PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA DE CIÊNCIAS EXATAS E DA COMPUTAÇÃO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DESENVOLVIMENTO DE UM JOGO DIGITAL EDUCACIONAL SOBRE REDES DE COMPUTADORES

JOÃO PAULO NUNES BORGES

GOIÂNIA
2020

JOÃO PAULO NUNES BORGES

DESENVOLVIMENTO DE UM JOGO DIGITAL EDUCACIONAL SOBRE REDES DE COMPUTADORES

Trabalho de Conclusão de Curso apresentado à Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Orientador:

Prof. Me. Rafael Martins Leal

Banca examinadora:

Prof. Dra. Carmen Cecilia Centeno

Prof. Me. Fernando G. Abadia

GOIÂNIA

2020

JOÃO PAULO NUNES BORGES

DESENVOLVIMENTO DE UM JOGO DIGITAL EDUCACIONAL SOBRE REDES DE COMPUTADORES

Este Trabalho de Conclusão de Curso julgado adequado para obtenção do título de Bacharel em Ciência da Computação, e aprovado em sua forma final pela Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, em __/__/____.

Prof. Ma. Ludmilla Reis Pinheiro dos Santos
Coordenadora de Trabalho de Conclusão de Curso

Banca examinadora:

Orientador: Prof. Me. Rafael Martins Leal

Prof. Dra. Carmen Cecilia Centeno

Prof. Me. Fernando G. Abadia

GOIÂNIA
2020

Aos meus pais, minha esposa, minha filha e aos meu amigos,que sempre me apoiaram para que eu chegasse até aqui.

AGRADECIMENTOS

Agradeço pela oportunidade que me foi concedida, de poder estar em uma instituição de ensino como a Pontifícia Universidade Católica de Goiás, que me deu o incentivo e o conhecimento para poder escrever o meu próprio destino.

Agradeço ao meu pai João Borges da Silva, que me apoiou em meus estudos e me deu segurança para investir no meu futuro, e também a minha mãe Maria Francinete Rego Nunes Borges, que sempre apoiou minhas decisões, e teve orgulho de mim.

Ao meu grande amigo Douglas Vieira do Nascimento que esteve comigo desde o primeiro dia deste curso, até o presente momento, e que trilhou junto comigo este caminho.

Agradeço enormemente ao professor Prof. Me. Rafael Martins Leal, por me acompanhar e me guiar no presente trabalho, sempre com paciência aos meus atrasos e muito companheirismo, o que tornou esse desafio mais leve de se encarar. Agradeço a todos os professores que estiveram comigo, no papel de tutores e amigos, professores que tornaram a experiência da graduação memorável, professor Joriver Canedo, Sibelius Vieira, Clarimar Coelho, Marco Antonio Menezes e a todo corpo docente. Agradeço a Professora Carmen Centeno, que no papel de coordenadora, sempre nos ajudou muito com quaisquer problemas que viessem a surgir. Um grande agradecimento em especial para os professores Alexandre Ribeiro, Arlindo Galvão e Max Gontijo, que foram grandes amigos e inspiração para mim.

Agradeço enormemente a minha esposa Natalie Tolentino Serafim, que sempre me apoiou e me ajudou em tudo que pode, mesmo nos momentos mais difíceis, mesmo quando nossa filha estava para chegar, ela ainda esteve ao meu lado, para me ajudar a conquistar os meus sonhos e torná-los realidade ao seu lado.

Agradeço também a todos os produtores de conteúdo de ensino gratuíto, ao qual tanto recorri durante todos esses anos. Um grande agradecimento para os canais, Nerckie, Projeto Plin, Matemática Rio, Univesp, Linguagem C Programação Descomplicada, omatematico.com, Hemerson Pistori, Paula Ramos, Instituto De Matemática Pura e Aplicada, Luis Antonio Aguirre, Documentários Revolução Científica, Khan Academy, labpib, SmallAdvantages e a todos os outros canais que se dedicam a proporcionar conhecimento de qualidade, livre e gratuíto a todos.

