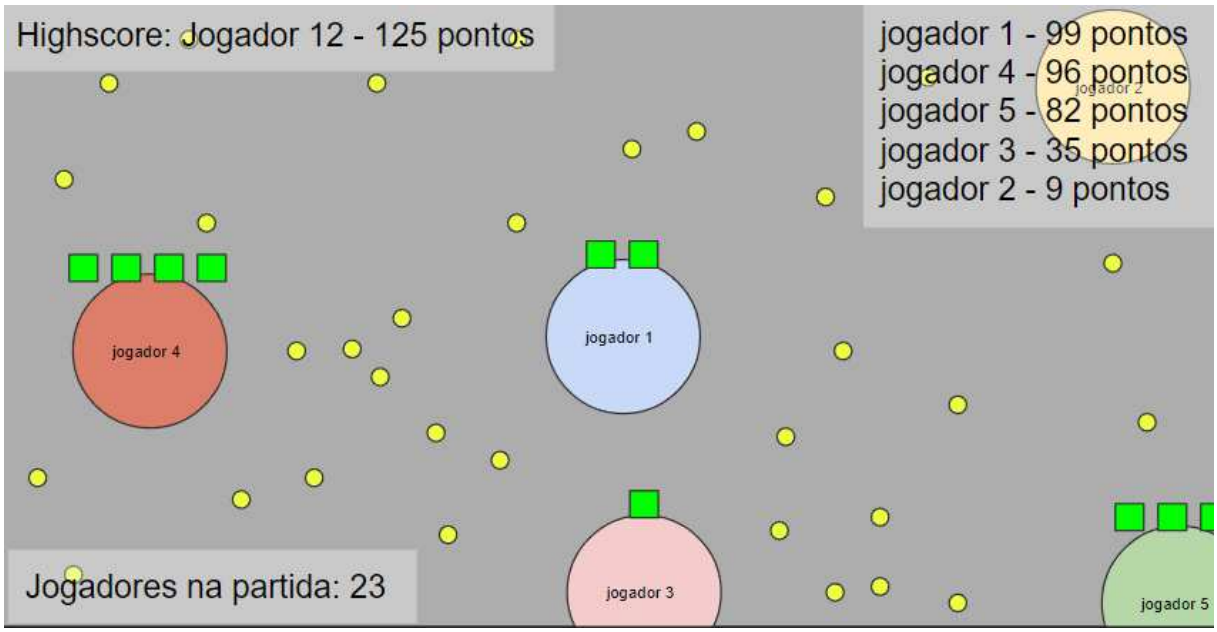
Esta seção explora as mecânicas de jogo. A subseção 3.3.1 descreve a mecânica de movimentação enquanto a subseção 3.3.2 explora a mecânica do desafio.

3.3.1 Movimentação

A ação básica de um jogador é a movimentação nos eixos horizontal e vertical, que pode ser realizada por meio das setas direcionais do teclado ou pelas teclas das letras WASD do teclado, conforme a preferência do jogador.

A velocidade desse movimento é fixa, a exceção de momentos específicos do jogo de forma autoritária e sem controle por parte do jogador.

Figura 6 – Esboço da tela de jogo



3.3.2 Desafio

O desafio começa quando dois jogadores colidem. Neste instante, ambos perdem a habilidade de movimentar-se e tornam-se imunes a novos contatos com outros oponentes.

No início do jogo é decidido qual tipo de desafio utilizar dentre os dois possíveis. A opção padrão é a randômica, em que a cada desafio, uma das duas opções é escolhida aleatoriamente pelo jogo com probabilidade uniforme. As possibilidades são:

a) Desafio de pressionamento de botão

- Os dois jogadores que estão em desafio lerão um aviso de que deverão pressionar a barra de espaço mais rápido que o adversário ao sinal verde. Aparece simultaneamente um sinal visual, na cor vermelha, da fase de preparação, indicando que ainda não é o momento de pressionar a tecla. Nesta etapa, cada um dos jogadores tem um limite de erros, para o caso de apertarem o botão antes do momento correto. Esta quantidade é configurável na criação do jogo via uma constante no código. Cometer um erro exibe um *feedback* negativo visual e remove uma tentativa. Caso sobre zero tentativa, deve-se esperar o jogador adversário terminar o desafio para saber se o resultado foi uma derrota ou um empate.

- Ao entrar na fase útil, o desafio é liberado para o *input* dos jogadores, o que é indicado pela mudança de cor do sinal visual, enquanto um contador decrescente indica o tempo restante até o término desta fase. Mesmo que conclua o desafio antes do término do contador, o participante deverá aguardar o fim deste período.

b) Desafio de pergunta e resposta

- Os dois jogadores que estão em desafio lerão um aviso de que deverão responder uma pergunta mais rápido que o adversário ao sinal verde. Aparece simultaneamente um sinal visual da fase de preparação, na cor vermelha, indicando que ainda não é o momento de responder.

- Ao entrar na fase útil, o desafio é liberado para o *input* dos jogadores, o que é indicado pela mudança de cor do sinal visual, e a pergunta surge na tela dos jogadores. Adicionalmente, um contador decrescente indicando o tempo restante até o término desta fase é exibido. Também é adicionada à tela também um campo de texto. O foco é direcionado automaticamente ao campo, para não ser necessária qualquer ação por parte do jogador antes de digitar a resposta. O contador só para no momento em que o jogador pressionar enter após digitar a resposta correta. Caso tenha acertado uma das possíveis respostas, ele recebe um *feedback* positivo visual. Se for pressionada a tecla enter, mas a resposta estiver errada, um *feedback* negativo visual é gerado. Mesmo que o jogador conclua o desafio antecipadamente, ele deverá aguardar o fim de todo o período.

A fase de finalização e o término do desafio acontece de forma muito semelhante nos dois tipos, com uma diferença sutil no caso de pergunta e resposta, pois as respostas aceitas aparecem na tela.

Com o começo da fase de finalização, o contador é removido e os resultados são mostrados aos jogadores, com destaque para o vencedor. Além disso, são apresentadas as respostas que seriam aceitas como respostas corretas, se este for o caso. Esta tela dura um tempo fixo.

Finalizado o desafio, o vencedor recebe sua pontuação e a vida adicional. O perdedor tem uma de suas vidas removida. Ambos os jogadores recebem um aumento temporário de velocidade de movimento para poderem se reposicionar antes que voltem a poder colidir com outros jogadores.

3.4 PROPOSTA PEDAGÓGICA

A proposta do jogo é propiciar um ambiente divertido e desafiador em que as motivações iniciais do projeto sejam atingidas, ao mesmo tempo em que se permite ao educador customizar as perguntas a serem respondidas e as possíveis respostas de forma fácil, rápida e intuitiva.

Dentre os arquivos do jogo, haverá um de extensão JSON em que deverão ser inseridas as perguntas e as respostas pelo educador. Os desafios buscarão nesse arquivo as questões a serem apresentadas, escolhendo uma de forma aleatória. Decidirão também, consultando estes dados, quais são as respostas válidas.

O arquivo terá uma lista de objetos compostos de uma chave *query* e uma chave *answer*, sendo *answer* uma lista de possíveis respostas e *query* uma *string* representando a pergunta. Um exemplo pode ser visto na figura 7.

3.5 CONSIDERAÇÕES ADICIONAIS DE PROJETO

Uma consideração importante a respeito dos desafios de questões foi a escolha de não diferenciar letras maiúsculas e minúsculas nas respostas. Isso pode inviabilizar alguns tipos de perguntas em que essa distinção seja importante, mas por serem estas uma porcentagem pequena das possíveis questões e levando em conta a simplicidade esperada de uma versão primária, este foi o caminho escolhido.

Apesar disso, a diferença de acentuação é levada em consideração, possibilitando que educadores possam exigir esse aspecto nas respostas corretas se ele for relevante.

Outro ponto a ser mencionado é a escolha por várias respostas possíveis diferentes. Devido à impossibilidade de se definir uma resposta única para diversas questões, e levando em conta o objetivo de gerar um *framework* flexível, optou-se por este caminho. Aquele que definir as questões tem total liberdade de listar as respostas possíveis e o que gostaria de considerar um erro ou não.

Utilizar uma única *string* como resposta pode dificultar a adição de respostas múltiplas no futuro. Isso exigiria alterações na chave JSON ou quebraria a compatibilidade com arquivos antigos. Um benefício do uso de uma lista já de começo é que o caso de resposta única já está coberto. Ao mesmo tempo, isso não cria um problema para o caso das maiúsculas e minúsculas, pois basta adicionar isto ao JSON sem necessidade de modificar uma chave já existente. É simples adicionar um booleano que defina se devem ser consideradas essas diferenças e tratar a ausência dele como um valor falso para manter o suporte aos arquivos antigos.

Ainda sobre o arquivo de perguntas e respostas, as escolhas feitas permitem o compartilhamento de arquivos JSON. o que é uma grande vantagem do *framework*. Permite-se, por exemplo, que diferentes educadores troquem os arquivos entre si via algum meio de comunicação, ou concatenem diferentes listas de perguntas e respostas em um novo arquivo. Também é possível que os próprios educandos populem um arquivo de perguntas e respostas junto do educador.

Sobre áudios e efeitos sonoros, a opção foi por não utilizar no primeiro momento. Seria útil que os *feedbacks* visuais citados em 3.2.5 e 3.3.2 fossem aliados a um *feedback* sonoro, mas não é absolutamente necessário.

Figura 7 – Exemplo de arquivo de questões

```
{
  "questions": [
    {
      "query": "Qual é a soma dos 5 primeiros números primos?",
      "answers": [
        "28",
        "Vinte e Oito"
      ]
    },
    {
      "query": "Qual é a capital do Brasil?",
      "answers": [
        "Brasília"
      ]
    },
    {
      "query": "Quantos mega bytes tem um giga byte?",
      "answers": [
        "1024",
        "1024MB",
        "1024 MB"
      ]
    },
    {
      "query": "Quantos bits tem um byte?",
      "answers": [
        "8",
        "oito",
        "oito bits",
        "8 bits"
      ]
    },
    {
      "query": "Como se escreve 'cachorro' em inglês?",
        "answers": [
            "dog"
        ]
    },
    {
      "query": "Escreva o valor 847 por extenso",
        "answers": [
            "oitocentos e quarenta e sete"
        ]
    },
    {
      "query": "Digite a seguinte palavra ao contrário: 'carteiro'",
      "answers": [
        "orietrac"
      ]
    }
  ]
}
```

4 IMPLEMENTAÇÃO DO JOGO

Este capítulo tem por finalidade descrever o processo de implementação do jogo e todas as questões que permeiam o assunto. Todo o código desta implementação pode ser encontrado no link do GitHub[1]. Os códigos citados no decorrer do texto encontram-se no apêndice A.

Na seção 4.1 serão apresentadas as tecnologias que foram utilizadas e suas motivações. Na seção 4.2 são discutidas as possibilidades de topologia possíveis e é defendida a escolha do autor. Na seção 4.3 será definida a arquitetura do jogo. A seção 4.4 explica de que forma é feita a sincronização de dados e as diferentes alternativas possíveis. Na seção 4.5 é explorada a implementação do jogador e suas particularidades. Analogamente, na seção 4.6 é explanada a implementação dos coletáveis do jogo e seu comportamento. A seção 4.7 detalha a implementação do desafio. Por fim, a seção 4.8 traz algumas considerações finais relevantes.

4.1 TECNOLOGIAS UTILIZADAS

Nesta seção discutem-se as tecnologias utilizadas e algumas escolhas feitas referentes a elas. A subseção 4.1.1 justifica a escolha da *game engine*. Na seção 4.1.2 são apresentados os *softwares* utilizados para suportar o desenvolvimento. Analogamente, a seção 4.1.3 comentará sobre bibliotecas utilizadas.

4.1.1 Game Engine

Nos primeiros protótipos deste trabalho, o intuito era que o desenvolvimento fosse feito sem frameworks ou *game engines*, justamente para entender as dificuldades que surgiriam e propor as soluções necessárias. Levando em conta os jogos *IO* já citados na seção 2.3 que serviram de base para este trabalho, os quais executam, em sua grande maioria, em navegadores, a linguagem escolhida havia sido o *JavaScript*, onde toda a implementação seria feita utilizando a *API* padrão de comunicação via *sockets*. Com os avanços dos estudos para o desenvolvimento, ficou claro que desenvolver a aplicação pretendida usando diretamente a API de *sockets* seria muito custoso.

Assim sendo, foi decidido que o trabalho seria desenvolvido por meio da *Game Engine* Unity *(Unity Technologies)*[2]. A escolha se deu pela familiaridade do autor com este ambiente de desenvolvimento e com a linguagem C#. Foi utilizada a versão 2022.3.10.

Uma *Game Engine* é um software intermediário que visa simplificar o processo de desenvolvimento de videogames. Ele possui um conjunto de ferramentas pré-construídas

[1] O código deste trabalho pode ser encontrado no GitHub em https://github.com/ruanramos/TCC2D
[2] Disponível em https://unity.com/

que fornecem elementos fundamentais para os jogos. Através dessas ferramentas podemos trabalhar com áudios, gráficos, simulações de física, luzes, editores de níveis e *scripts* programados para manipularem estes elementos conforme necessário.

4.1.2 Softwares Auxiliares

Softwares auxiliares são, geralmente, utilizados no desenvolvimento de jogos eletrônicos. Eles permitem estender as capacidades básicas de uma *Game Engine* nas suas áreas específicas.

Como o propósito é justamente que o jogo seja simples graficamente e não possua áudio inato, decidiu-se por não utilizar qualquer programa para auxiliar nestes aspectos. O desenvolvimento se deu unicamente utilizando as ferramentas simples disponibilizadas pela Unity e pelas suas bibliotecas.

4.1.3 Bibliotecas

Além das bibliotecas básicas da Unity de criação de textos, gráficos, inputs e algumas outras, duas bibliotecas importantes que serão utilizadas são a *Unity Netcode for Gameobjects*[3], a *Unity Multiplayer Tools*[4] e o *Quantum Console*[5].

A *Unity Netcode for Gameobjects*, como descrito na seção 1.3, ajuda na comunicação entre os componentes pela rede e abstrai a lógica mais complexa. Foi utilizada a versão 1.8.0. Esta é uma versão relevante, pois existem algumas diferenças na forma como se definem as *Remote Procedure Calls* (RPCs).

A *Unity Multiplayer Tools* agrega algumas ferramentas que podem facilitar o desenvolvimento de jogos multijogador, principalmente no tocante a simulações de condições de rede, visualização de estatísticas de rede e monitoramento de desempenho. Obviamente são instrumentos mais úteis em partes mais avançadas de desenvolvimento e testes de carga, ou para *debugs* mais específicos no tráfico de rede. Foi utilizada a versão 1.1.1.

O *Quantum Console* (*QFSW*) é uma biblioteca que integra um console dentro do jogo, extremamente útil para ver logs durante o desenvolvimento enquanto o jogo executa, além de possibilitar criação de comandos via código, facilitando tarefas de *debug*. Foi utilizada a versão 2.6.5.

Um último adendo relevante é a diferença entre duas bibliotecas de nomes parecidos, porém de finalidades distintas. A *Unity* oferece duas soluções principais para desenvolvimento de rede multijogador: *Netcode for GameObjects* e *Netcode for Entities*.

A solução *Netcode for GameObjects* é voltada para projetos *Unity* mais simples e padrões que usam o sistema de *GameObjects* tradicional. Ele gera desenvolvimentos mais

[3] Documentação disponível em https://docs-multiplayer.unity3d.com/netcode/current/about/

[4] Documentação disponível em https://docs.unity3d.com/Manual/com.unity.multiplayer.tools.html

[5] Disponível na Unity asset store em https://assetstore.unity.com/packages/tools/utilities/quantum-console-211046

rápidos e é compatível com alguns outros serviços da própria *Unity* que podem evitar o uso de servidores dedicados em alguns casos. A *Unity Netcode for Entities* utiliza o sistema ECS *(Entity Component System)* e é projetado para jogos competitivos de larga escala e que exijam alto desempenho, além de prover recursos mais avançados de predição, interpolação e gerenciamento granular de dados, permitindo enviar mais dados pelo tráfego de rede e com um melhor desempenho.

4.2 TOPOLOGIA DO JOGO

A topologia de um jogo no contexto de jogos multijogador se refere à maneira como os jogadores e o servidor (ou servidores) se conectam e interagem entre si. Escolher a topologia mais adequada é a primeira coisa a ser feita em um trabalho como este, pois é o que vai sustentar o resto do desenvolvimento.

Para escolher o tipo de topologia que será utilizada, muitos fatores devem ser levados em consideração, como o número de jogadores que se pretende suportar simultaneamente e localização geográfica destes jogadores.

Existem dois principais tipos de topologias quando se trata de um jogo onde jogadores podem jogar remotamente. A subseção 4.2.1 fala sobre a chamada topologia cliente-servidor, enquanto a subseção 4.2.2 comenta sobre a topologia *peer-to-peer*. Por fim, a subseção 4.2.3 discute qual foi a escolha do autor para este trabalho.

4.2.1 Cliente-Servidor

Cliente-servidor é a modalidade mais comum, onde os clientes se conectam a um servidor central que gerencia o estado do jogo. Trata-se de uma topologia escalável, que reduz a possibilidade de trapaças da parte dos clientes, uma vez que o controle do jogo fica do lado do servidor. A figura 8 exemplifica este modelo.

Nomeia-se este estilo de topologia onde o cliente apenas recebe informações de estado do servidor sem muito poder de controlar este estado localmente de servidor autoritário.

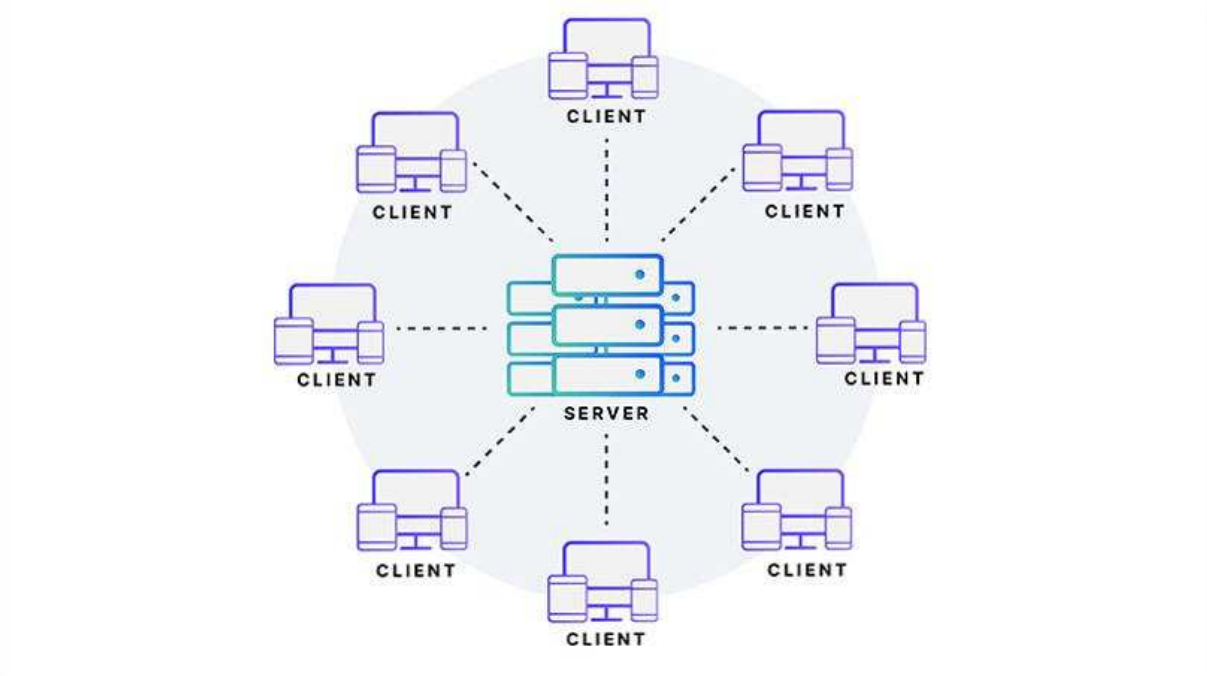
Dentre as desvantagens, podemos citar o alto custo de infraestrutura para servidores, a latência que varia conforme a distância do cliente ao servidor e o ponto único de falha.

4.2.2 Peer-to-peer (P2P)

Nesta modalidade, os jogadores se conectam diretamente entre si, sem necessidade de um servidor central. Neste caso, a lógica do jogo executa exclusivamente nos clientes, facilitando muito as tentativas de trapaça. A figura 9 exemplifica este modelo.

Também é comum surgirem problemas de escalabilidade, juntamente com a complexidade que é controlar o estado do jogo e a comunicação entre os jogadores. No entanto, é o modo mais recomendado se é necessário um baixo custo de infraestrutura e uma menor latência entre jogadores, sem um ponto de falha único.

Figura 8 – Modelo de servidor centralizado

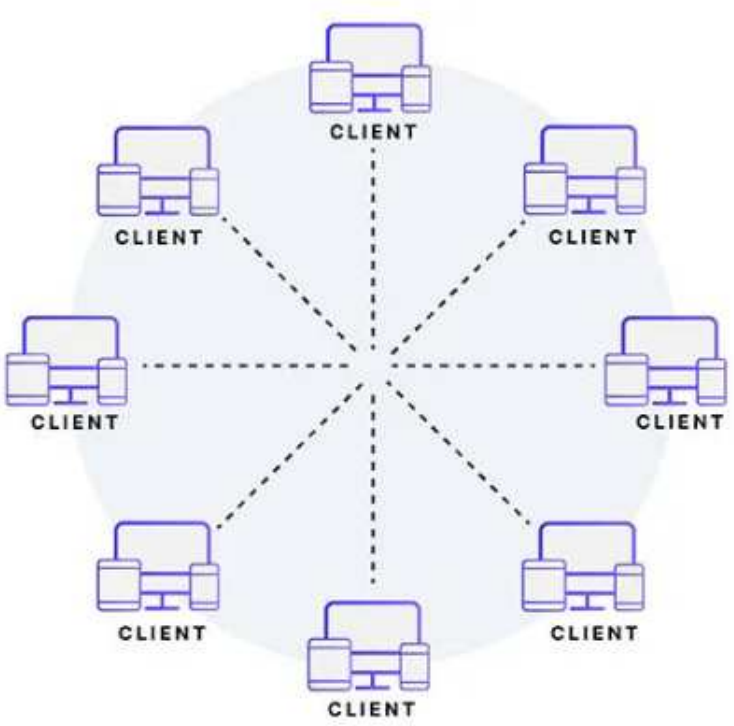


Fonte: https://unity.com/how-to/intro-network-server-models. Acessado em 06/05/2024 às 23:03

Uma variação bastante comum deste tipo de topologia é conhecida como *Server Peer-to-Peer*. Nela um dos clientes executa o lado servidor e o lado cliente. Desta maneira, os demais clientes precisam se comunicar apenas com um dos clientes, também chamado de *host*, reduzindo exponencialmente o custo de comunicação comparado ao *Peer-to-Peer* simples. Uma desvantagem desta variação é o que se chama de *Host Advantage*, que consiste no problema oriundo de os clientes terem uma latência ao se comunicar com o *host* e que para ele é mitigada. Dependendo do caso de uso, isso pode impactar o jogo muito negativamente e gerar uma sensação de frustração, dando a impressão de o host executar ações primeiro ou ver mudanças de estado primeiro. Este caso é mostrado na figura 10.

Uma segunda variação de topologia *Peer-to-Peer* é conhecida por *Servidor Relay*, onde existe um jogador *host* do jogo conforme a variação anterior, porém sem comunicação entre clientes. O que é feito para substituir esta comunicação é o uso de um *Relay* que se comunica com o *host* e também com os outros clientes. A diferença deste modelo para o de servidor se deve ao fato do *Relay* não ter qualquer controle sobre o estado do jogo, servindo apenas como ponte para a comunicação. Esta configuração pode ser vista graficamente na figura 11.

Figura 9 – Modelo de peer-to-peer



Fonte: https://unity.com/how-to/intro-network-server-models. Acessado em 06/05/2024 às 23:00

4.2.3 Escolha do Autor

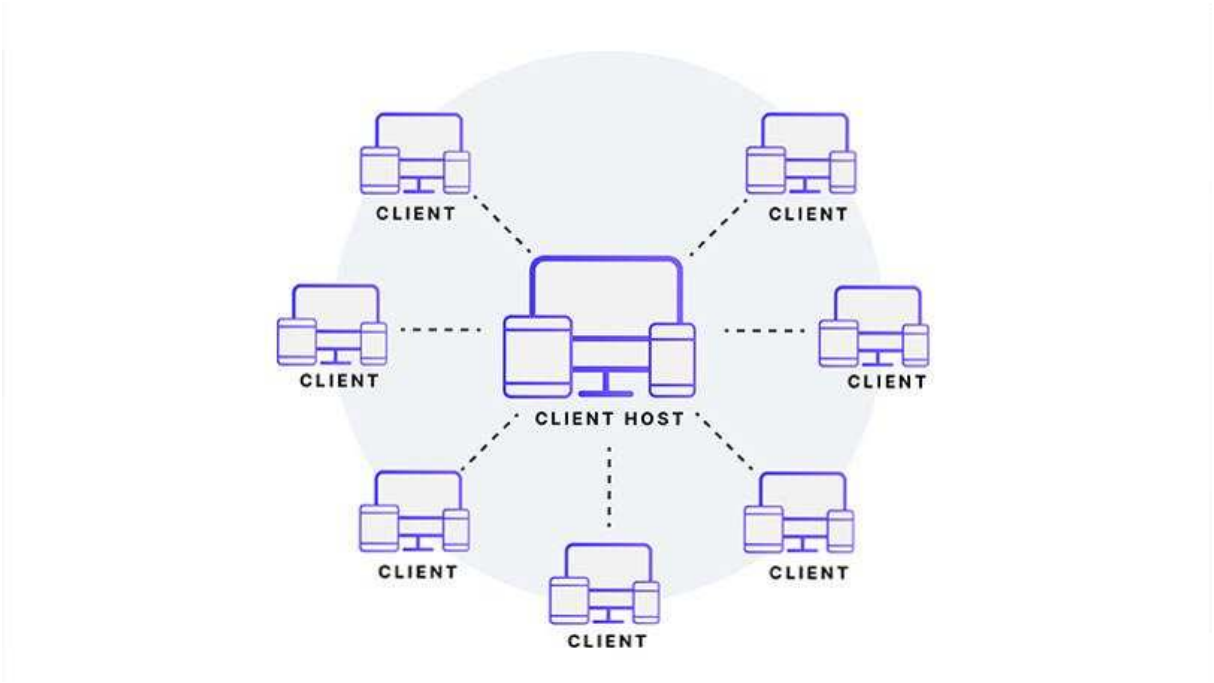
Para o projeto deste trabalho levou-se em consideração que o objetivo do jogo é impedir trapaças. Nesse sentido, optou-se pelo modelo cliente-servidor com o servidor autoritário, concentrando todo o estado do jogo e repassando aos clientes. Maiores detalhes serão explorados na seção 4.3.

4.3 ARQUITETURA DO JOGO

Definida a topologia a ser utilizada, parte-se para o detalhamento da arquitetura. As subseções seguintes destinam-se a explorar os elementos relevantes do desenvolvimento do jogo e como se relacionam entre si.

A subseção 4.3.1 abordará as constantes de configuração utilizadas para garantir maior flexibilidade no desenvolvimento. A subseção 4.3.2 abordará os objetos básicos e mais gerais. A subseção 4.3.3 discutirá sobre um assunto importante que é a sincronização dos objetos. Já a subseção 4.3.4 discorrerá acerca dos objetos de gerenciamento do jogo. Por fim, a subseção 4.3.5 analisará os objetos pré-fabricados, os *prefabs*.

Figura 10 – Modelo de cliente como host



Fonte: https://unity.com/how-to/intro-network-server-models. Acessado em 06/05/2024 às 23:03

4.3.1 Constantes de Configuração

Este é um elemento bastante importante do jogo, onde são definidas todas as constantes importantes a serem usadas. Muitos destes valores já foram citados no capítulo 3. A construção do jogo levou em conta que a maioria das decisões de projeto pode ser modificada neste arquivo antes da criação da partida. O código 1 mostra parte destas constantes.

Idealmente esta definição poderia ser feita via uma tela de configurações dentro do jogo antes da criação do *lobby*, mas por se afastar um pouco do escopo definido para o trabalho, a solução de um arquivo de constantes pareceu boa o suficiente.

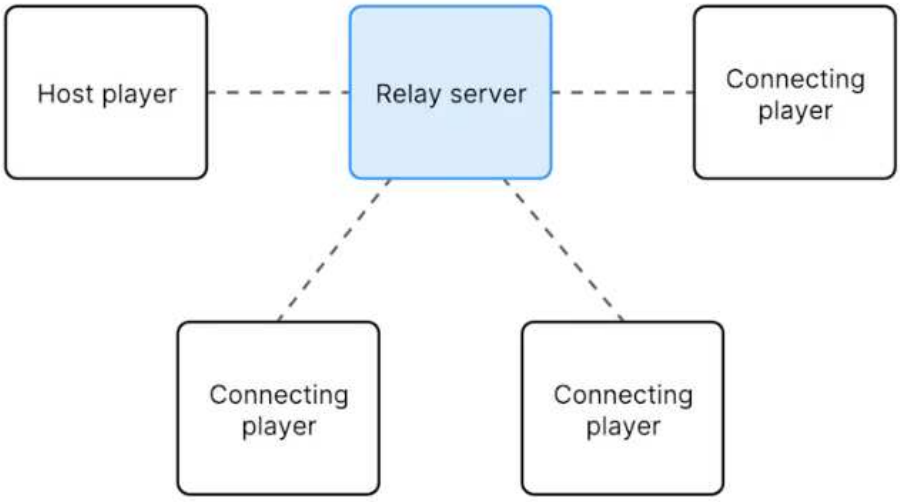
4.3.2 GameObjects Básicos

GameObject é como se chama o bloco básico no desenvolvimento de jogos na *Unity*. Eles são contêineres que ganham capacidades a partir de componentes anexados a eles.

Estes componentes podem ser visuais, de comportamento, de som ou de muitos outros tipos. Combinando estes diferentes componentes, é possível criar *GameObjects* de comportamentos únicos e flexíveis.

Agora deve-se entender os *GameObjects* básicos que foram utilizados neste jogo e quais suas funções na arquitetura do jogo. A subseção 4.3.2.1 trata da câmera de jogo. A subseção 4.3.2.2 mostra o objeto que compõe o espaço de jogo. Já a subseção 4.3.2.3

Figura 11 – Modelo de cliente como host e servidor relay



Fonte: https://docs.unity.com/ugs/manual/relay/manual/relay-servers. Acessado em 06/05/2024 às 23:14

trata do objeto que controla o que é mostrado na tela relacionado a pontuações.

4.3.2.1 Main Camera

Este é o *GameObject* padrão de câmera, que possui um componente *Camera* que controla todas as características da câmera principal de jogo. O componente foi modificado para que o tipo de projeção seja ortográfica.

4.3.2.2 Environment

Este *GameObject* tem por objetivo concentrar todos os objetos que estão relacionados com o espaço de jogo como seus *child objects*. Como este jogo é bastante simples, ele apenas concentra as paredes que delimitam o espaço de jogo e também a imagem de fundo. Estas paredes contam com componentes *Box Collider 2D* da parte de física da *game engine*.

4.3.2.3 Score UI Canvas

Este *GameObject* tem o papel de centralizar o que é mostrado na tela relacionado à pontuação dos jogadores. Ele tem dois objetos filhos, sendo um o que cuida da visualização do *score* do cliente local e o outro o que cuida do destaque do *ranking* do jogo, conforme explicado na seção 3.2.6.

4.3.3 Sincronização de GameObjects

Pode ser um pouco confuso de começo fazer uma aplicação na *Unity* que funcione simultaneamente nos clientes e no servidor, por isso, faz-se necessário entender bem um exemplo simples e básico.

Supondo uma situação onde o servidor é ligado e os jogadores 'jogador1' e 'jogador2' se conectam, temos um estado onde o servidor tem um espaço de jogo com um objeto representando o 'jogador1' e outro objeto representando o 'jogador2'. Simultaneamente, o cliente número 1 tem um objeto que o representa e que ele é o dono, e também um objeto que representa o jogador adversário cujo dono é o jogador adversário.

Fica claro que, com um servidor ligado e dois clientes conectados, existem dois objetos na rede representando os seis objetos locais. Generalizando, tem-se que, para um servidor online e n clientes conectados, sendo cada cliente representado por um objeto e ignorando os objetos no servidor, existem n^n objetos locais que representam esses clientes.

Pode-se pensar nisso como uma matriz quadrada simples onde os elementos p_{ij} representam o objeto do jogador i na tela do jogador j. Por exemplo, com três jogadores conectados, teríamos a seguinte matriz $\begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{pmatrix}$.

O elemento p_{23} representa o 'jogador2' na tela do 'jogador3'. Isto é importante por diversos motivos que serão vistos mais a frente, mas principalmente para que o cliente saiba qual objeto é o seu jogador local (sendo estes os elementos da diagonal principal). Os demais elementos são os objetos que representam um cliente nos demais clientes. Tendo solidificado este entendimento, é preciso entender um segundo ponto passível de confusões, que é o mesmo objeto estar presente em mais de um cliente e também no servidor.

Ao gerar um input que move o objeto do jogador, ele realiza uma translação localmente. Esse deslocamento precisa ser sincronizado com os outros clientes e isso é feito automaticamente pela biblioteca *Netcode for GameObjects* ao adicionar ao objeto um componente *NetworkTransform* e tendo a opção *SynchronizeTransform* ativa no componente *NetworkObject*. Importante também notar que se pode desativar sincronizações desnecessárias de rotação, escala ou movimentação em eixos que não serão modificados. Neste caso, foram habilitadas apenas sincronizações de posição no eixo xy, evitando envio de dados que não seriam utilizados.

Cada objeto que representa um jogador terá um *script* C# atrelado a ele como um componente. Este *script* é quem vai cuidar dos *inputs* do jogador e das ações que devem acontecer como consequência deles.

Isso significa que cada objeto destes que representa um jogador, mesmo que ele não seja o dono daquele objeto, executará o código que está no *script*. De forma mais prática, supondo um servidor com três clientes e um código que execute um *print* quando o jogador

pressiona a barra de espaço, pode-se observar no console que este *print* será executado uma vez para cada jogador, a menos que se faça uma checagem se o cliente é dono daquele objeto. A figura 12 exemplifica esta situação hipotética.

Figura 12 – Exemplo de scripts executando simultaneamente no pressionamento da barra de espaço

Quando um jogador move seu personagem localmente, é preciso que no código esteja claro que ele quer mover apenas o objeto o qual ele é dono, caso contrário todos os jogadores se moverão.

Localmente, o jogador tem a diferenciação visual de qual é seu jogador, ou seja, o objeto jogador que ele é o dono, via a cor do *label* acima dele, sendo azul a cor do jogador local e vermelho a cor dos adversários.

4.3.4 Manager GameObjects

Esta seção destina-se a explorar os *GameObjects* conhecidos como *managers*. Estes elementos têm um papel importante de concentrar atribuições específicas e ajudam na organização da arquitetura.

A subseção 4.3.4.1 trata do principal objeto de rede, o *NetworkManager*. A subseção 4.3.4.2 fala sobre o objeto que controla a interface visual de conexão. Na subseção 4.3.4.3 será discutido o objeto gerenciador geral do jogo. Por fim, nas sub subseções 4.3.4.4, 4.3.4.5 e 4.3.4.6 serão examinados os gerenciadores de coletáveis, desafios e questões, respectivamente.

4.3.4.1 NetworkManager

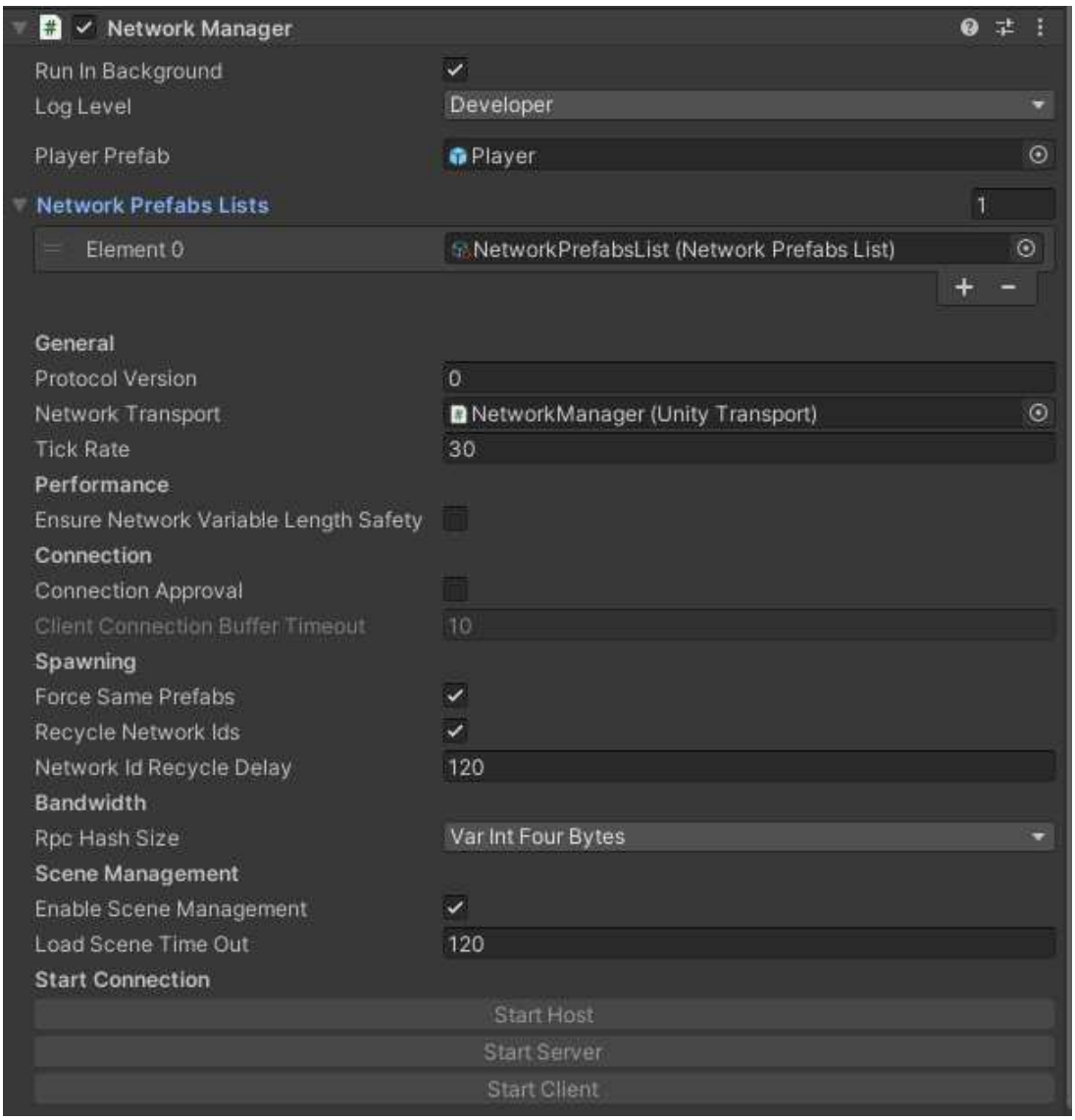
O *GameObject* central do jogo multijogador em tempo real é o *NetworkManager*. Ele tem o papel de gerenciar a comunicação entre todos os agentes do jogo, sincronização dos seus estados e também a configuração desta comunicação.