No mais, agradeço a todos os demais que estiveram comigo nesta jornada, que vivenciaram de longe ou de perto, mas que fizeram parte deste trajeto, seja apoiando ou comemorando, ajudando ou tornando a caminhada mais leve, foi um prazer ter compartilhado estes anos com vocês.

RESUMO

Neste trabalho foi desenvolvido um jogo educativo chamado Packet Journey, onde seu objetivo consiste em tentar propiciar uma abstração da viagem realizada por um pacote de dados através da Internet. Este trabalho foi desenvolvido focado na camada de rede da internet, onde o jogador irá realizar sua jornada de salto em salto entre os roteadores, calculando as melhores rotas para sua viagem sempre que necessário, e percorrendo seu caminho pelo interior dos cabos de transmissão, até então chegar ao seu destino final. Espera-se que este jogo consiga trazer alguns conceitos não triviais sobre redes de comunicação para um público diverso, de uma forma interativa, divertida e educacional, atuando tanto como um bom jogo casual de raciocínio, como uma porta de entrada para um possível interesse em conhecer um pouco mais sobre o funcionamento de algo tão presente no nosso dia a dia como a Internet.

***Palavras-chave**: Jogo Educacional, Internet, Redes de Comunicação.*

ABSTRACT

In this work, an educational game called Packet Journey was developed, where its objective is to try to provide an abstraction of the journey made by a data package through the Internet. This work was developed focused on the network layer of the internet, where the player will perform his journey from jump to jump between routers, calculating the best routes for his trip whenever necessary, and traveling his way through the transmission cables, until then reach your final destination. It is hoped that this game will be able to bring some non-trivial concepts about communication networks to a diverse audience, in an interactive, fun and educational way, acting both as a good casual reasoning game, and as a gateway to a possible interest in know a little more about the functioning of something as present in our daily lives as the Internet.

Keywords*: Educational Game, Internet, Communication Networks.*

LISTA DE FIGURAS

LISTA DE SIGLAS

AS *Autonomous System*, em português, Sistema Autônomos

ECMP *Equal Cost MultiPath*

IDE *Integrated development environment*, em português, ambiente integrado de desenvolvimento

IETF *Internet Engineering Task Force*

ISP *Internet Service Provider*, em português, Provedor de Serviços de Internet

JIT *just-in-time*

OSPF *Open Shortest Path First*

RIP *Routing Information Protocol*, em português, Protocolo de Informação de Roteamento

SUMÁRIO

1 INTRODUÇÃO

Os videogames são uma porta de entrada para o mundo da tecnologia, estando entre os meios mais diretos de acesso de crianças e jovens. A maioria das crianças ocidentais brincam com consoles de videogames e seu primeiro contato com computadores é por meio de um jogo de computador (GROS, 2003).

Os jogos sempre fizeram parte da vida do ser humano, não somente na infância mas por todas as fases da vida. Por suas características, os jogos podem possuir aplicações eficientes no âmbito educacional, pois eles são motivadores e divertidos ao mesmo tempo, facilitando o aprendizado e retenção daquilo que foi aprendido, exercitando as funções mentais e intelectuais daquele que o joga. Além disso, nos jogos, é necessário que o jogador desenvolva o reconhecimento e entendimento de regras, identificação dos contextos em que elas se aplicam e até mesmo a invenção de novos contextos para a modificação das aplicabilidades das mesmas. Através do jogo se revelam a autonomia, criatividade, originalidade e a possibilidade de simular e experimentar situações onde muitas vezes não seria possível no nosso cotidiano (TAROUCO et al., 2004).

Os jogos digitais têm chamado a atenção de estudiosos em uma variedade de disciplinas. Hoje, estudiosos de campos diversos que vão desde design de mídia, literatura, ciência da computação, educação, até estudos de teatro, têm juntos contribuído para a compreensão deste emergente meio e seu fenômeno como ferramenta de aprendizagem (HSIAO, 2007).

Segundo o professor Geraldo Magela (SILVA, 2008), a questão primordial sobre a implantação de novas tecnologias para o suporte à educação, está em despertar na pessoa o interesse e motivação para buscar a informação desejada, de modo que, o paradigma tradicional da educação, possa ser convertido para uma educação como entretenimento.