Aqui faz-se necessário fazer uma diferença entre dois conceitos que serão muito citados daqui para frente. Uma instanciação é a criação de um objeto dentro da cena de jogo, enquanto um *spawn* é a instanciação deste objeto na rede, via *NetworkManager*. A instanciação acontece apenas localmente, sem conhecimento dos outros elementos da rede. Caso ocorra uma instanciação sem ser seguida por um *spawn*, ocorrerá uma assimetria entre os nós. Além disso, é imprescindível que todo *GameObject* que seja instanciado na rede possua um componente *Network Object* atrelado a ele, o qual é responsável pela comunicação do objeto na Unity com o *NetworkManager* e o *Netcode for GameObjects*.

O objeto *NetworkManager* tem dois componentes necessários para o desenvolvimento da aplicação, o *NetworkManager* (figura 13) e o *Unity Transport* (figura 14).

O componente *NetworkManager* tem alguns campos importantes que devem ser observados:

Figura 13 – Componente NetworkManager



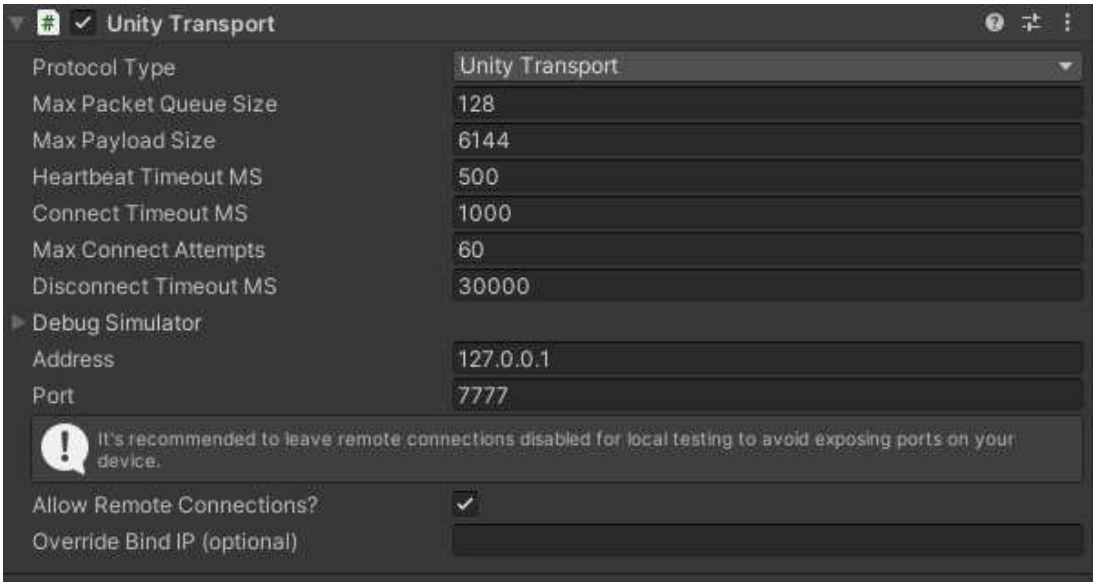
a) *Player Prefab*: um *prefab* (ou pré-fabricado) é um *GameObject* que foi construído, configurado e salvo como um modelo. Ele pode ser então reutilizado onde e quantas vezes forem necessárias. O *Player Prefab* é o *prefab* que representa o jogador. Ao definir este campo, quando um cliente se conecta, automaticamente o *NetworkManager* instancia e faz o *spawn* do jogador utilizando esse *prefab*.

b) *Network Prefabs List*: qualquer objeto que será instanciado na rede em algum momento pelo *NetworkManager* precisa estar adicionado a esta lista. Caso contrário, em tempo de execução irá ser lançado um erro de que não é possível fazer o *spawn* do objeto.

c) *Network Transport*: é recomendável o uso do *Unity Transport*, que é uma implementação de *NetworkTransport* permitindo o uso do *Unity Transport Package* como um protocolo de transporte de baixo nível para o *Netcode for GameObjects*. Dessa forma, consegue-se uma comunicação de rede baseada em UDP e multi-plataforma em um projeto *Unity*.

O componente *Unity Transport* precisa estar com o *address* correto de *localhost* quando o jogo for testado localmente. O caso remoto será explorado no capítulo 5.

Figura 14 – Componente Unity Transport



4.3.4.2 Network UI Manager

Uma das grandes dificuldades que surgiu no desenvolvimento deste trabalho está relacionada ao tempo entre realizar uma mudança, criar o *build*, executar um servidor e executar diversos clientes para testes. Esse procedimento foi repetido inúmeras vezes e precisava ser otimizado.

O *GameObject NetworkUIManager* tem por objetivo criar botões para iniciar um cliente ou um servidor via *NetworkManager* no canto da tela. Essa facilidade permite que sejam executados clientes e servidores de forma prática fora do editor da *Unity*, o que se fez muito necessário durante todo o processo. Ele também controla o input de modo de desafio (seção 3.3.2) e apelido do jogador.

4.3.4.3 Game Manager

O objeto *GameManager* contém um *script* C# *GameManager* cujas funções são diversas, mas principalmente relacionadas a guardar os clientes conectados e gerar sua posição randômica de *spawn*, mostrar na tela o ranking de jogadores e decidir qual o tipo de desafio foi escolhido por quem cria o *lobby*. Além disso, existem alguns métodos para exibir medidas de FPS e latência quando ativado, com propósito de *debug*. Este objeto tem também um componente *Network Object* atrelado a ele, permitindo que seja instanciado na rede.

4.3.4.4 Collectibles Manager

O objeto *CollectiblesManager* é o gerenciador de coletáveis do jogo. Ele contém um *script* C# *CollectiblesNetworkManager* que é quem comanda o comportamento dos coletáveis do jogo. Como é o padrão para a maioria dos gerenciadores, ele também precisa de um componente *Network Object* atrelado a ele.

4.3.4.5 Challenge Manager

O *Challange Manager* é o componente que controla os desafios do jogo. Ele segue o mesmo padrão dos outros, contendo um *script* C# *ChallengeNetwork* que coordena a criação e destruição dos desafios e seu ciclo de vida descrito na seção 3.2.5. Novamente, ele conta também com um componente *Network Object* para ser instanciado e ter acesso aos métodos de ciclo de vida na rede.

4.3.4.6 Questions Manager

O último gerenciador é aquele que comanda toda a lógica das questões que serão utilizadas nos desafios. Ele é o único que não é necessário em um caso específico, o de desafios de pressionamento de botão (item a) da subseção 3.3.2). Nos outros casos ele é imprescindível para garantir o funcionamento dos desafios de questões (item b) da subseção 3.3.2).

O papel deste gerenciador é fazer o carregamento das questões incluídas no arquivo de questões (exemplificado na figura 7), gerar uma questão para um desafio específico e checar as respostas dos clientes.

4.3.5 Prefabs

O jogo conta com três *GameObjects* pré-fabricados importantes utilizados em todos os nós da rede frequentemente. Os *prefabs* podem ser encontrados no diretório "Assets/-resources/prefabs".

As sub subseções 4.3.5.1, 4.3.5.2 e 4.3.5.3 analisam os objetos pré-fabricados do jogador, do coletável e do desafio, respectivamente.

4.3.5.1 Player

Este é o objeto que representa um jogador. Ele é formado por diversos componentes que definem o que é o *player*, junto de dois objetos filhos, sendo um o chamado *label*, que é o visual do número de vidas e do apelido do jogador acima do seu personagem, e o outro o *sprite2D* que é o visual de círculo renderizado pelo componente *SpriteRenderer*.

Os componentes do objeto pai do *prefab* dividem-se entre os que fazem parte da lógica de rede e os que fazem parte da lógica local.

Os componentes *Rigidbody2D* e *CircleCollider2D* são os que controlam a física de colisão localmente. Além disso, existe o *script* C# *PlayerCamera* que é quem faz o trabalho de mover a *Main Camera* seguindo o objeto do jogador conforme ele se move no espaço de jogo.

Os demais componentes são relacionados à lógica de rede. *NetworkObject* funciona conforme já discutido no caso dos objetos gerenciadores. Já *NetworkTransform* e *NetworkRigidBody2D* são os componentes adicionados pela biblioteca *Netcode for GameObjects* que já permitem a sincronização da posição e da física pela rede conforme discutido na seção 4.3.3. Isso significa que, ao mover o *Transform* local, que define a posição do objeto no espaço, automaticamente o objeto é movido na rede e sua posição é atualizada em todos os nós. Da mesma forma, ao colidir localmente, a colisão ocorre em todos os nós de forma sincronizada.

4.3.5.2 Collectible

Este é o objeto que representa um coletável do jogo. Ele é um objeto simples, sem objetos filhos, e bastante similar em estrutura de componentes ao que foi discutido sobre o *prefab* do jogador, também tendo a ideia de componentes locais e componentes de rede.

Os componentes locais são o *SpriteRenderer*, que tem o papel de renderizar na tela a forma de quadrado do coletável na cor que foi definida, e o *RigidBody2D* e *BoxCollider2D*.

Os componentes de rede também são similares aos do *prefab* do jogador. A única diferença sendo o *script* C# *CollectibleBehaviour*, que define o comportamento de cada coletável da partida.

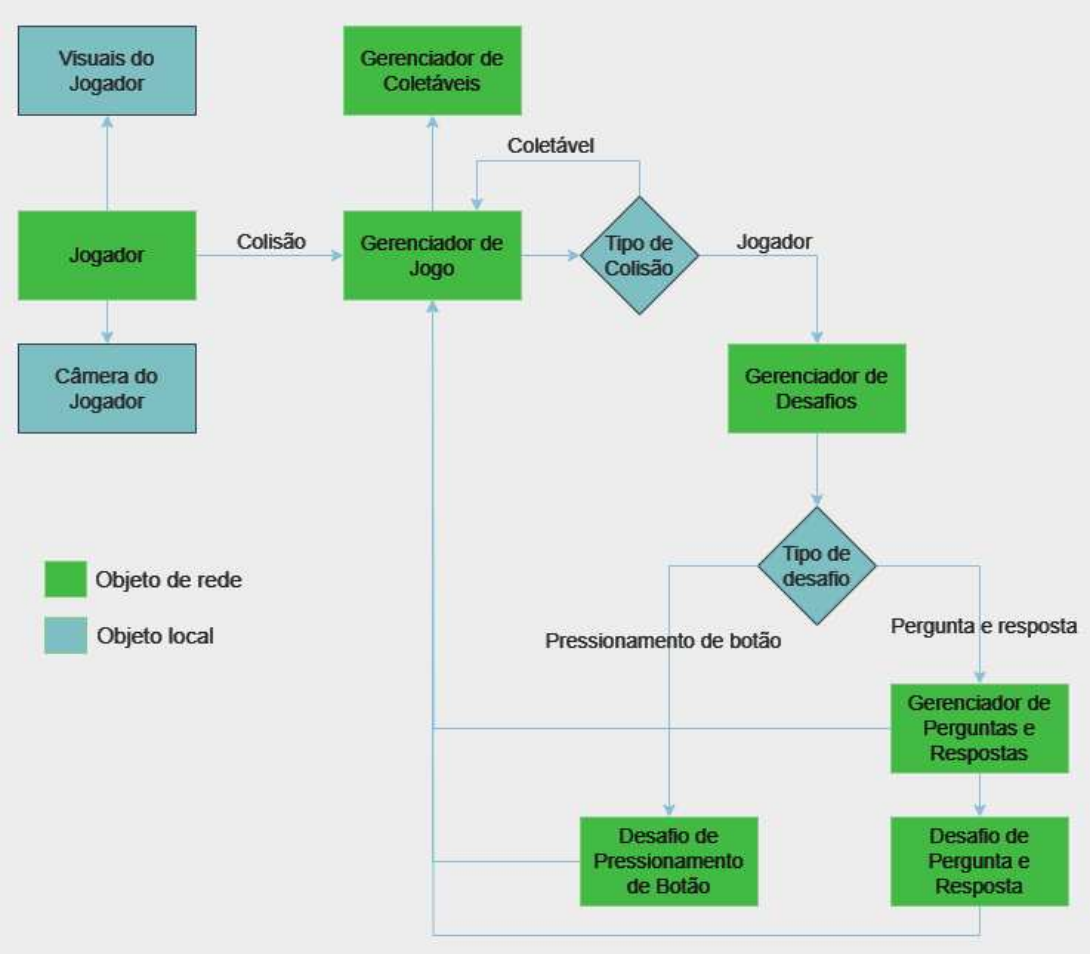
4.3.5.3 Challenge

Este é o *prefab* mais complexo do projeto, com diversos objetos filhos, que representa um desafio. Ele é dividido em um objeto *InnerCanvas* e um objeto *OutterCanvas*. Essa diferenciação foi feita com intuito de separar o que deve ser apresentado. O *InnerCanvas* trata dos elementos mais específicos de cada desafio, que mudam conforme o tipo de desafio. O *OutterCanvas* trata de elementos e informações mais gerais que precisam ser apresentados, como contador de tempo ou jogadores envolvidos.

4.3.6 Diagrama de Arquitetura Lógica

A figura 15 mostra um diagrama de arquitetura lógica com os objetos descritos no decorrer da seção. Os objetos destacados na cor verde são objetos instanciados na rede, enquanto os azuis são objetos de instanciação local.

Figura 15 – Diagrama de arquitetura lógica



4.4 SINCRONIZAÇÃO DE DADOS

Precisa-se entender que no *framework Netcode for Gameobjects* existem duas formas básicas de se trabalhar com sincronização de dados entre servidor e clientes. O primeiro destes métodos são as variáveis de rede. A segunda forma é via *RPCs*, ou *Remote Procedure Calls*. A subseção 4.4.1 se dedicará às variáveis de rede enquanto a subseção 4.4.2 discorrerá sobre as *RPCs*.

4.4.1 Variáveis de Rede

Variáveis de rede são adequadas para dados que mudam raramente, que não exigem consistência imediata e dados que podem ser tolerados com atrasos. Elas funcionam como variáveis normais, que podem ser alteradas em um nó da rede e são automaticamente modificadas também nos outros nós. Isso significa que, por exemplo, a quantidade de vidas de um jogador representada por uma variável de rede pode ser atualizada após um desafio com outro jogador e é automaticamente atualizada para todos os jogadores.

Essas variáveis de rede são *read-only* para clientes. Ou seja, podem ser escritas apenas pelo servidor, o que é muito útil para o caso deste trabalho onde tentamos fazer um servidor o mais autoritário possível.

Outro ponto relevante é que estas variáveis aceitam os tipos primitivos da linguagem e também tipos herdados da *Unity Engine*, como *UnityEngine.Color*. Para usar outros tipos não suportados nativamente pelo *framework*, precisamos implementar as interfaces *INetworkSerializable* e *System.IEquatable<T>* no tipo que queremos sincronizar. No caso deste trabalho, usaram-se apenas tipos já suportados. Para o nome do jogador, foi necessário usar uma variável de rede do tipo *FixedString32Bytes*, dado que não é possível sincronização com *Strings*.

Também foi feito uso de um recurso de *callbacks*. Ao realizar uma mudança em uma variável de rede, pode-se chamar um método para ser executado. Essa *callback* é adicionada no método *OnNetworkSpawn* e é removida no método *OnNetworkDespawn* do objeto de rede. No código 3 encontra-se um exemplo de como foi utilizada no desenvolvimento a variável de rede que representa a pontuação do jogador.

As vantagens de se usar estas variáveis são sua simplicidade na implementação, o menor tráfico de rede gerado comparado às *RPCs* e o controle mais granular sobre a sincronização de dados.

Uma dificuldade que se pôde observar ao usar variáveis de rede na implementação do trabalho é a inconsistência temporária entre os dados que pode existir entre os nós da rede.

4.4.2 RPCs

Remote Procedure Calls (*RPCs*) são métodos que clientes e servidores podem utilizar para se comunicar. Eles podem ser *ServerRPC*, no caso de ser um método executado no servidor invocado por um cliente, ou *ClientRPC*, um método executado no cliente invocado pelo servidor.

Rpcs devem ser utilizadas em ações que exigem consistência imediata, eventos que precisam ser acionados em todos ou em um subconjunto de clientes e também em interações mais complexas entre objetos.

Existem diversos tipos de uso de *RPCs* no desenvolvimento deste trabalho. Alguns exemplos são a que é utilizada para gerar e aplicar a cor aleatória de um novo jogador que se conectou na partida, a que mostra os resultados de um desafio para os clientes, a que guarda o resultado final do desafio de um jogador e a que envia os inputs ao servidor.

As vantagens de se usar *RPCs* são a consistência de estado, a flexibilidade de implementar diretamente o método que será executado e a facilidade de depuração com eventos em comparação com as variáveis de rede, principalmente por conta do controle da execução e das mudanças que serão feitas.

Dentre as desvantagens, podemos citar o maior tráfico de rede e a latência, que pode gerar atrasos perceptíveis devido à ida e volta dos dados pela rede. Esta latência pôde ser observada facilmente durante todo o desenvolvimento, até mesmo em ambiente local, dado que é necessária a comunicação entre os processos cliente e servidor da mesma forma.

4.5 IMPLEMENTAÇÃO DO JOGADOR

Nesta seção, discute-se de que forma foi feito o desenvolvimento das mecânicas e dinâmicas do jogador. A subseção 4.5.1 detalha como é feita a movimentação. Na subseção 4.5.2 se discorre acerca das vidas do jogador. Por fim, a subseção 4.5.3 explica o desenvolvimento relacionado à pontuação.

4.5.1 Movimentação do Jogador

A movimentação do jogador foi implementada via *RPC*. O método *Update* checa a cada *frame* se existe um *input* do jogador, representado por um vetor bi-dimensional. No caso de existir um vetor de magnitude diferente de zero, o cliente invoca uma chamada de método do servidor para realizar a movimentação.

O servidor então move o *transform* do objeto do jogador localmente, que consequentemente gera o movimento em toda a rede por conta do componente *NetworkTransform* presente no objeto. O código 5 demonstra de forma simplificada este controle da movimentação.

Desta forma, o cliente apenas informa de sua intenção de se mover e a movimentação é feita toda pelo servidor e repassada. Isso impede que o cliente tenha muito controle sobre

o jogo, evitando problemas de segurança. Apesar disso, se cria uma latência, conforme explicado na subseção 4.4.2.

No desenvolvimento deste trabalho não foram tomadas medidas para se reduzir essa sensação de atraso que ocorre nesta movimentação aos olhos do cliente. Uma forma comum de se lidar com esse problema é usar algum tipo de predição do lado do cliente, para gerar uma possível posição do jogador e evitar que a movimentação ocorra no cliente apenas após um RTT completo. É importante salientar que esse método pode gerar uma correção de posição após o recebimento de uma nova posição do servidor.

Um último ponto sobre a movimentação do jogador é que a câmera principal precisa de um pequeno *script* para seguir apenas o jogador local. O código 6 mostra este segmento de código.

Do lado do servidor, foi criada uma câmera que pode se mover livremente via botões de movimentação e uma mecânica de zoom com a roda do mouse, possibilitando assim uma espécie de modo espectador de tudo que ocorre no jogo.

4.5.2 Vidas do Jogador

As vidas do jogador são uma simples variável de rede do tipo inteiro. São disponibilizados métodos para adicionar ou remover vidas que são chamados segundo a dinâmica do jogo. Também é adicionada uma *callback* chamada na mudança deste valor, que faz a atualização das informações visuais das vidas do jogador.

4.5.3 Pontuação do Jogador

Similar às vidas, a pontuação do jogador também é uma variável de rede do tipo inteiro. Esta variável é modificada nas situações de vencer um desafio ou colisão com um coletável, sendo apenas um incremento simples sem maiores complexidades.

Esta variável de rede também conta com uma *callback* que é chamada após a mudança de valor, responsável por fazer a atualização da classificação dos jogadores e a interface visual do valor para o jogador local.

4.6 IMPLEMENTAÇÃO DOS COLETÁVEIS

O comportamento dos coletáveis é bastante simples. Ao iniciar as partidas, os coletáveis são instanciados localmente em posições iniciais aleatórias calculadas para cada um e realiza-se o *spawn* na rede. O código 9 realiza o *spawn* inicial.

No instante que uma colisão entre um jogador e o coletável acontece, ele deveria ser destruído e um novo *prefab* deveria ser criado e usado para se realizar uma nova instanciação local e na rede via *spawn*. Porém, isso é bastante custoso.

Uma solução mais interessante utiliza o fato de o número de coletáveis ser fixo durante a partida. Quando um coletável deve sumir e um novo ser criado, podemos simplesmente

chamar o método *despawn* para este coletável a partir do servidor, removendo-o da rede, gerar uma nova posição aleatória para ele dentro do espaço de jogo e fazer um novo *spawn* do objeto na nova posição, sem necessidade de instanciações locais custosas nos clientes, como pode ser visto no código 10.

4.7 IMPLEMENTAÇÃO DO DESAFIO

Esta é a parte mais desafiadora da implementação. O desafio da forma que foi projetado exige medições de tempo, diferentes visualizações para os clientes envolvidos, além de aguardo de resposta do servidor de forma mais explícita que a movimentação.

Na subseção 4.7.1 será descrita como foi tratada a colisão entre os jogadores para a implementação do desafio. Na subseção 4.7.2 será feita uma análise da criação e inicialização. A subseção 4.7.3 trata a parte da execução. A subseção 4.7.4 se dedica a explorar o componente *Challenge* do objeto. Finalmente, a subseção 4.7.5 examina o término do desafio.

4.7.1 Colisão Entre Jogadores

O desafio ocorre quando dois jogadores colidem entre si, conforme explicado na subseção 3.2.5. Quando isso ocorre, o método *OnCollisionEnter2D* da *Unity* é executado. Este método pode ser observado no código 11.

As variáveis *challengeOpponent*, que é do tipo *ulong*, e *isInChallenge*, que é do tipo booleano, são variáveis de rede. O valor dessas variáveis é alterado na colisão para o *id* do jogador adversário e verdadeiro no caso do booleano. Nesse momento, conforme discutido na subseção 4.4.1, uma *callback* é executada para cada uma delas.

A *callback* executada na alteração da variável *isInChallenge* tem duas funções principais. A primeira é na mudança de valor de falso para verdadeiro e a segunda no sentido oposto. Na subseção 4.7.2 trata-se do primeiro caso. O segundo caso é discutido na subseção 4.7.5.

4.7.2 Criação e Inicialização do Desafio

Quando o jogador entra em um desafio, ou seja, o valor de *isInChallenge* é modificado para verdadeiro, além de desligar os colisores, ocorre a criação do desafio através do método *CreateAndSpawnChallenge*, que faz parte do *script ChallengeNetwork*, cujo papel é controlar a sincronização do desafio na rede.

Importante notar que este *script* é atrelado ao objeto do jogador e a criação só deve ocorrer no servidor. Isto significa que o código que trata a colisão é executado uma vez para cada jogador envolvido no desafio em cada um dos nós, mas apenas no servidor ele chama o método de criação.

Isso também significa que é necessária uma checagem na execução se esse desafio entre estes dois jogadores já não foi criado, dado que o código é executado duas vezes no servidor, uma para cada jogador.

Tendo sido feita essas duas checagens, o desafio então é criado, armazenado e iniciado via uma corrotina de nome *StartChallenge*, que começa a dinâmica em si. Uma corrotina na linguagem C# nada mais é que um método executado assincronamente.

Por fim, uma segunda corrotina é executada para realizar os efeitos visuais do jogador. O código 12 mostra esta sequência.

O método *CreateAndSpawnChallenge* recebe os jogadores e o tipo de desafio. É feita então a instanciação local do *prefab* do desafio e também o *spawn* na rede, além do armazenamento de uma referência em uma lista simples, tudo de forma bastante direta. Esta parte do desenvolvimento pode ser observada no código 13.

4.7.3 Execução do Desafio

O começo da execução do desafio ocorre com o acionamento da corrotina *StartChallenge*, executada pelo método invocado na colisão. Ele é quem trata da dinâmica principal do jogo.

Este método tem 3 invocações de *WaitForSeconds*, que suspende a execução da corrotina por um determinado tempo em segundos. É dessa forma que se criam os intervalos de tempo discutidos na subseção 3.2.5. Entre estes intervalos também ocorrem as mudanças nos objetos dos jogadores, como adicionar e remover vidas e mudança de variáveis de rede. Além disso, ele recebe o objeto do desafio que foi criado anteriormente e acessa o componente *Challenge* do objeto do desafio, que é o *script* que controla o seu comportamento local.

O código 14 mostra o funcionamento desta execução do desafio. No intervalo entre esses instantes de suspensão, ocorrem diversas chamadas a métodos deste componente de comportamento local. É através dele que se decide o vencedor e se mostram os resultados para os clientes.

4.7.4 O Componente Challenge

Tendo entendido o comportamento do componente *ChallengeNetwork* e como ele controla a execução do desafio e seu tempo de vida conforme o seu tipo, agora é necessário entender o funcionamento do componente *Challenge*, acoplado ao objeto do desafio. Conforme citado anteriormente, ele tem por função controlar o comportamento local do desafio e o que deve acontecer em cada intervalo de tempo.

Existem partes deste *script* que lidam com o desafio quando seu tipo é de pergunta e resposta, outras que lidam com o desafio do tipo pressionamento de botão e a maioria é executada em ambos os casos. Primeiro serão mostradas as partes compartilhadas na

subseção 4.7.4.1 para depois entrar nas individuais, sendo a subseção 4.7.4.2 referente ao desafio de pressionamento de botão e a subseção 4.7.4.3 referente ao de pergunta e resposta.

4.7.4.1 Partes de Código Compartilhadas

Aqui se faz necessário deixar claro que o desafio executa todo do lado do servidor e por isso o desafio é um objeto de rede e não local. Os clientes se comunicam via *RPCs*. As poucas mudanças que ocorrem localmente são sempre relacionadas a visual que não precisa de sincronização ou que é exclusivo de um determinado cliente, como seria, por exemplo, o que está sendo digitado e ainda não foi enviado. Por conta disso, faz-se necessário um método simples que verifique se o cliente local está ou não envolvido no desafio que é criado, pois ele é criado para todos os clientes, mas deve ser mostrado apenas para os envolvidos. O método *LocalClientInChallenge* mostrado no código 15 tem esta função.

As primeiras questões importantes no código do componente *Challenge* são alguns métodos simples relacionados ao tempo de vida do desafio. O primeiro deles é uma *ServerRPC* que calcula o valor aleatório que compõe o tempo de preparação do desafio, explicado na subseção 3.2.5. Esse método é executado no *spawn* do objeto do desafio.

Em seguida, existem alguns métodos com a função de entender em que intervalo de tempo o desafio se encontra, sendo eles *PassedTime*, *IsInPreparation*, *IsInChallenge*. Através deles e de variáveis que guardam o tempo de início do desafio, o tempo até o começo do tempo útil e o tempo até o anúncio do vencedor, criam-se as janelas de tempo diferentes onde as ações são executadas. O código 16 mostra estes métodos e variáveis relevantes.

No *framework Netcode for Gameobjects*, o tempo é fundamental e possui natureza relativa, sendo gerenciado por um sistema de *ticks* fixos. Variáveis de rede são atualizadas e enviadas aos demais dispositivos da rede no instante do próximo *tick*. Para realizar cálculos de tempo, utiliza-se a classe *NetworkTime*, que oferece acesso a dois métodos principais: *LocalTime* e *ServerTime*. Cada cliente tem um *ServerTime* e um *LocalTime* atrelado a si.

LocalTime representa o tempo local do cliente, que pode estar ligeiramente à frente do tempo do servidor. Se uma *RPC* for enviada neste tempo local por um cliente, espera-se que ela seja recebida no *ServerTime* do servidor, ou seja, a diferença de tempo é igual ao *RTT*.

ServerTime representa o tempo do servidor nos clientes. Este tempo é o que o cliente acredita que o servidor está, estando sempre atrasado em relação ao servidor real. Se uma RPC for enviada neste tempo pelo servidor para os clientes, espera-se que ela seja recebida no *ServerTime* dos clientes.

Em resumo, deve-se usar *LocalTime* para casos de autoridade do cliente e *ServerTime* para casos de autoridade do servidor. Ou seja, nos nossos cálculos de tempo estaremos

sempre usando *NetworkTime.ServerTime*.

Um último tópico relacionado ao tempo é que essas checagens são todas feitas no método *Update* da *Unity*. Nele se modifica o que o cliente está vendo na tela conforme o intervalo que se encontra. Coisas como o tempo restante de desafio e a imagem verde ou vermelha de preparação são atualizados na tela pelo método *GetTimeoutText* e por ações no próprio método *Update*. O código 17 mostra alguns exemplos desse controle visual.

Quando o jogador pressiona botões, é necessário algum tipo de *feedback* para que o jogo pareça mais fluido e responsivo. Esse *feedback* pode ser feito de algumas formas diferentes, mas, neste trabalho, todos eles são feitos de forma visual. Para isso, também existem alguns métodos no componente *Challenge* com esse intuito de gerar respostas positivas ou negativas ao jogador mediante mudanças rápidas de cores na tela, verde ou vermelho, respectivamente. Estes métodos se encontram no código 18.

A maneira como os clientes enviam *input* durante o desafio é similar à movimentação do jogador discutida na subseção 4.5.1, com a diferença de a checagem ser feita dentro do componente *Challenge* via uma *RPC*.

Se faz necessário o uso de uma estrutura de dados do tipo dicionário para armazenar o tempo de conclusão de cada um dos jogadores. Essa é a variável *clientFinishTimestamps* do tipo *Dictionary<ulong, double>*, que associa o id do jogador com o seu tempo de finalização .

Com este dicionário criado na inicialização do desafio, usa-se o método *StoreFinishTimestampServerRpc* para guardar este par quando o cliente finaliza seu *input*. Essa ação será realizada dentro do método *Update* do componente *Challenge*, em momentos diferentes dependendo do tipo de desafio que está acontecendo. O conteúdo é bastante simples. Ele tenta guardar o par caso não haja um presente para aquele *id*, evitando duplicatas ou erros. O código 19 demostra este processo.

Um adendo muito importante sobre os *timestamps* recebidos pelo servidor é que existe uma questão problemática neste processo. Conforme já citado diversas vezes, este trabalho tem por intuito fazer o servidor ser o mais autoritário possível e remover todas as atribuições possíveis dos clientes. Porém, esse se mostrou um trabalho bastante complexo ao tentar calcular tempo de reação de um nó da rede. Idealmente se trabalharia com uma *RPC* para o servidor, que indicaria o fim do desafio, e o servidor definiria este instante de tempo. No entanto, isso se torna muito complexo pela imprevisibilidade da rede e da distância entre os nós até o servidor. Não seria possível definir a corretude da ordem de término a partir da ordem de recebimento.

Por estes motivos, no intuito de simplificar esta questão, foi decidido que os clientes seriam responsáveis por definir e enviar seus *timestamps* ao fim do desafio, removendo assim o problema do tempo de comunicação variável e diferente entre os nós. Por conta dessa decisão, abre-se um ponto de fragilidade contra ataques e códigos maliciosos do lado do cliente, e é importante que isso fique claro. Porém, no contexto deste trabalho, isso

será ignorado e sempre se irá supor que o valor recebido pelo servidor não sofreu qualquer tipo de alteração indevida.

Ao término do tempo útil de desafio, é preciso decidir se algum dos jogadores venceu ou se houve um empate. Este resultado é obtido pelo método *DecideWinner*, que retorna o *id* do jogador vencedor ou zero em caso de empate.Para que isso seja possível, olhamos para a estrutura de dados com os tempos de finalização. A partir deste dicionário, o método *DecideWinner* faz essa checagem conforme pode ser visto no código 20. Perceba que no caso de não ser encontrado o *id* de um jogador, pressupõe-se que ele não finalizou o desafio, logo, é atribuído valor *ulong.MaxValue* ao seu tempo de conclusão, possibilitando as comparações de forma correta.

Tendo a informação de quem venceu o desafio, é preciso mostrar aos dois jogadores. Esta etapa também mostra uma tela em comum para ambos, sem diferenciação do tipo de desafio, exceto por um detalhe no caso do desafio de pergunta e resposta que será explicado na subseção 4.7.4.3.

A exibição das informações de final do desafio é feita via *RPCs*. A primeira delas é uma *ServerRpc* invocada no próprio servidor de nome *DisplayResultsServerRPC*, executada na corotina *StartChallenge* explicada na subseção 4.7.3 . Seu papel é pedir ao servidor o resultado do desafio. O servidor então invoca uma segunda *RPC* chamada de *DisplayResultsClientRPC*, executada nos dois clientes envolvidos.

Essa *RPC* gera um *feedback* visual aos jogadores, gerando um fundo verde para o vencedor e vermelho ao perdedor ou, em caso de empate, para os dois jogadores. Depois, calcula o tempo de resposta de cada um usando os *timestamps* recebidos via parâmetros com o método *GetClientResponseTime*, calcula a diferença entre os tempos e gera o texto que diz quem foi o vencedor e qual a diferença entre os tempos de resposta. O código 21 mostra o funcionamento dessas duas funções.

Com isso, descrevemos todo o código executado pelo desafio em qualquer um dos casos. A diferença entre os dois tipos de desafio ocorre no método *Update* do componente *Challenge*. Ou seja, durante a execução do jogo, dependendo do tipo de desafio, diferentes ações são chamadas dentro desta função via cláusulas *if* simples. Nas próximas subseções descreveremos os métodos específicos para cada tipo de desafio.

4.7.4.2 Desafio de Pressionamento de Botão

Neste caso, o código simplesmente faz uma checagem se o jogador é o cliente local (lembrando que esse código executa também no outro cliente), se ele apertou espaço quando se está no tempo útil de desafio e se ele tem pressionamentos restantes. Essa mecânica de pressionamentos foi adicionada como um recurso de *game design* que impede o jogador de pressionar a barra de espaço consecutivamente antes do tempo útil de desafio, garantindo assim que seja de fato medido um tempo de resposta à mudança visual na tela.

Caso o jogador gaste todos os seus pressionamentos restantes, ele perde o desafio automaticamente e aguarda o adversário finalizar o dele ou não, para saber se houve um empate em que ambos perdem, ou se houve a vitória do outro jogador. Esse valor de pressionamentos é configurável no arquivo de constantes, discutido na subseção 4.3.1.

Como esse contador de pressionamentos foi implementado também é via uma *ClientRPC*. O servidor notifica o cliente específico para que ele atualize seu contador local e a interface visual para o jogador, de forma bem direta. Este método pode ser visto no código 22.

Caso ocorra pressionamento no momento correto, o cliente recebe *feedback* visual positivo. Caso contrário, negativo. Então, ele aguarda o fim do tempo útil para o anúncio do resultado. O código 23 mostra esta parte completa.

4.7.4.3 Desafio de Pergunta e Resposta

No caso de ser executado um desafio de pergunta e resposta, algumas coisas diferentes acontecem. Em primeiro lugar, é preciso falar do componente *QuestionManager*, que está acoplado ao objeto homônimo. Quando este componente é instanciado na rede, o método *LoadQuestions* carrega em memória todas as questões incluídas no arquivo *Questions.json*. Isso não é um grande problema dado que não há perspectiva de arquivos com número muito grande de questões. A classe *Question* apenas tem dois atributos, *string query* que guarda a pergunta e *string* [] answers que guarda todas as possíveis respostas.

O método *OnNetworkSpawn* do componente *Challenge* executa um pequeno trecho de código que tem a função de ativar o campo de digitação para a resposta e fazer o servidor executar a *ServerRPC CreateRandomQuestionServerRpc*, que faz o servidor gerar uma nova questão que será a pergunta e resposta deste desafio específico. Ou seja, cada desafio ao ser instanciado na rede define qual será sua questão dada a lista de questões carregada anteriormente. Essa *RPC* repassa também ao componente *QuestionManager* o dever de selecionar aleatoriamente um elemento da lista que já foi carregada.

O código 24 mostra o componente *QuestionManager* e o código 25 mostra a execução do desafio de questões dentro do método *Update* do componente *Challenge*.

4.7.5 Terminando um Desafio

O término do desafio acontece ao fim da corrotina *StartChallenge* no componente *ChallengeNetwork*. Após sua execução, o desafio terá acontecido, os resultados terão sido mostrados aos jogadores, as vidas concedidas ou retiradas e a variável de rede *isInChallenge* de cada um dos componentes *PlayerNetwork* modificadas para falso.

Neste momento, a *callback* da variável de rede é executada, fazendo a chamada a uma corrotina que dá a velocidade aumentada e invulnerabilidade momentânea aos dois jogadores para poderem se reposicionar no espaço de jogo e também ordenando que o

servidor faça o *despawn* do objeto *Challenge* e a destruição do objeto localmente. O código 26 mostra esta passagem.

4.8 CONSIDERAÇÕES FINAIS

Com isso, todo o desenvolvimento foi discutido, com pouquíssimos trechos de código tendo sido omitidos, de forma que ficasse bem claro todo o trabalho feito, as motivações por trás das decisões tomadas e alguns dos principais problemas enfrentados durante o processo. O capítulo 5 vai abordar alguns dos testes realizados do resultado que foi alcançado.