Neste contexto, este trabalho visa desenvolver um jogo digital, com a temática de redes de computadores, onde o jogador possa ter uma experiência divertida e ao mesmo tempo possa exercitar seu cérebro com a resolução de problemas advindos das redes de computadores e também aprender um pouco sobre o contexto de comunicação pela internet, que é um ponto tão presente em nosso dia a dia.

1.1 OBJETIVOS

1.1.1 Objetivo geral

O Desenvolvimento de um jogo digital com a temática de redes de computadores.

1.1.2 Objetivos específicos

Utilizar dos conceitos contidos em redes de computadores para criar desafios para que os jogadores possam aprender e se divertir enquanto jogam.

Trazer o contexto da comunicação pela internet, propiciando uma abstração sobre a

jornada dos pacotes de rede pela camada de rede.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados com maior profundidade, os conceitos utilizados durante o desenvolvimento deste trabalho, assim como as técnicas utilizadas para alcançar o objetivo proposto.

2.1 Jogos Digitais

Provavelmente todos nós temos uma noção intuitiva muito boa do que é um jogo. O termo geral "jogo"abrange jogos de tabuleiro como xadrez e Banco Imobiliário, jogos de cartas como poker e *blackjack*, jogos de cassino como roleta e caça-níqueis, jogos de guerra militar, jogos de computador, vários tipos de jogos entre crianças, e a lista continua. Na academia, às vezes falamos da teoria dos jogos, em que múltiplos agentes selecionam estratégias e táticas para maximizar seus ganhos dentro da estrutura de um conjunto bem definido de regras de jogo. Quando usada no contexto de consoles ou entretenimento baseado em computador, a palavra “jogo” geralmente evoca imagens de um mundo virtual tridimensional apresentando algo como um humanóide, animal ou veículo como o personagem principal sob o controle do jogador. Raph Koster em seu livro, *A Theory of Fun for Game Design* de 2013, define um jogo como uma experiência interativa que fornece ao jogador uma sequência cada vez mais desafiadora de padrões que ele aprende e eventualmente domina. A afirmação de Koster é que as atividades de aprendizagem e domínio estão no cerne do que chamamos de "diversão", assim como uma piada se torna engraçada no momento em que a "entendemos"reconhecendo o padrão (GREGORY, 2009)

2.2 Unity

A Unity, também conhecida como Unity3D, é uma *Game Engine* (motor de jogo) e Ambiente Integrado de desenvolvimento (*Integrated development environment*, IDE) para a criação de mídia interativa, normalmente videogames.

Como colocado pelo CEO da Unity Technologies, David Helgason, a Unity é um conjunto de ferramentas usado para construir jogos, e é a tecnologia que executa os gráficos, o áudio, a física e as interações na rede. A Unity é famosa por seus recursos de prototipagem rápida e grande número de alvos de publicação. A primeira versão da Unity (1.0.0) foi criada pelos colegas: David Helgason, Joachim Ante e Nicholas Francis na Dinamarca. O produto foi lançado em 6 de junho de 2005. O objetivo era criar um motor de jogo acessível, com ferramentas profissionais para desenvolvedores de jogos amadores e democratizar a indústria de desenvolvimento de jogos. Quando lançado originalmente, a Unity estava disponível apenas para Mac OS X, e os desenvolvedores só podiam implantar suas criações em algumas plataformas. Agora, a Unity é oferecida em mais de uma dúzia de plataformas (HAAS, 2014).

Um motor de jogo é um programa de computador e/ou conjunto de bibliotecas, para

simplificar e abstrair o desenvolvimento de jogos eletrônicos ou outras aplicações com gráficos em tempo real, para videogames e/ou computadores rodando sistemas operacionais. A funcionalidade principal normalmente fornecida por um mecanismo de jogo inclui um motor gráfico para renderizar gráficos 2D ou 3D, um mecanismo de física ou detecção de colisão (e resposta de colisão), som, script, animação, inteligência artificial, rede, *streaming*, gerenciamento de memória, segmentação, suporte de localização, gráfico de cena e pode incluir suporte de vídeo para cinemática.(WIKIPEDIA, 2020).

Os motores de jogos são as porcas e os parafusos que ficam nos bastidores de cada videogame. O motor é responsável por tomar decisões que vão desde a arte até a matemática que decide cada quadro na tela. Motores 3D modernos são um dilúvio de códigos meticulosamente escritos e, como tal, uma vez usados para o propósito pretendido (que é a produção de um jogo para o qual foram feitos), esses motores são frequentemente vendidos, modificados e reutilizados. A estrutura e os comandos dos motores de jogos comerciais requerem milhares e milhares de linhas de código para funcionar. Este código é entregue na Unity com a ajuda da compilação *just-in-time* (JIT), usando a biblioteca C++ de código aberto Mono. Ao usar a compilação JIT, mecanismos como o Unity podem tirar vantagem da compilação de alta velocidade, em que o código que você escreverá para o Unity é compilado para Mono antes de ser executado. Isso é crucial para jogos que devem executar código em momentos específicos durante o tempo de execução. Além da biblioteca Mono, o Unity também tira proveito de outras bibliotecas de software em sua funcionalidade, como o mecanismo de física PhysX da Nvidia, OpenGL e DirectX para renderização 3D e OpenAL para áudio. Todas essas bibliotecas são embutidas no aplicativo, para que você não precise se preocupar em aprender como usá-las individualmente (GOLDSTONE, 2010).

2.2.1 Conceitos da Unity

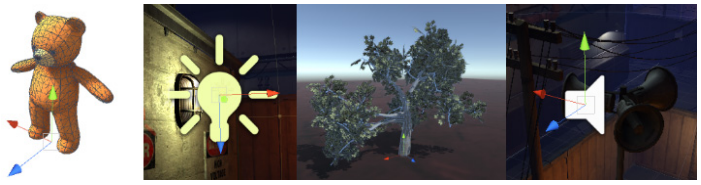
Nesta seção, são apresentandos conceitos importantes da Unity que foram utilizados para o desenvolvimento do presente trabalho. Todo o conteúdo foi retirada do manual oficial da Unity disponibilizado online gratuitamente, e as indicações de onde encontrar mais informações estão indicadas em cada subseção conforme citações.

2.2.1.1 GameObject

O *GameObject* é o conceito mais importante no Editor do Unity. Todos os objetos em um jogo desenvolvido com a Unity são um *GameObject*, dos personagens e itens colecionáveis as luzes, câmeras e efeitos especiais. Contudo um *GameObject* não é capaz de realizar nada por si só. É necessário que sejam atribuídas propriedades a ele, para que este possa se tornar um personagem, um ambiente ou um efeito especial, por exemplo. A Fig.1 ilustra quatro *GameObjects* dotados de diferentes propriedades.

GameObjects são os objetos fundamentais na Unity que representam personagens, ade-

Figura 1 – Quatro tipos diferentes de GameObject: um personagem animado, uma luz, uma árvore e uma fonte de áudio

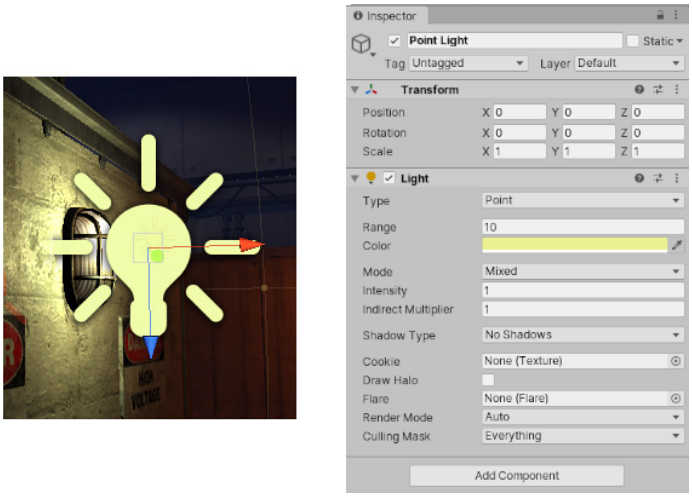


Fonte: https://docs.unity3d.com/Manual/GameObjects.html

reços e cenários. Eles não realizam muito por si próprios, mas atuam como contêineres para *Components*, que implementam funcionalidades e definem comportamentos. Para dar a um *GameObject* as propriedades necessárias para se tornar uma luz, uma árvore ou uma câmera, é necessário adicionar ao *GameObject* os *components* que implementam estas propriedades.