5 AVALIAÇÃO E RESULTADOS

Esse capítulo tem por objetivo mostrar os resultados obtidos ao fim do desenvolvimento deste trabalho e avaliar algumas questões relacionadas ao produto final. Na seção 5.1 serão apresentadas as diferenças e funções das *builds* cliente e servidor. Na seção 5.2 é discutido como se realiza as mudanças no arquivo JSON de questões. Na seção 5.3 serão abordados os resultados do desenvolvimento. A seção 5.4 explica de que foi feito o teste remoto. Na seção 5.5 são exploradas as impressões dos jogadores que participaram do teste. Por fim, a seção 5.6 traz algumas considerações finais relevantes.

5.1 BUILDS CLIENTE E SERVIDOR

O objeto *NetworkManager* descrito na subseção 4.3.4.1 tem um componente onde se insere o endereço para a conexão. Quando se está desenvolvendo e fazendo testes em aplicações deste tipo localmente, não é necessário se preocupar com isso, pois o endereço local é utilizado tanto para a aplicação cliente quanto para o servidor.

No entanto, quando se trata de uma aplicação cliente que vai ser executada remotamente, precisamos modificar o endereço para o da máquina que vai hospedar o servidor. Para que isso seja feita de forma mais simples, geram-se dois *builds* do jogo, sendo o primeiro com o endereço do servidor, o qual é a aplicação cliente, e um segundo de endereço *localhost*, que é a aplicação servidor. Desta forma, obtêm-se dois programas separados, que deve cada um ser utilizado para uma função.

Idealmente, este processo de *build* poderia se tornar algo automatizado para acelerar o desenvolvimento e possíveis modificações que sejam feitas. Também seria interessante futuramente que o cliente e o servidor não precisassem interagir com botões para escolher sua função dentro de jogo via o *NetworkUIManager* descrito na subseção 4.3.4.2. Ao invés disso, cada *build* já teria seu papel e a existência deste componente seria desnecessária.

O processo de *build* para a plataforma Windows gera uma pasta que contém os arquivos do jogo e um executável para iniciar. A Unity conta com suporte de *builds* para outras plataformas, como Linux, servidor dedicado, entre outros. No entanto, neste trabalho optou-se por suportar apenas o sistema operacional Windows e os outros não foram testados.

5.2 MODIFICANDO O ARQUIVO JSON

Um problema que surgiu após o fim do desenvolvimento foi com relação à modificação do arquivo JSON, por falta de conhecimento sobre a plataforma *Unity* e o seu processo

de *build*. Ao utilizar a pasta *Resources*, o processo de *build* faz um *packing* desses *assets* e não deixa que eles sejam visíveis ou modificáveis na pasta.

Por conta disso, seria impossível fazer as modificações no arquivo e executar o jogo em seguida com as modificações feitas, ele só ficaria editável anteriormente ao processo de *build*, o que acabaria com o ponto forte do *framework*.

Para resolver este problema, fez-se uso então da pasta *StreamingAssets* para armazenar o JSON. Esta pasta, sim, mantém os arquivos preservados posteriormente ao processo de *build*, permitindo então o comportamento desejado. Desta forma fica bastante simples modificar o arquivo do caminho "/Data/StreamingAssets/Questions/Questions.json". Existia também a possibilidade de se usar a pasta destinada a arquivos persistentes, que varia de local dependendo da plataforma que se está sendo utilizada, mas que não fica dentro da pasta do jogo gerada pelo processo de *build*. Por esse motivo, optou-se por facilitar o acesso via *StreamingAssets*. Uma limitação importante que isso causa é a impossibilidade de escrita em tempo de execução, mas continua cobrindo a proposta inicial do trabalho.

5.3 RESULTADOS DO DESENVOLVIMENTO

O jogo final ficou consoante com o que foi proposto. A maioria das funcionalidades que foram especificadas no decorrer do projeto foram implementadas com sucesso, com alguns problemas pontuais detectados durante os testes, e que serão elencados no decorrer deste capítulo.

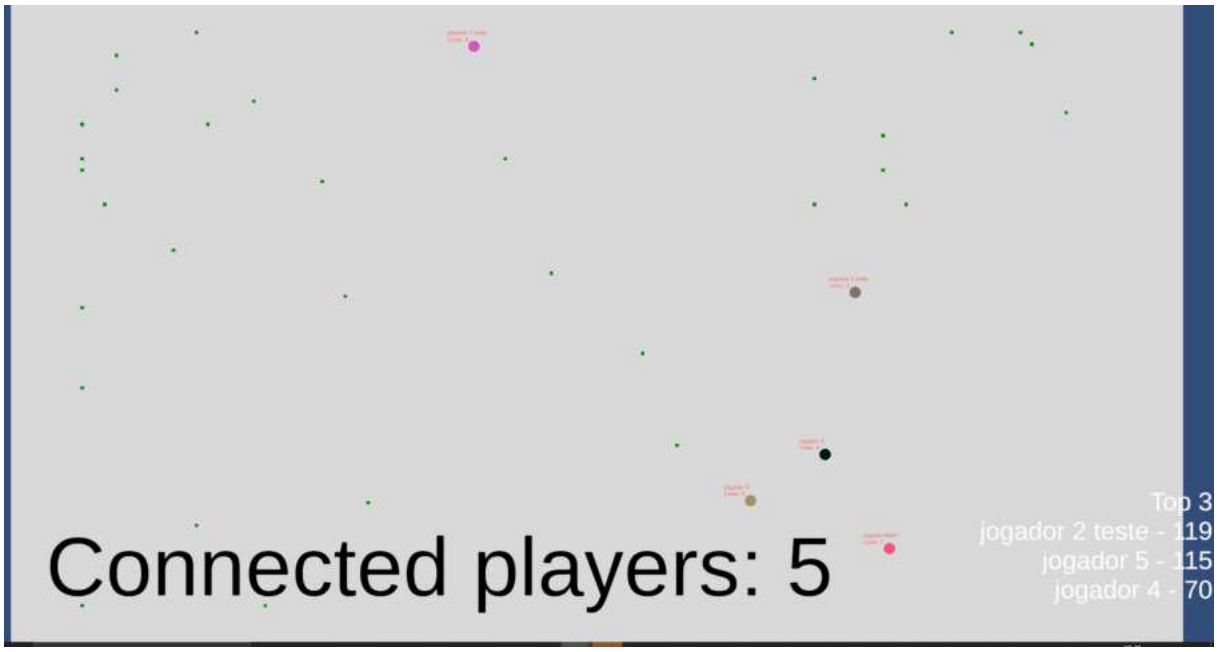
A figura 16 mostra a tela inicial do jogo, com a área para o apelido do jogador, botões para conexão como cliente ou servidor e a escolha do tipo de desafio que se deseja. Importante salientar que a conexão como servidor ignora o campo do apelido, enquanto a conexão como cliente ignora a escolha de desafio. Conforme discutido na seção 5.1, idealmente os *builds* diferentes removeriam este tipo de problema e seriam uma boa melhoria futura para evitar qualquer confusão que possa ocorrer.

As figuras 17 e 18 mostram, respectivamente, a visão de espectador do servidor e a visão durante o jogo do cliente de apelido "jogador 4". Importante ressaltar que foram feitas algumas modificações de posicionamento da interface visual de pontuação, tendo sido movido para o canto inferior direito o *ranking* dos jogadores e removida do canto esquerdo superior o apelido e pontuação máxima obtida na partida que era mantida mesmo após desconexão daquele jogador. Este canto superior esquerdo se tornou um indicativo da pontuação do cliente local.

A figura 19 mostra a visão de um jogador de outros quatro jogadores que estão participando de dois desafios separados. O jogador local, de apelido "Player 4", observa que existe um desafio acontecendo entre os jogadores de apelidos "Player 5"e "Player 7"e outro entre os jogadores de apelidos "Player 1"e "Player 6". Uma melhoria que deveria ser feita futuramente seria não permitir a sobreposição de *labels* que confunde o entendimento.

Figura 16 – Tela inicial

Figura 17 – Visão de espectador do servidor



O apêndice A mostra os estados possíveis do desafio de pressionamento de botão, enquanto o apêndice B mostra os estados possíveis do desafio de pergunta e resposta.

5.4 TESTES REMOTOS

Para que um teste remoto fosse realizado, optou-se por utilizar-se a plataforma Azure no seu plano gratuito para executar na nuvem uma máquina virtual Windows padrão

Figura 18 – Visão do cliente em uma partida

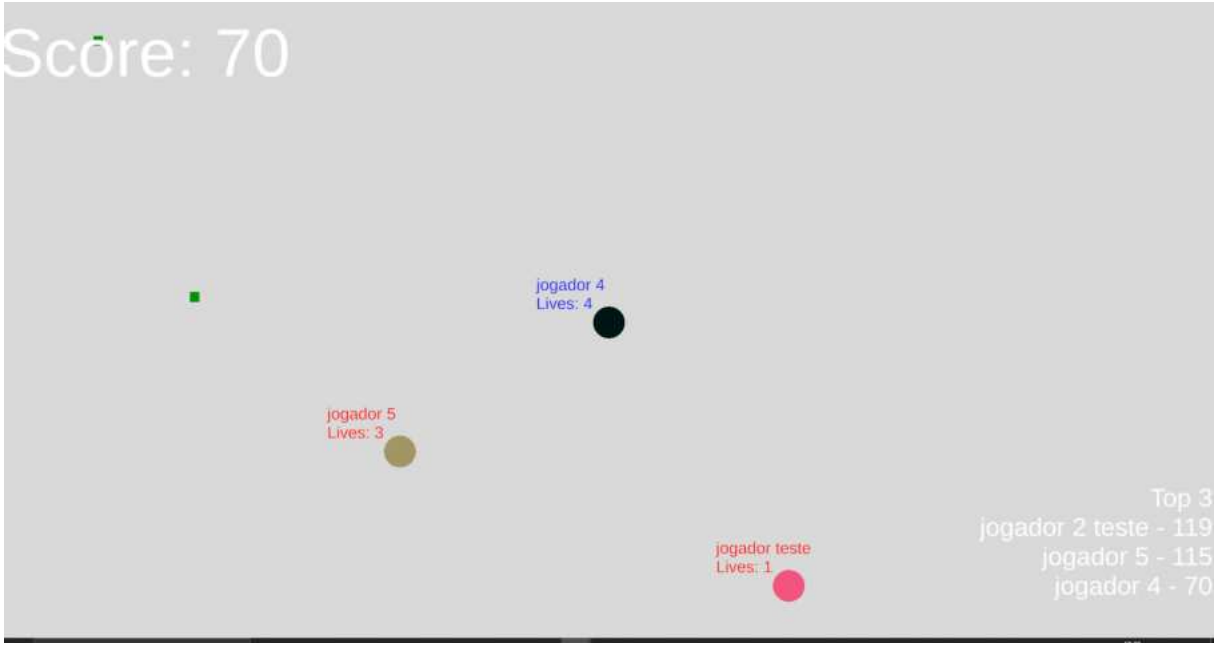
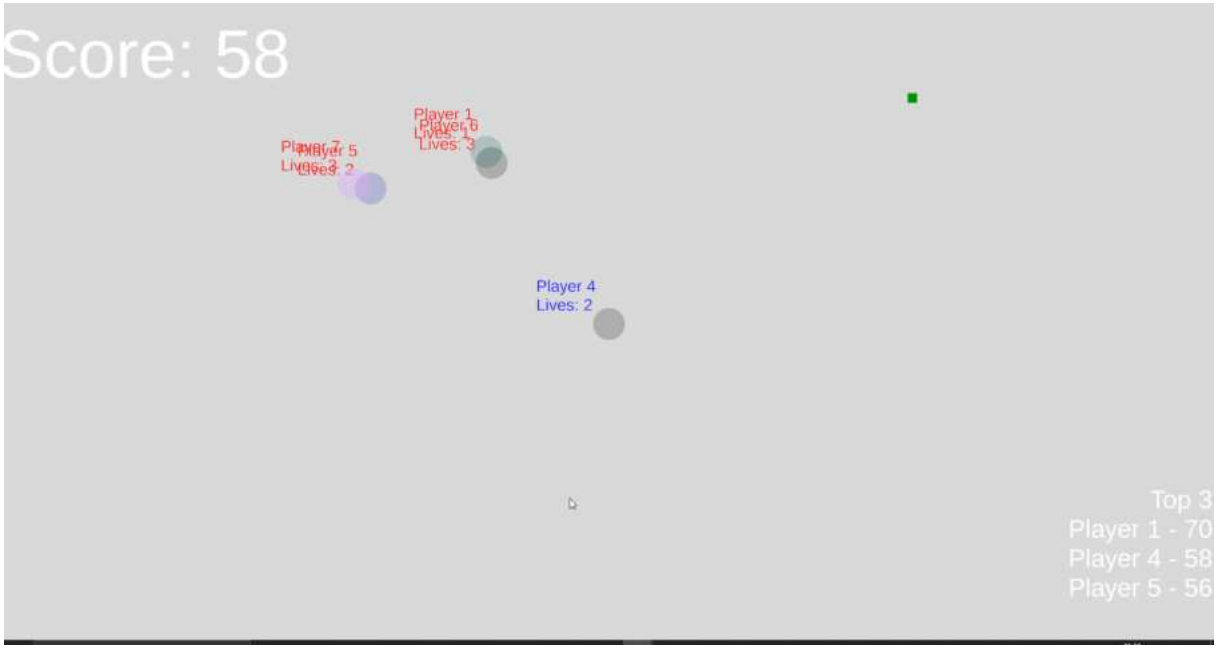


Figura 19 – Visão do jogador de outros dois desafios



onde o *build* de servidor pudesse ser executado e controlado. No entanto, problemas ocorreram logo nas primeiras tentativas. Os poucos recursos disponíveis desta máquina impossibilitaram a realização dos testes.

Passou-se então a uma máquina paga utilizando o crédito gratuito de duzentos dólares ou trinta dias, o que acontecer primeiro, oferecido a novas contas. Essa máquina com 2 vcpus e 8GiB de memória foi então capaz de executar a aplicação servidor de forma mais

satisfatória.

Realizou-se então um teste com até quatro jogadores simultâneos conectados ao servidor testando as funções do jogo. Durante este tempo, alguns comportamentos não esperados chamaram a atenção.

O primeiro deles foi relacionado a velocidade de movimento dos jogadores. Notou-se que computadores mais potentes faziam o personagem deste cliente se mover notavelmente mais rápido que os personagens dos clientes com menor poder computacional. O motivo está no código que tinha sido utilizado para a movimentação do objeto controlado pelo jogador. Anteriormente, o código da movimentação era como o do código 27.

Observando o código 5 que descreveu a movimentação do jogador, nota-se que a sutil diferença é a presença do multiplicador *Time.deltaTime* dentro da *RPC*. Este valor é o tempo em segundos que levou para o último quadro ser executado dividido pela taxa de quadros da máquina. Este valor vai variar de máquina para máquina e ele precisa ser utilizado para garantir que os objetos se movam a uma velocidade constante independente da sua taxa de quadros.

O problema está no fato de esta compensação estar sendo feita utilizando o *deltaTime* do servidor dentro da *ServerRPC*, ou seja, a cada quadro do cliente. Ao mover a operação para o método *Update* que executa no cliente, torna-se uma movimentação de distância fixa em um intervalo de tempo em qualquer que seja a taxa de quadros e todos os jogadores passam a se mover com a mesma velocidade, que era o pretendido inicialmente.

Além disso, o tamanho da janela da aplicação servidor na máquina virtual influenciava a velocidade de todos os jogadores de forma diretamente proporcional. Acredita-se que isto seja um impacto da quantidade pequena de recursos disponíveis na máquina onde o servidor estava executando. Possivelmente em uma máquina com uma configuração mais sofisticada se alcançasse uma estabilidade quanto a isto.

5.5 PERCEPÇÃO DOS JOGADORES

Não houve a possibilidade de testes com muitos clientes simultâneos por questões logísticas e pelo tempo de uso gratuito da máquina virtual da Azure, o que também dificultou solicitar um *feedback* formal dos jogadores, como um questionário, por exemplo. No entanto, nos testes que foram feitos com algumas pessoas, foi explicitado por eles que o jogo estava divertido e parecia bastante responsivo. Também não houve reclamações quanto ao resultado de desafios. Aqueles que saíam derrotados sentiam que tinham de fato perdido no tempo de reação para o seu oponente.

Com relação às perguntas e respostas, geraram-se algumas dúvidas a respeito de maiúsculas e minúsculas, assunto que já foi discutido na seção 3.5. Seria um ponto de melhoria bastante relevante a ser desenvolvido futuramente.

Também foi sugerido que se pudessem adicionar perguntas e respostas de dentro da

aplicação e não apenas diretamente no arquivo JSON, mas esta já seria uma mudança para ser cogitada mais a frente.

Com relação às explicações, foi sugerido que houvesse um tutorial mais detalhado e também a possibilidade de mudança de idioma. O jogo foi todo feito utilizando-se o inglês como padrão. Apesar de as perguntas e respostas serem configuráveis no *framework* para a linguagem que se desejar, os textos e explicações não foram feitos dessa forma.

5.6 CONSIDERAÇÕES FINAIS

O final do desenvolvimento do jogo se demonstrou bastante satisfatório. Infelizmente não houve a possibilidade de testes com um número maior de jogadores e uma coleta de opiniões mais robusta, o que com certeza geraria outras questões para serem melhoradas. Além disso, teria sido bom receber *feedbacks* também de educadores que fizessem uso da ferramenta com alunos para entender melhor a efetividade do *framework* em casos reais de aprendizado. Apesar disso, acredita-se que os objetivos foram atingidos ao fim do processo.

6 CONCLUSÃO

Neste trabalho, foi desenvolvido um jogo multijogador distribuído em tempo real de arquitetura cliente-servidor com um servidor o mais autoritário possível, que explora tomada de tempo de resposta dos jogadores ao mesmo tempo que provê um *framework* de customização de perguntas e respostas com intuito pedagógico. Os resultados foram satisfatórios e condizentes com o que se esperava, um jogo dinâmico e que proporciona flexibilidade ao educador que fizer uso do mesmo, permitindo cobrir todo tipo de contexto.

Foi detalhado o processo de desenvolvimento na *game engine Unity* e a codificação dos comportamentos dos objetos. O processo de sincronização dos objetos e como os *scripts* são executados em cada nó da rede foram abordados, demonstrando alguns dos desafios enfrentados. Também comentou-se sobre os *builds* diferentes para servidor e cliente e como poderiam ter sido melhor implementados separadamente.

As principais decisões referentes ao *design* do jogo estão relacionadas a como calcular os tempos de resposta e a opção por não considerar questões de segurança para que isto fosse possível. Outras decisões buscaram evitar complexidades desnecessárias para uma versão inicial.

A avaliação dos jogadores que participaram de um teste remoto foi de que a dinâmica do jogo ficou responsiva e não houve problemas com relação à sensação de tempo de resposta.

6.1 TRABALHOS FUTUROS

Para um momento posterior, seria muito interessante que fossem feitas melhorias de interface visual e de usabilidade para os usuários sentirem uma melhora na usabilidade. Também seria bem-vindo que educadores fossem consultados e pudessem testar o jogo como ferramenta pedagógica para gerar mais *feedbacks* relevantes.

Com relação ao *design* do jogo, seria importante que uma versão futura tivesse o suporte a maiúsculas e minúsculas, adicionando ao arquivo JSON um campo booleano que diga se deve ser suportado ou não. Assim, não seria quebrada a compatibilidade com arquivos antigos e a modificação poderia ser feita no código.

Seria relevante que fossem adicionados efeitos sonoros nos *feedbacks* dos desafios, mas possivelmente com opção de configuração para ligar e desligar. Isso permitiria o uso do jogo com pessoas autistas que sejam afetadas por sons. Também pensando em acessibilidade, as cores dos personagens não são selecionáveis e são atribuídas randomicamente. Pessoas com daltonismo de diversos tipos podem ser afetadas dependendo da escolha de cores. Uma opção de selecionar as cores ou de permitir apenas certos espectros para cada tipo de daltonismo poderia ser adicionada.

Uma ideia com relação a configurações seria também criar uma melhor separação entre os *builds* cliente e servidor e incrementar o lado servidor com uma tela de configuração da partida onde todos os parâmetros constantes pudessem ser ajustados para criar uma customização maior e cobrir mais casos de uso.

REFERÊNCIAS

CASTRO, D. F. de; TREDEZINI, A. L. de M. A importância do jogo/lúdico no processo de ensino-aprendizagem. **Perquirere**, v. 1, n. 11, p. 166–181, 2014.

EMMENS, C. Beep. bong. replace shh! in the library when video games are added to the collection. **Collection Building**, v. 6, n. 1, p. 15–18, 1984.

FERNANDES, L. D. et al. Jogos no computador e a formação de recursos humanos na indústria. **VI Simpósio Brasileiro de Informática Educação. Anais**, 1995.

FLETCHER, J.; TOBIAS, S. Using computer games and simulations for instruction: A research review. Orlando, FL, v. 108, 2 2006. Paper presented at the Society for Applied Learning Technology Meeting.

HAJESMAEEL-GOHARI, S. et al. Digital games for rehabilitation of speech disorders: A scoping review. **Health Science Reports**, Wiley Online Library, v. 6, n. 6, p. e1308, 2023.

HSIAO, H.-C. A brief review of digital games and learning. In: IEEE. **2007 First IEEE International Workshop on Digital Game and Intelligent Toy Enhanced Learning (DIGITEL'07)**. Los Alamitos, CA, USA, 2007. p. 124–129.

JAYAKANTHAN, R. Application of computer games in the field of education. **The electronic library**, MCB UP Ltd, v. 20, n. 2, p. 98–102, 2002.

KAZMI, S. et al. Interactive digital serious games for the assessment, rehabilitation, and prediction of dementia. **International Journal of Computer Games Technology**, Wiley Online Library, v. 2014, n. 1, p. 701565, 2014.

KUSUMA, G. P. et al. Analysis of gamification models in education using mda framework. **Procedia Computer Science**, Elsevier, v. 135, p. 385–392, 2018.

LOPES, R. P. et al. Digital technologies for innovative mental health rehabilitation. **Electronics**, MDPI, v. 10, n. 18, p. 2260, 2021.

MENDES, C. L. **Jogos Eletrônicos: Diversão, Poder e Subjetivação**. 1. ed. Rio de Janeiro: Papirus, 2006. 156 p.

MUNDAY, P. Duolingo. gamified learning through translation. **Journal of Spanish Language Teaching**, Taylor & Francis, v. 4, n. 2, p. 194–198, 2017.

PRENSKY, M. Computer games and learning: Digital game-based learning. **Handbook of computer game studies**, v. 18, n. 2005, p. 97–122, 2005.

RODRIGUES, L. C.; MUSTARO, P. N. Social network analysis of virtual communities in online games. v. 2007, p. 237–244, 2007.

TYLOR, E. B. The history of games. **Fortnightly**, Chapman and Hall., v. 25, n. 149, p. 735–747, 1879.

WHITTON, N. **Digital games and learning: Research and theory**. 1. ed. New York: Routledge, 2014. 225 p.

WIJMAN, T. **Global Games Market to Generate $175.8 Billion in 2021; Despite a Slight Decline, the Market Is on Track to Surpass $200 Billion in 2023**. 2021. Disponível em: https://newzoo.com/resources/blog/global-games-market-to-generate-175-8-billion-in-2021-despite-a-slight-decline-the\-market-is-on-track-to-surpass-200-billion-in-2023.

APÊNDICE A – TRECHOS DE CÓDIGOS.

Código 1 – Arquivo GameConstants.cs

```
public static class GameConstants {
    // Camera
    public const float CameraHeight = -10;
    public const float BaseObserverCameraMovespeed = 50f;
    // Game space
    public const int MinHorizontal = -45;
    public const int MaxHorizontal = 45;
    public const int MinVertical = -45;
    public const int MaxVertical = 45;
    // Collectibles
    public const int CollectibleValue = 1;
    public const int NumberOfCollectibles = 80;
    // Players
    public const int StartingLives = 3;
    public const int MaxLives = 5;
    public const float BaseMovespeed = 10f;
    // Game
    public const bool MaximumGamePointsCapped = false;
    public const int MaximumGamePointsCap = 5000;
    public const int MaximumPlayersToDisplayScore = 3;
    // Challenges
    public enum ChallengeType {
        KeyboardButtonPress,
        QuestionChallenge,
        Random
    }
    public const int ChallengeValue = 50;
    public const float ChallengeStartDelayInSeconds = 3f;
    public const float PostChallengeInvincibilityTimeInSeconds = 3f;
    public const float PostChallengeSpeedMultiplier = 2f;
    public const float PlayerAlphaWhileInChallenge = 0.4f;
    public const float ChallengeWinnerTime = 5f;
    public const int ChallengeTimeoutLimitInSeconds = 8;
    public static readonly Color ScreenFlashGreen = new(0f, 255f, 0f,
       0.4f);
    public static readonly Color ScreenFlashRed = new(255f, 0f, 0f, 0.4f
       );
    public const float ColorFlashTime = 0.1f;
    public const float RandomTimeSugarWindow = 1.5f;
    public const int MaxPressesAllowed = 1;
}
```

Código 2 – Arquivo NetworkManagerUI.cs

```
public class NetworkManagerUi : MonoBehaviour {
    ...
    private void Awake() {
        _playerNicknameObject = GameObject.Find("NicknameInput");
        _playerNicknameInputField = _playerNicknameObject.GetComponent<
            TMP_InputField>();
        _playerNicknameInputField.Select();
        _playerNicknameInputField.ActivateInputField();

        serverButton.onClick.AddListener(() => {
            NetworkManager.Singleton.StartServer();
            serverButton.image.color = Color.green;
            Destroy(_playerNicknameObject);
            GameManager.DestroyToggleGroup();
            ActivateDisconnectButton();
        });

        clientButton.onClick.AddListener(() => {
            NetworkManager.Singleton.OnConnectionEvent += (_, data) => {
                if (data.EventType != ConnectionEvent.ClientConnected)
                    return;
                var playerNickname = _playerNicknameObject.GetComponent<
                    TMP_InputField>().text;
                NetworkManager.Singleton.LocalClient.PlayerObject.
                    GetComponent<PlayerNetwork>().SetPlayerNameServerRpc(
                    playerNickname);
                clientButton.image.color = Color.green;
                Destroy(_playerNicknameObject);
                GameManager.DestroyToggleGroup();
            };
            NetworkManager.Singleton.StartClient();
            ActivateDisconnectButton();
        });
        disconnectButton.onClick.AddListener(DisconnectUiBehavior);
        GameObject.Find("PlayerScore").GetComponent<TextMeshProUGUI>().
            text = "";
    }

    private void Update(){
        if (!Input.GetKeyDown(KeyCode.Return)) return;
        if (!NetworkManager.Singleton.IsClient && !NetworkManager.
            Singleton.IsServer)
      clientButton.onClick.Invoke();
    }
}
```

Código 3 – Uso de Network Variables para armazenamento da pontuação

```
public class PlayerNetwork : NetworkBehaviour {

  private NetworkVariable<int> _score = new();

  public override void OnNetworkSpawn()
  {
    _score.OnValueChanged += TreatScoreChanged;
  }

  private void TreatScoreChanged(int previousScore, int currentScore)
  {
    GameManager.UpdateHighscoreList();
    if (IsServer)
    {
      print($"Player {OwnerClientId} had a score of {previousScore} and
          now has a score of {currentScore}");
      return;
    }

    if (!IsOwner) return;
    _scoreText.text = $"Score: {currentScore}";
  }
}
```

Código 4 – Exemplo de uso de RPC para gerar e aplicar a cor de um jogador

```
public class PlayerVisuals : NetworkBehaviour {

  private NetworkVariable<Color> _color = new();
    private SpriteRenderer _spriteRenderer;
    private PlayerNetwork _playerNetwork;

  public override void OnNetworkSpawn()
    {
        _color.OnValueChanged += TreatColorChanged;
        if (IsOwner) SetUpColorServerRpc();
        else gameObject.GetComponentInChildren<SpriteRenderer>().color =
            _color.Value;
    }

  [ServerRpc]
    private void SetUpColorServerRpc(ServerRpcParams serverRpcParams =
       default)
    {
        if (!IsServer) return;
        _color.Value = Random.ColorHSV();
        _spriteRenderer.color = _color.Value;
        print($"Applying color {_color.Value} at {NetworkManager.
           ServerTime.Time}");
    }
}
```

Código 5 – Movimentação do jogador via RPC

```
public class PlayerNetwork : NetworkBehaviour
{
  [ServerRpc]
    private void SendClientInputServerRpc(Vector2 inputValue,
       ServerRpcParams serverRpcParams = default)
    {
        if (inputValue.magnitude > 1)
        {
            inputValue.Normalize();
        }

        transform.position += new Vector3(inputValue.x, inputValue.y) *
           (BaseMovespeed * _speedMultiplier);
    }

  private void Update()
    {
        if (!IsOwner) return;

        var movement = new Vector2(Input.GetAxisRaw("Horizontal"), Input
           .GetAxisRaw("Vertical")) * Time.deltaTime;
        if (movement.magnitude > 0 && !_isInChallenge.Value)
        {
            SendClientInputServerRpc(movement);
        }
    }
}
```

Código 6 – Câmera do jogador

```
public class PlayerCamera : NetworkBehaviour
{
    private void Start()
    {
        if (!IsLocalPlayer) return;
        Transform transform1;
        (transform1 = Camera.main!.transform).SetParent(transform);
        transform1.localPosition = new Vector3(0, 0, -10);
    }
}
```

Código 7 – Código simplificado que lida com as vidas do jogador

```
public class PlayerNetwork : NetworkBehaviour
{
  public override void OnNetworkSpawn()
    {
    _lives.OnValueChanged += TreatLivesChanged;
  }

  private void TreatLivesChanged(int previousLives, int currentLives)
    {
        if (currentLives == 0)
        {
            ...
        }

        if (currentLives < previousLives)
        {
            ...
        }

        if (currentLives > previousLives)
        {
            ...
        }

        UpdatePlayerLabel();
    }

  public void AddLives(int n)
    {
        _lives.Value += n;
    }

    public void RemoveLives(int n)
    {
        _lives.Value -= n;
    }
}
```

Código 8 – Código simplificado que lida com a pontuação do jogador

```
public class PlayerNetwork : NetworkBehaviour
{
  private NetworkVariable<int> _score = new();
  public override void OnNetworkSpawn()
    {
    _score.OnValueChanged += TreatScoreChanged;
  }
  private void OnCollisionEnter2D(Collision2D other)
    {
        switch (other.gameObject.tag)
        {
            case CollectibleTag:
                _score.Value += CollectibleValue;
                break;
            case PlayerTag:
                ...
        }
    }
  private void TreatScoreChanged(int previousScore, int currentScore)
    {
        GameManager.UpdateHighscoreList();
        if (IsServer)
        {
            return;
        }

        if (!IsOwner) return;
        _scoreText.text = $"Score: {currentScore}";
    }
  private void TreatLivesChanged(int previousLives, int currentLives)
    {
    ...
    if (currentLives > previousLives)
        {
            if (IsServer)
            {
                _score.Value += ChallengeValue;
            }
        }
    ...
  }
}
```

Código 9 – Criação inicial dos coletáveis da partida

```
public class CollectiblesNetworkManager : NetworkBehaviour
{
    [SerializeField] private GameObject collectiblePrefab;

    public static CollectiblesNetworkManager Instance;

    private void Awake()
    {
        Instance = this;
    }

    public override void OnNetworkSpawn()
    {
        if (!IsServer) return;
        InitialCollectibleSpawn();
    }

    private void InitialCollectibleSpawn()
    {
        for (var i = 0; i < NumberOfCollectibles; i++)
        {
            var collectibleTransform = Instantiate(collectiblePrefab,
                new Vector3(
                    Random.Range(MinHorizontal, MaxHorizontal),
                    Random.Range(MinVertical, MaxVertical),
                    0),
                collectiblePrefab.transform.rotation);
            collectibleTransform.GetComponent<NetworkObject>().Spawn(
               true);
        }
    }
}
```

Código 10 – CollectibleBehaviour.cs

```
public class CollectibleBehaviour : NetworkBehaviour
{
    public void OnCollisionEnter2D(Collision2D other)
    {
        if (!other.gameObject.tag.Equals("Player")) return;
        MoveCollectible();
    }

    private void MoveCollectible()
    {
        gameObject.GetComponent<NetworkObject>().Despawn(destroy: false)
            ;
        transform.position = new Vector3(
            Random.Range(MinHorizontal, MaxHorizontal),
            Random.Range(MinVertical, MaxVertical),
            0);
        gameObject.GetComponent<NetworkObject>().Spawn(destroyWithScene:
            true);
    }
}
```

Código 11 – Método executado pelo jogador após uma colisão qualquer

```
public class PlayerNetwork : NetworkBehaviour
{
  ...

  private void OnCollisionEnter2D(Collision2D other)
  {
    // This will run only on server side
    switch (other.gameObject.tag)
    {
      case CollectibleTag:
        _score.Value += CollectibleValue;
        break;
      case PlayerTag:
        var opponentId = other.gameObject.GetComponent<NetworkBehaviour
            >().OwnerClientId;
        print($"Collision between players {OwnerClientId} and " +
            $"{opponentId} happened");
        _challengeOpponent.Value = opponentId;
        print($"Changed {OwnerClientId} challenge opponent value to {
            _challengeOpponent.Value} at time {NetworkManager.Singleton.
            ServerTime.Time}");
        _isInChallenge.Value = true;
        break;
    }
  }
}
```

Código 12 – Variável isInChallenge modificado para verdadeiro

```
public class PlayerNetwork : NetworkBehaviour
{
  ...

  private void TreatInChallengeChanged(bool wasInChallenge, bool
     isInChallenge)
    {
        ...
        if (!isInChallenge) {
            ...
            return;
        }

        // Entered a challenge
        print($"Player {OwnerClientId} entered a challenge at time {
           NetworkManager.Singleton.ServerTime.Time}");
        gameObject.GetComponent<CircleCollider2D>().enabled = false;
        gameObject.GetComponent<NetworkRigidbody2D>().enabled = false;

        if (IsServer && !ChallengeExists(OwnerClientId,
           _challengeOpponent.Value) && _challengeOpponent.Value != 0)
        {
            // Server will create challenge object and spawn
            _currentChallenge = CreateAndSpawnChallenge(OwnerClientId,
               _challengeOpponent.Value,
                GameManager.GetCurrentChallengeType());
            var opponentGameobject = NetworkManager.Singleton
                .ConnectedClients[_challengeOpponent.Value]
                .PlayerObject
                .gameObject;
            StartCoroutine(StartChallenge(_currentChallenge, gameObject,
                opponentGameobject));
        }
        StartCoroutine(_visuals.MakePlayerTransparent(
           PostChallengeInvincibilityTimeInSeconds));
    }
}
```

Código 13 – Método que cria o desafio

```
public static GameObject CreateAndSpawnChallenge(ulong client1, ulong
    client2, ChallengeType type)
{
  var challengePrefab = Resources.Load<GameObject>("Prefabs/Challenge");
  var instance = Instantiate(challengePrefab, Vector3.zero, Quaternion.
     identity);
  AddNewChallenge(instance);
  var challengeComponent = instance.GetComponent<Challenge>();
  if (!instance.GetComponent<NetworkObject>().IsSpawned)
  {
    if (type == ChallengeType.Random)
    {
      var random = Random.Range(0, 2);
      type = random == 0 ? ChallengeType.QuestionChallenge :
         ChallengeType.KeyboardButtonPress;
    }
    challengeComponent.Type.Value = type;
    instance.GetComponent<NetworkObject>().Spawn();
    challengeComponent.Client1Id.Value = client1;
    challengeComponent.Client2Id.Value = client2;
    var connectedClients = NetworkManager.Singleton
      .ConnectedClients;
    challengeComponent.Client1Name.Value = connectedClients[client1]
      .PlayerObject.gameObject.GetComponent<PlayerNetwork>().
         GetPlayerName();
    challengeComponent.Client2Name.Value = connectedClients[client2]
      .PlayerObject.gameObject.GetComponent<PlayerNetwork>().
         GetPlayerName();
  }
  print(
    $"<color=#00FF00>Spawned challenge network object for players {
       challengeComponent.Client1Name.Value} and {challengeComponent.
       Client2Name.Value} at time {NetworkManager.Singleton.ServerTime.
       Time}</color>");
  return instance;
}
```

Código 14 – Método que executa o desafio

```
public class ChallengeNetwork : NetworkBehaviour
{
...
public static IEnumerator StartChallenge(GameObject challenge,
    GameObject player1, GameObject player2) {
  yield return new WaitForSeconds(ChallengeStartDelayInSeconds);
  ...
  if (challengeComponent.IsQuestionChallenge()) {
    yield return new WaitForSeconds(ChallengeTimeoutLimitInSeconds);
  }
  else if (challengeComponent.IsButtonPressChallenge()) {
    yield return new WaitForSeconds(ChallengeTimeoutLimitInSeconds);
  }
  var winnerId = challengeComponent.DecideWinner();
  var loserId = winnerId == player2Id ? player1Id : player2Id;
  challengeComponent.DisplayResultsServerRpc(player1Id, player2Id,
     winnerId);
  yield return new WaitForSeconds(ChallengeWinnerTime);
  player1Network.SetIsInChallenge(false);
  player2Network.SetIsInChallenge(false);
  if (winnerId == 0) {
    // Both players lose a health
    player1Network.RemoveLives(1);
    player2Network.RemoveLives(1);
    yield break;
  }
  var loser = NetworkManager.Singleton.ConnectedClients[loserId].
     PlayerObject.gameObject;
  var winner = NetworkManager.Singleton.ConnectedClients[winnerId].
     PlayerObject.gameObject;
  loser.gameObject.GetComponent<PlayerNetwork>().RemoveLives(1);
  if (winner.gameObject.GetComponent<PlayerNetwork>().GetLives() <
     MaxLives){
    winner.gameObject.GetComponent<PlayerNetwork>().AddLives(1);
    yield break;
  }
  print($"Could not increment player {winnerId} lives because it is
     already at max lives");
}
}
```

Código 15 – Mostrando o canvas do desafio apenas para os clientes envolvidos

```
public class Challenge : NetworkBehaviour, IEquatable<Challenge> {
  ...
  private bool LocalClientInChallenge()
  {
    return (Client1Id.Value == NetworkManager.LocalClient.ClientId ||
        Client2Id.Value == NetworkManager.LocalClient.ClientId);
  }

  private void Start()
  {
    _challengeHeader.text = Client1Id.Value == NetworkManager.
       LocalClient.ClientId
      ? $"{Client1Name.Value} X {Client2Name.Value}"
      : $"{Client2Name.Value} X {Client1Name.Value}";

    if (LocalClientInChallenge())
    {
      _challengeOuterCanvas.SetActive(true);
      _challengeInnerCanvas.SetActive(true);
      _challengeInfo.SetActive(true);
    }

    ...
  }
}
```