Os *components* definem o comportamento desse GameObject. Dependendo do tipo de objeto que se deseja criar, é necessária a adição de diferentes combinações de components a um GameObject. A Unity possui muitos tipos de components nativos diferentes, e também é possível criar novos components utilizando-se da API de script da Unity. Como um exemplo, para que um objeto na cena se torne uma fonte de emissão de luz, é necessária a adição do *Light component* nativo a este objeto. A Fig.2 ilustra o exemplo descrito (UNITYUSERMANUAL, 2020a).

Figura 2 – Um GameObject acrescido do componente nativo de Luz da Unity

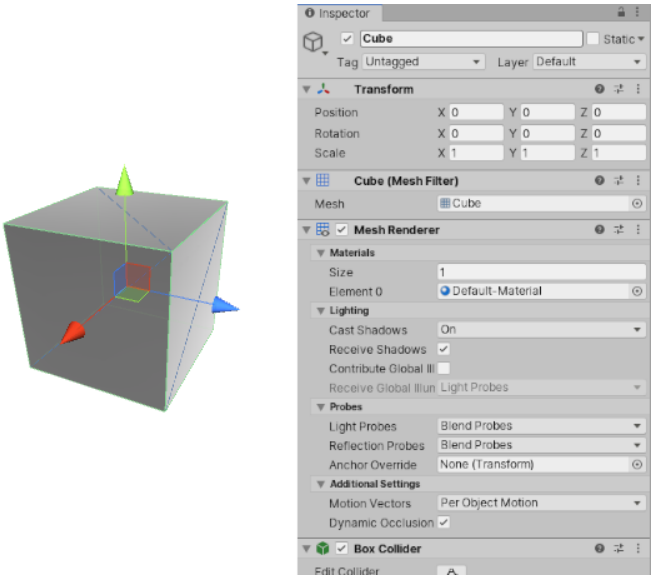


Fonte: https://docs.unity3d.com/Manual/GameObjects.html

Ao adicionarmos um objeto nativo da Unity do tipo Cubo a uma cena, podemos observar pela janela Inspector, que o mesmo possui os components *Mesh Filter* e *Mesh Renderer* para

desenhar a superfície do cubo, e um *component* de *Box Collider* 2.2.1.5, para representar o volume do sólido em termos do mecanismo de física. A Fig.3 ilustra um cubo sólido e seus respectivos components (UNITYUSERMANUAL, 2020a).

Figura 3 – Um GameObject de cubo simples com vários componentes



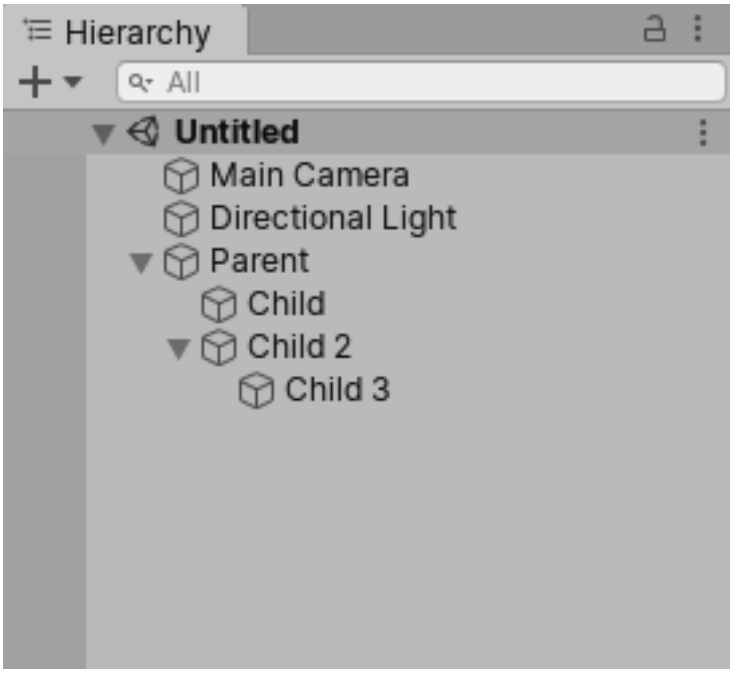
Fonte: https://docs.unity3d.com/Manual/GameObjects.html