Código 16 – Controle de tempo de vida do desafio

```
public class Challenge : NetworkBehaviour, IEquatable<Challenge>
{
  ...
  private double _challengeStartTime;
  private double _randomTimeSugar;
    private double _timeUntilWinner;
    private double _timeUntilStart;

  private void Start() {
    ...
    _challengeStartTime = NetworkManager.Singleton.ServerTime.Time;
    _timeUntilWinner = ChallengeTimeoutLimitInSeconds +
        ChallengeStartDelayInSeconds + _randomTimeSugar;
    _timeUntilStart = ChallengeStartDelayInSeconds + _randomTimeSugar;
    ...
  }

  private double PassedTime() {
    return NetworkManager.Singleton.ServerTime.Time -
        _challengeStartTime;
  }

  private bool IsInPreparation() {
    return PassedTime() <= _timeUntilStart;
  }

  private bool IsInChallenge() {
    return !IsInPreparation() && PassedTime() < _timeUntilWinner;
  }

  public override void OnNetworkSpawn() {
    ...
    CreateRandomSugarServerRpc();
  }

  [Rpc(SendTo.Server)]
  private void CreateRandomSugarServerRpc(RpcParams rpcParams = default)
      {
    _randomTimeSugar = Random.Range(-RandomTimeSugarWindow,
        RandomTimeSugarWindow);
  }

}
```

Código 17 – Controle do que o cliente vê na tela durante o desafio

```
public class Challenge : NetworkBehaviour, IEquatable<Challenge> {
  private void Update()
    {
    ...
    _challengeTimeoutText.text = GetTimeoutText();
    if (IsInPreparation() && _startIndicatorImage.color != Color.red)
    {
      _startIndicatorImage.color = Color.red;
    }
    if (IsInChallenge() && _startIndicatorImage.color != Color.green)
    {
      _startIndicatorImage.color = Color.green;
      _challengeTimeout.SetActive(true);
      ...
    }

    if (!IsInChallenge() && !IsInPreparation() &&
                (_startIndicatorImage.enabled || _challengeTimeout.
                    activeSelf))
    {
      _startIndicatorImage.enabled = false;
      _answerInput.SetActive(false);
      _challengeTimeout.SetActive(false);
    }
  }

  private string GetTimeoutText()
  {
    var timeoutCounter = ChallengeTimeoutLimitInSeconds - PassedTime() +
        _timeUntilStart;

    var timeoutText = timeoutCounter > ChallengeTimeoutLimitInSeconds ?
       ChallengeTimeoutLimitInSeconds :
      timeoutCounter < 0 ? 0 : Math.Round(timeoutCounter, 1);

    return $"{timeoutText}";
  }
}
```

Código 18 – Métodos de feedback visual

```
public class Challenge : NetworkBehaviour, IEquatable<Challenge> {

  ...

  private IEnumerator TurnScreenGreenAfterPress()
  {
    var originalColor = _challengeFlashImage.color;
    _challengeFlashImage.color = ScreenFlashGreen;
    yield return new WaitUntil(() =>
      PassedTime() >= ChallengeStartDelayInSeconds +
         ChallengeTimeoutLimitInSeconds);
    _challengeFlashImage.color = originalColor;
  }

  private void TurnScreenGreen()
  {
    _challengeFlashImage.color = ScreenFlashGreen;
  }

  private void TurnScreenRed()
  {
    _challengeFlashImage.color = ScreenFlashRed;
  }
}
```

Código 19 – Guardando o tempo de finalização dos clientes

```
public class Challenge : NetworkBehaviour, IEquatable<Challenge> {
  ...

  private void Update() {
    if (IsButtonPressChallenge()) {
      ...
      StoreFinishTimestampServerRpc(NetworkManager.Singleton.ServerTime.
          Time, Input.inputString);
      ...
    }

    if (IsQuestionChallenge()) {
      ...
      StoreFinishTimestampServerRpc(NetworkManager.Singleton.ServerTime.
          Time, Input.inputString);
      ...
    }
  }

  [ServerRpc(RequireOwnership = false)]
  private void StoreFinishTimestampServerRpc(double time, string key,
    ServerRpcParams serverRpcParams = default) {
    var clientId = serverRpcParams.Receive.SenderClientId;
    if (_clientFinishTimestamps.ContainsKey(clientId)) return;
    _clientFinishTimestamps.TryAdd(clientId, time);
  }
}
```

Código 20 – Definição do vencedor

```
public class Challenge : NetworkBehaviour, IEquatable<Challenge> {

  private Dictionary<ulong, double> _clientFinishTimestamps = new();
  ...
  public ulong DecideWinner() {
    switch (_clientFinishTimestamps.Count)
    {
      // Check if any player completed the challenge
      case 0:
        return 0;
      // Check if both players completed the challenge
      case < 2:
      {
        foreach (var clientId in new[] { Client1Id.Value, Client2Id.
            Value })
        {
          if (_clientFinishTimestamps.ContainsKey(clientId)) continue;
          // Set timestamp to ulong.MaxValue if client didn't finish
              challenge
          _clientFinishTimestamps[clientId] = ulong.MaxValue;
        }
        break;
      }
    }
    var winner = _clientFinishTimestamps[Client1Id.Value] <
        _clientFinishTimestamps[Client2Id.Value]
      ? Client1Id.Value
      : Client2Id.Value;
    return winner;
  }
}
```

Código 21 – Mostrando o resultado aos jogadores

```
public class Challenge : NetworkBehaviour, IEquatable<Challenge> {
  ...
  [Rpc(SendTo.Server)]
  public void DisplayResultsServerRpc(ulong client1Id, ulong client2Id,
     ulong winnerId,
    RpcParams rpcParams = default) {
    var client1Timestamp = _clientFinishTimestamps.GetValueOrDefault(
       client1Id, 0);
    var client2Timestamp = _clientFinishTimestamps.GetValueOrDefault(
       client2Id, 0);
    DisplayResultsClientRpc(winnerId, client1Timestamp, client2Timestamp
       ,
      RpcTarget.Group(new[] { client1Id, client2Id }, RpcTargetUse.Temp)
         );
  }
  [Rpc(SendTo.SpecifiedInParams)]
  private void DisplayResultsClientRpc(ulong winnerId, double
     client1Timestamp, double client2Timestamp,
    RpcParams rpcParams = default) {
    if (NetworkManager.Singleton.LocalClientId == winnerId) {
      TurnScreenGreen();
    }
    else {
      TurnScreenRed();
    }
    _pressCounterText.enabled = false;
    var client1Reaction = GetClientResponseTime(client1Timestamp);
    var client2Reaction = GetClientResponseTime(client2Timestamp);
    var differenceInTime = Math.Abs(client2Reaction - client1Reaction);
    var win = winnerId == Client1Id.Value ? Client1Name.Value :
       Client2Name.Value;
    _challengeInfoText.text = winnerId == 0
      ? "No winner"
      : $"{win} wins the challenge and was {Math.Round(differenceInTime,
         4)} seconds faster!";

    if (!IsQuestionChallenge()) return;
    ...
  }
  private double GetClientResponseTime(double clientTimestamp) {
    return clientTimestamp - _challengeStartTime -
       ChallengeStartDelayInSeconds - _randomTimeSugar;
  }
}
```

Código 22 – RPC do contador de pressionamentos e seu uso

```
public class Challenge : NetworkBehaviour, IEquatable<Challenge>
{
  private int _pressesCounter;
  ...
  if (IsButtonPressChallenge())
  {
    _challengeInnerCanvasHeaderText.text =
       InnerCanvasTitleKeyboardPressChallenge;
    if (Input.GetKeyDown(KeyCode.Space) &&
      LocalClientInChallenge() &&
      _pressesCounter < MaxPressesAllowed)
    {
      UpdatePressesCounterClientRpc(
        RpcTarget.Group(new[] { NetworkManager.Singleton.LocalClientId
           }, RpcTargetUse.Temp));

      ...
    }
  }

  [Rpc(SendTo.SpecifiedInParams)]
  private void UpdatePressesCounterClientRpc(RpcParams rpcParams =
     default)
  {
    _pressesCounter++;
    var pressesText = MaxPressesAllowed - _pressesCounter > 0
      ? $"{MaxPressesAllowed - _pressesCounter}"
      : "No more. Waiting other player...";
    _pressCounterText.text = $"Remaining Attempts: {pressesText}";
  }
}
```

Código 23 – Execução do desafio de pressionamento de botão

```
public class Challenge : NetworkBehaviour, IEquatable<Challenge>
{
  private int _pressesCounter;
  ...
  if (IsButtonPressChallenge())
  {
    _challengeInnerCanvasHeaderText.text =
       InnerCanvasTitleKeyboardPressChallenge;
    if (Input.GetKeyDown(KeyCode.Space) &&
      LocalClientInChallenge() &&
      _pressesCounter < MaxPressesAllowed)
    {
      UpdatePressesCounterClientRpc(
        RpcTarget.Group(new[] { NetworkManager.Singleton.LocalClientId
           }, RpcTargetUse.Temp));

      if (PassedTime() > _timeUntilStart &&
        PassedTime() < _timeUntilWinner &&
        IsInChallenge())
      {
        // Send player positive feedback
        if (!_clientFinishTimestamps.ContainsKey(NetworkManager.
           Singleton.LocalClientId))
        {
          StartCoroutine(TurnScreenGreenAfterPress());
          _pressCounterText.enabled = false;
        }

        StoreFinishTimestampServerRpc(NetworkManager.Singleton.
           ServerTime.Time,
          Input.inputString);
      }

      if (PassedTime() < ChallengeStartDelayInSeconds)
      {
        // Send player negative feedback
        StartCoroutine(FlashScreenRed());
      }
    }
  }
}
```

Código 24 – Componente QuestionManager

```
public class QuestionManager : NetworkBehaviour {

  private static Questions _questions;

  public override void OnNetworkSpawn() {
    if (IsServer) LoadQuestions();
  }

  private static Questions LoadQuestions() {
    var textAsset = Resources.Load<TextAsset>("Questions/Questions");
    if (textAsset == null) {
      Debug.LogError("Failed to load file!");
      return null;
    }
    _questions = JsonUtility.FromJson<Questions>(textAsset.text.ToLower
        ());
    if (_questions.questions == null) {
      Debug.LogError("Failed to parse data!");
    }
    return _questions;
  }

  public static Question GetRandomQuestion() {
    if (_questions == null || _questions.questions.Length == 0) {
      print("No questions loaded!");
      return null;
    }

    var randomIndex = Random.Range(0, _questions.questions.Length);
    print($"Random index: {randomIndex}");
    print("Random question: " + _questions.questions[randomIndex].query)
        ;
    return _questions.questions[randomIndex];
  }
}
```

Código 25 – Execução do desafio de pergunta e resposta

```
public class Challenge : NetworkBehaviour, IEquatable<Challenge>
{
  ...
  if (IsQuestionChallenge())
  {
    _challengeInnerCanvasHeaderText.text =
       InnerCanvasTitleQuestionChallenge;
    if (LocalClientInChallenge())
    {
      if (PassedTime() >= ChallengeStartDelayInSeconds &&
        PassedTime() < _timeUntilWinner &&
        IsInChallenge() &&
        _challengeInnerCanvasHeaderText.text.Equals(
           InnerCanvasTitleQuestionChallenge))
      {
        _challengeInnerCanvasHeaderText.text = Query.Value.ToString();
      }

      // Send the answer to clients with a client rpc
      if (Input.GetKeyDown(KeyCode.KeypadEnter) || Input.GetKeyDown(
         KeyCode.Return))
      {
        if (PassedTime() > _timeUntilStart &&
          PassedTime() < _timeUntilWinner &&
          IsInChallenge())
        {
          // Check if the answer is correct
          var playerAnswer = _answerInputText.text;
          var correctAnswer = Answer.Contains(playerAnswer.ToLower());
          if (correctAnswer)
          {
            TurnScreenGreen();
            StoreFinishTimestampServerRpc(NetworkManager.Singleton.
               ServerTime.Time,
              Input.inputString);
            _answerInputText.enabled = false;
          }
          else
          {
            StartCoroutine(FlashScreenRed());
            _answerInputText.Select();
            _answerInputText.ActivateInputField();
          }
        }
      }
    }
  }
}
```

Código 26 – Variável isInChallenge modificado para falso

```
public class PlayerNetwork : NetworkBehaviour
{
  ...

  private IEnumerator PlayerPostChallengeBehavior(float multiplier)
    {
        _speedMultiplier = multiplier;
        yield return new WaitForSeconds(
           PostChallengeInvincibilityTimeInSeconds);
        _collider.enabled = true;
        _networkRigidbody.enabled = true;
        _speedMultiplier = 1;
    }

  private void TreatInChallengeChanged(bool wasInChallenge, bool
     isInChallenge) {
        ...
        if (!isInChallenge) {
            StartCoroutine(PlayerPostChallengeBehavior(
               PostChallengeSpeedMultiplier));
            if (IsServer) {
                if (ChallengeExists(OwnerClientId, _challengeOpponent.
                   Value)) {
                    RemoveChallenge(_currentChallenge);
                    _currentChallenge.GetComponent<NetworkObject>().
                       Despawn();
                    _challengeOpponent.Value = 0;
                }
            }

            if (!IsOwner) return;
            return;
        }

        // Entered a challenge
        print($"Player {OwnerClientId} entered a challenge at time {
           NetworkManager.Singleton.ServerTime.Time}");
        ...
    }
}
```

Código 27 – Código de movimentação antigo

```
[ServerRpc]
private void SendClientInputServerRpc(Vector2 inputValue,
    ServerRpcParams serverRpcParams = default)
{
  if (inputValue.magnitude > 1)
  {
    inputValue.Normalize();
  }

  transform.position += new Vector3(inputValue.x, inputValue.y) * (Time.
     deltaTime * BaseMovespeed * _speedMultiplier);
}

private void Update()
{
  if (!IsOwner) return;

  var movement = new Vector2(Input.GetAxisRaw("Horizontal"), Input.
     GetAxisRaw("Vertical"));
  if (movement.magnitude > 0 && !_isInChallenge.Value)
  {
    SendClientInputServerRpc(movement);
  }
}
```

APÊNDICE B – ESTADOS POSSÍVEIS DO DESAFIO DE PRESSIONAMENTO DE BOTÃO.