Um *GameObject* sempre possui um component chamado *Transform* anexado a ele, este component possui as informações referentes a posição e orientação do objeto, e não é possível removê-lo. Todos os outros tipos de *components*, nativos ou criados pelo usuário, podem ser adicionados a partir do editor da Unity ou de um script (UNITYUSERMANUAL, 2020a).

2.2.1.2 Hierarchy

A janela *Hierarchy* contém uma lista de cada *GameObject* na cena atual. Alguns deles são instâncias diretas de arquivos de *Asset* (como modelos 3D) e outros são instâncias de *Prefabs*, que são *GameObjects* personalizados que constituem a maior parte de um jogo. Quando *GameObjects* são adicionados ou removidos de uma cena (ou quando a mecânica do jogo os adiciona ou remove), eles aparecem ou desaparecem da *Hierarchy* também. Por padrão, a janela *Hierarchy* lista os *GameObjects* por ordem de criação, com os *GameObjects* criados mais recentemente na parte inferior. A Unity possui um conceito chamado de *Parenting* onde ao criar um grupo de *GameObjects*, o *GameObject* ou cena que estiver no nível superior é chamado de "*GameObject* pai", e todos os *GameObjects* agrupados abaixo dele são chamados de "*GameObjects* filhos" ou "filhos". Também é possível criar *GameObjects* pai-filho aninhados (chamados de "descendentes" do *GameObject* pai de nível superior). A Fig 4 ilustra uma tela de *Hierarchy* onde existem *GameObjects* pais e filhos (UNITYUSERMANUAL, 2020b).

Figura 4 – Child and Child 2 são os GameObjects filhos do pai. O Child 3 é um GameObject filho do Child 2 e um GameObject descendente do Paren

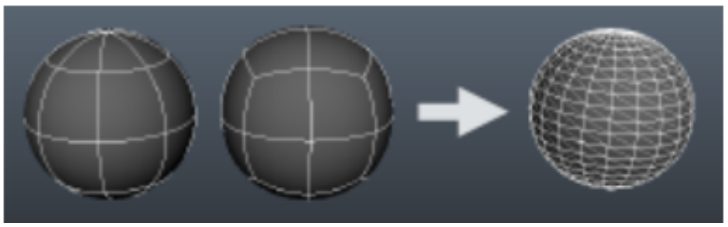


Fonte: https://docs.unity3d.com/Manual/Hierarchy.html

2.2.1.3 Mesh

Na Unity a classe *Mesh* contém os dados referentes aos *vertex* (no contexto da unity, um *vertex* é uma estrutura de dados que guarda informações de posição, vetores normais, coordenadas etc.) e dados das "faces", sendo as faces, planos geométricos formados pela ligação de vários *vertex*. A Unity suporta vários tipos de topologia de *mesh*, quadrangulares, triangulares, de linhas ou pontos. Por exemplo, para *mesh* de linha, cada linha é composta de dois índices de *vertex* e assim por diante. Mas normalmente as faces são compostas por triângulos. Utilizando-se desta estrutura, é possível guardar as informações necessárias para se compor um modelo 3D. A Fig 5 exemplifica um *mesh*.