Figura 20 – Desafio de pressionamento de botão no estágio de preparação

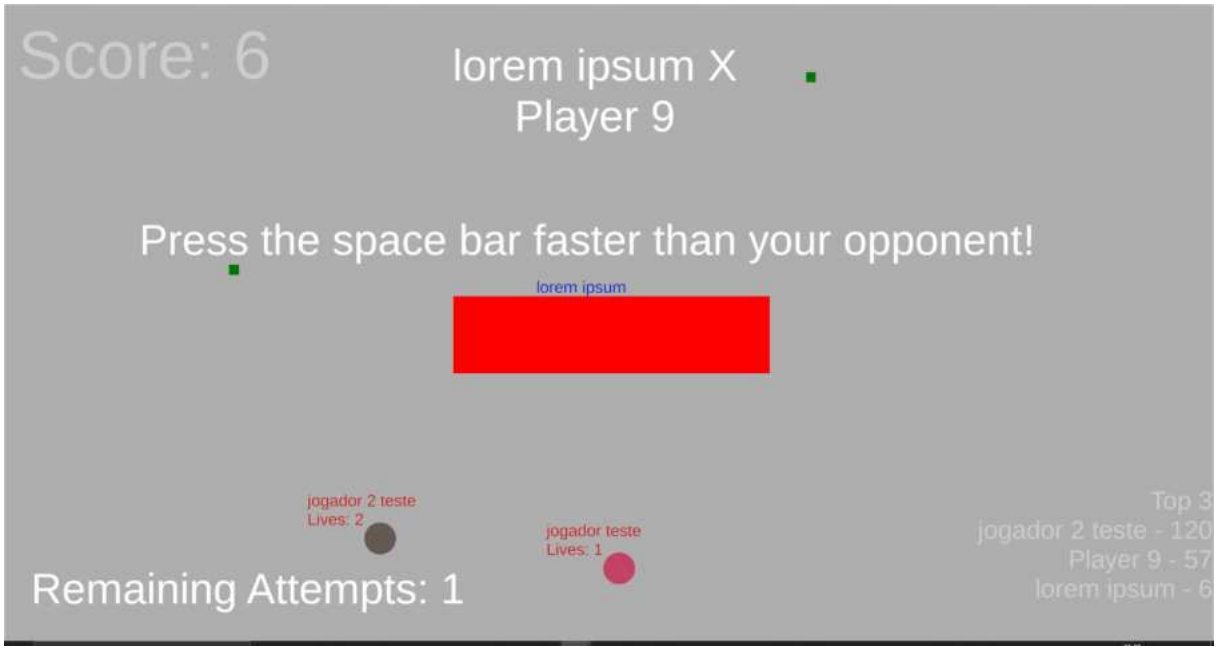


Figura 21 – Desafio de pressionamento de botão - sem pressionamentos restantes

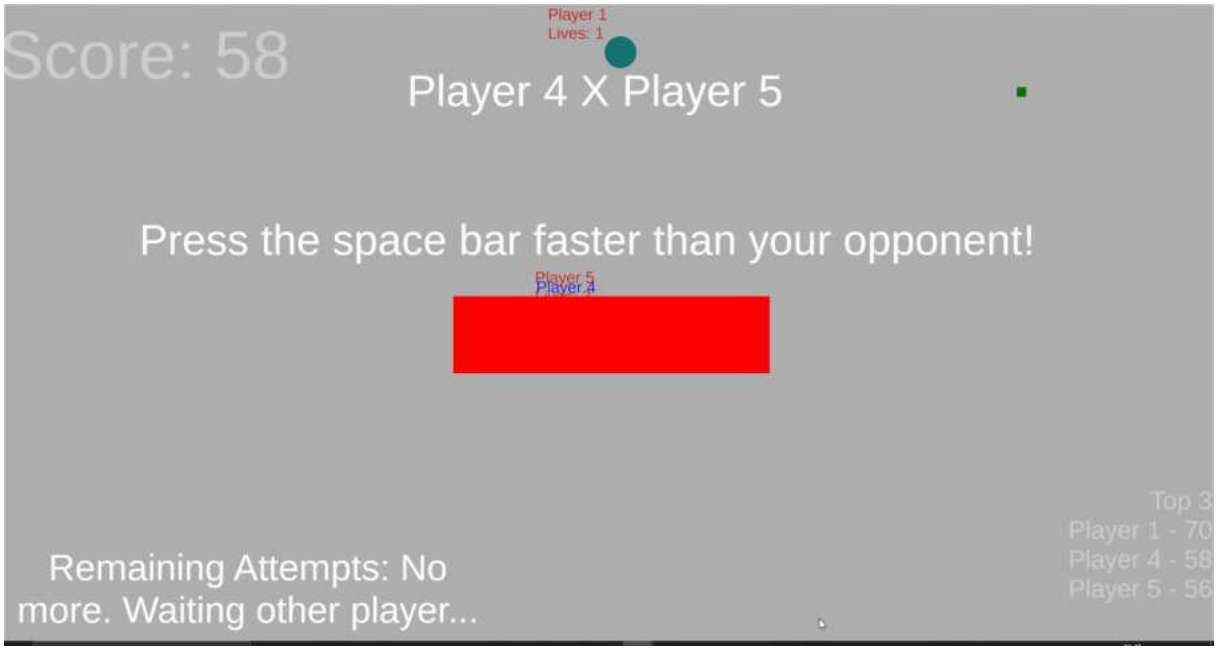


Figura 22 – Desafio de pressionamento de botão no estágio útil

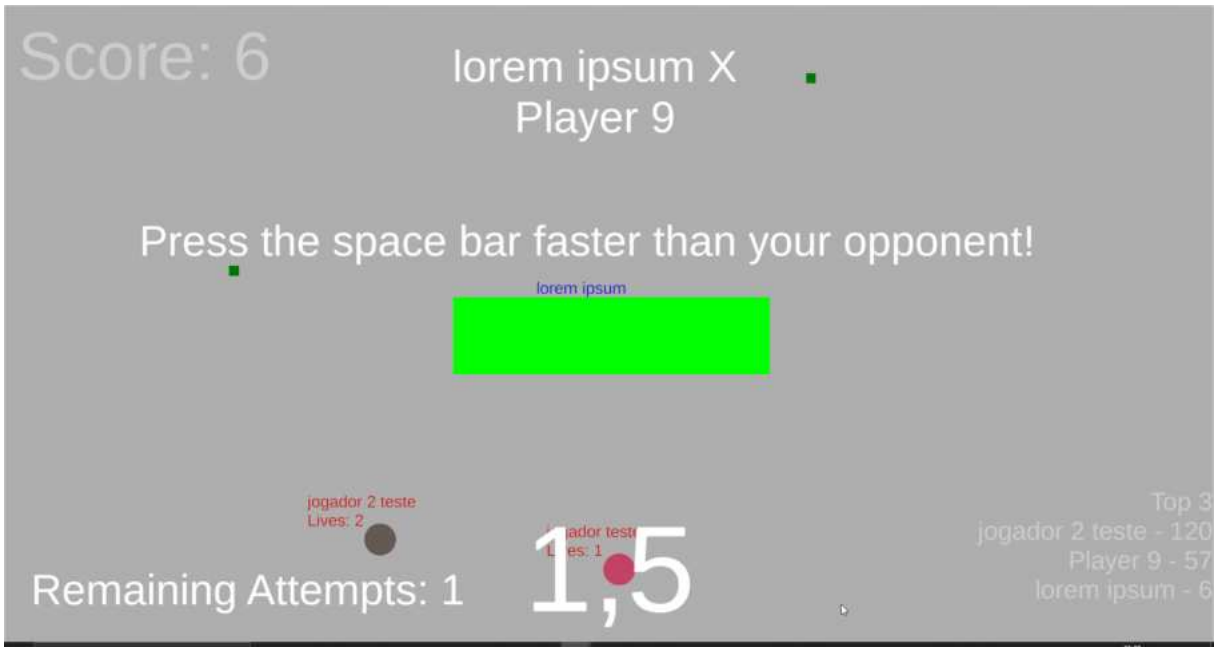


Figura 23 – Desafio de pressionamento de botão concluído

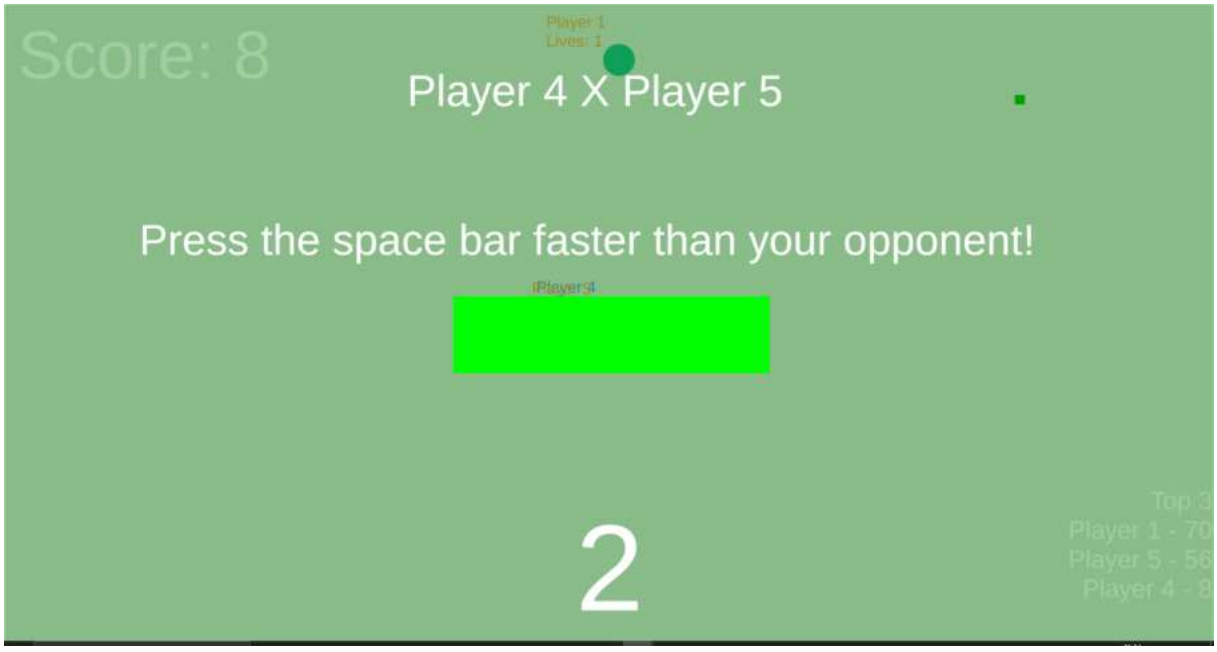


Figura 24 – Desafio de pressionamento de botão - vitória

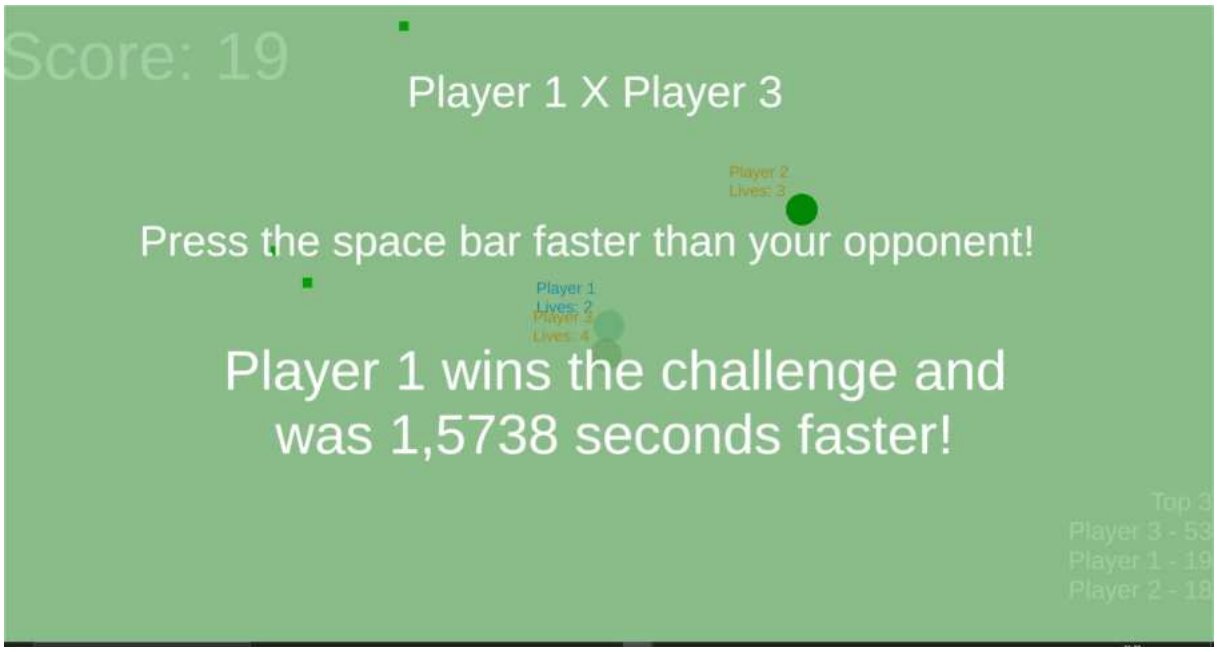


Figura 25 – Desafio de pressionamento de botão - derrota

Figura 26 – Desafio de pressionamento de botão - sem vencedor

APÊNDICE C – ESTADOS POSSÍVEIS DO DESAFIO DE PERGUNTA E RESPOSTA.

Figura 27 – Desafio de pergunta e resposta no estágio de preparação

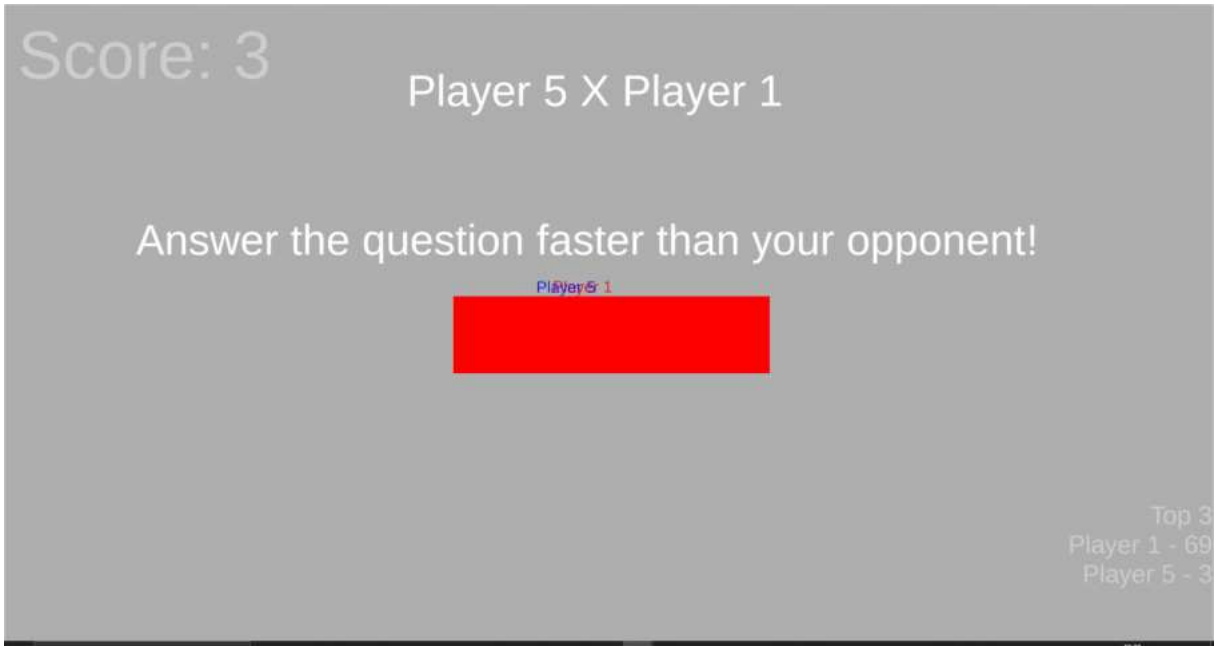


Figura 28 – Desafio de pergunta e resposta - feedback de resposta correta

Figura 29 – Desafio de pergunta e resposta - feedback de resposta errada

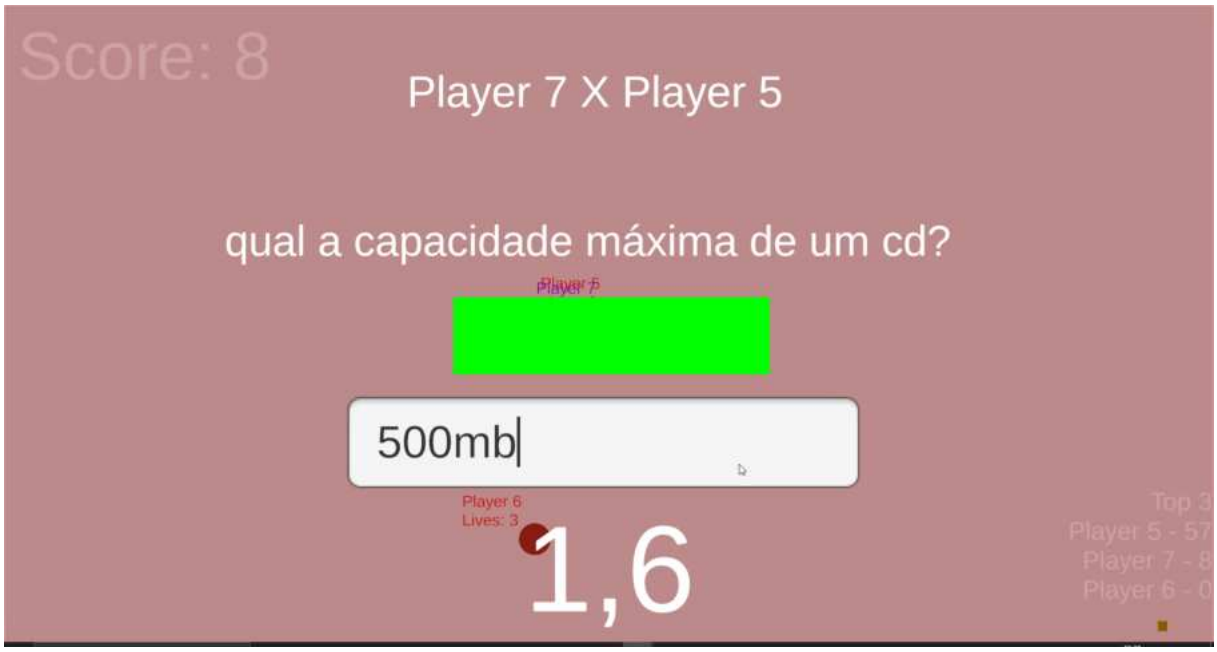


Figura 30 – Desafio de pergunta e resposta - vitória

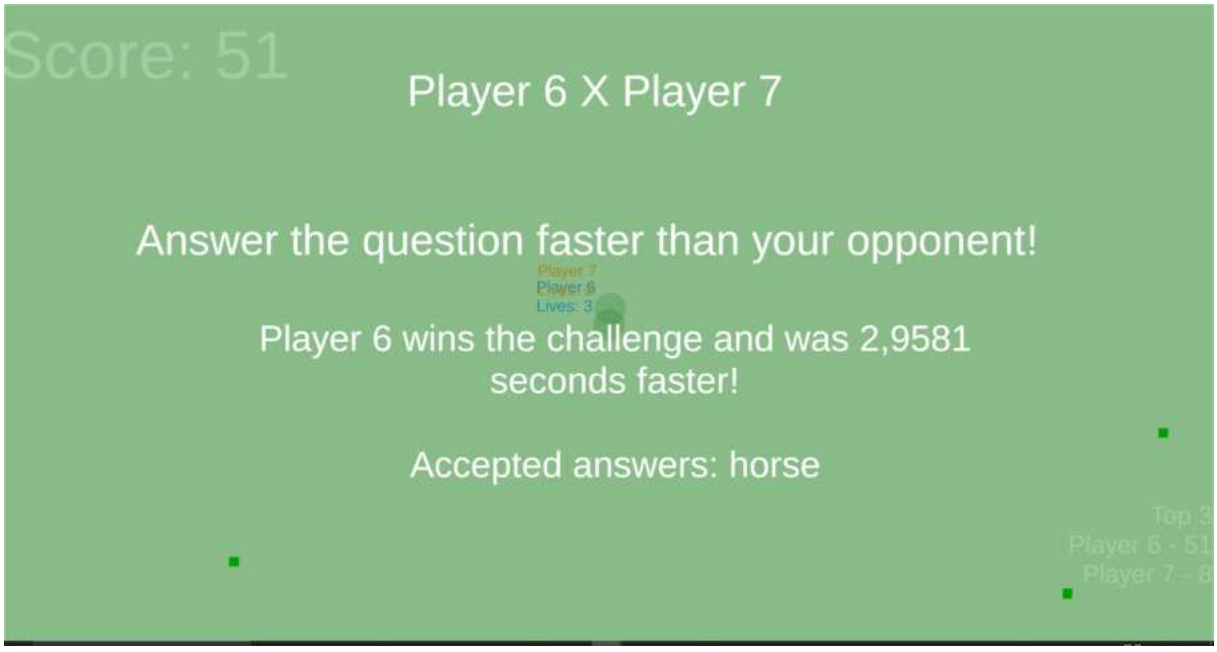


Figura 31 – Desafio de pergunta e resposta - derrota

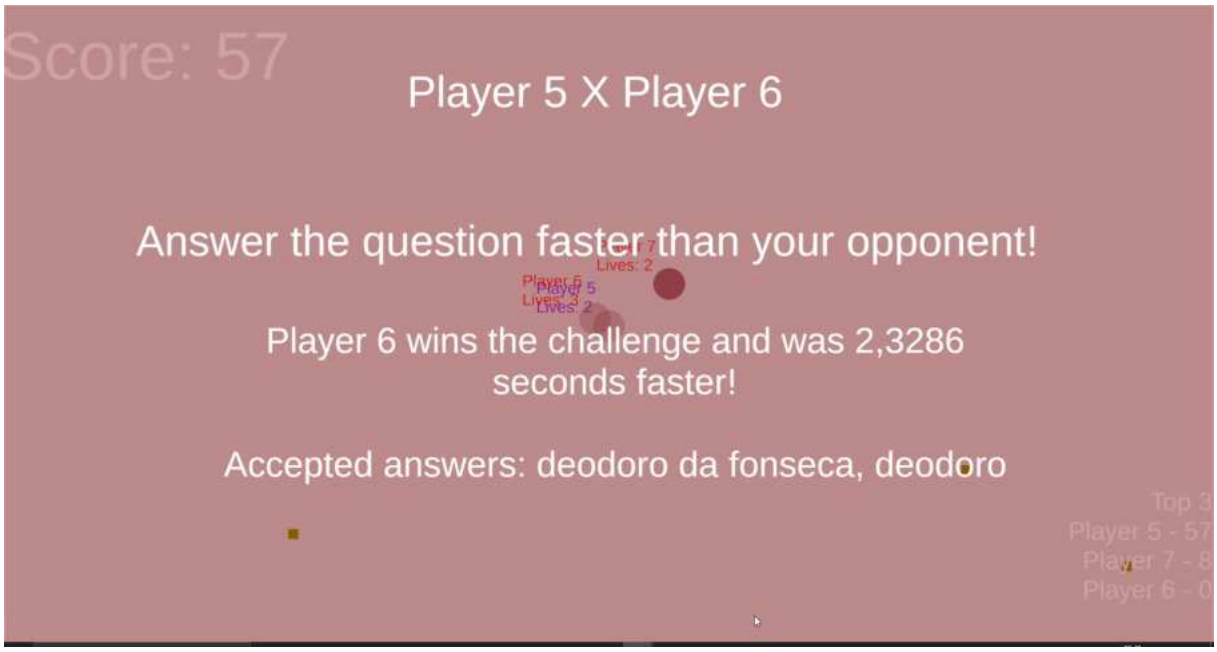


Figura 32 – Desafio de pergunta e resposta - sem vencedor