Figura 5 – Exemplo de Mesh



Fonte: https://docs.unity3d.com/Manual/class-Mesh.html

Conceitualmente, todos os dados de *vertex* são guardados em vetores de mesmo tamanho. Por exemplo, se existe um mesh de 100 *vertex*, e é requerido que para cada *vertex* sejam armazenados posição, normal e duas coordenadas de texturas, então o *mesh* deve ter vértices, normais, matrizes uv e uv2, cada uma tendo 100 de tamanho. Os dados para o i-ésimo *vertex* estão no índice "i"em cada matriz(UNITYUSERMANUAL, 2020c).

2.2.1.4 Rigidbody

Um Rigidbody é um component que permite que GameObjects atuem sob o controle do mecanismo de física. O Rigidbody pode receber forças e torque para fazer objetos se moverem de forma realista. Qualquer GameObject deve conter um Rigidbody para ser influenciado pela gravidade, agir sob forças adicionais por meio de script ou interagir com outros objetos por meio do mecanismo de física NVIDIA PhysX (UNITYUSERMANUAL, 2020d).

2.2.1.5 Colliders

Os *Collider Components* definem a forma de um *GameObject* para fins de colisões físicas. Um *Collider*, que é invisível, não precisa ter exatamente a mesma forma que o*mesh* do *GameObject*. Uma forma aproximada do *mesh* é frequentemente mais eficiente em termos de desempenho e indistinguível no jogo. Os *colliders* mais simples (e com menos uso de processador) são os *colliders* do tipo primitivos. Em 3D, são: *Box Collider*, *Sphere Collider* e *Capsule Collider*. Em 2D, os *colliders* primitivos são *Box Collider 2D* e *Circle Collider 2D*. É possível também, adicionar qualquer quantidade de *colliders* primitivos a um único *GameObject* para criar um *compound collider*. Os *colliders* interagem entre si de maneira diferente dependendo de como seus *Rigidbody Components* estão configurados. As três configurações importantes são o *Static Collider* (ou seja, nenhum *Rigidbody* está conectado), *Rigidbody Collider* e *Kinematic Rigidbody Collider* (UNITYUSERMANUAL, 2020e).

2.2.1.5.1 Static Colliders

Um *static collider* é um *GameObject* que tem um *Collider*, mas não tem um *Rigidbody*. Os *static colliders* são utilizados principalmente para geometria de nível, que sempre permanece no mesmo lugar e nunca se move. Objetos que possuem um *Rigidbody* conseguem colidir com um *static colliders*, mas não os movem. Em casos específicos, o mecanismo de física é otimizado para *static colliders*. Por exemplo, um veículo em cima de um *Static Collider* permanece adormecido mesmo se o *Static Collider* for movido. É possível habilitar, desabilitar ou mover os *static colliders* em tempo de execução sem afetar, especialmente a velocidade de computação do mecanismo de física. Além disso, é possível dimensionar com segurança um *static Mesh Collider*, desde que a escala seja uniforme (não inclinada)(UNITYUSERMANUAL, 2020e).

2.2.1.5.2 Rigidbody Collider

Este é um *GameObject* com um *Collider* e um *Rigidbody* não *Kinematic* anexado. Os colisores de *Rigidbody* são totalmente simulados pelo mecanismo de física e podem reagir a colisões e forças aplicadas a partir de um script. Eles podem colidir com outros objetos (incluindo *static colliders*) e são a configuração de *Collider* mais comumente usada em jogos que usam física (UNITYUSERMANUAL, 2020e).

2.2.1.5.3 Kinematic Rigidbody Collider

Este é um *GameObject* com um *Collider* e um *Kinematic Rigidbody* anexado, ou seja, a propriedade *IsKinematic* do *Rigidbody* está habilitada. É possível mover um objeto *Kinematic Rigidbody* através de um *script*, modificando seu *Transform Component* 2.2.1.1, mas ele não responderá a colisões e forças como um *Rigidbody* não *Kinematic*. *Kinematic Rigidbodies* devem ser usados para *colliders* que podem ser movidos ou desabilitados/habilitados ocasionalmente, mas que, de outra forma, deveriam se comportar como um *Static Collider*2.2.1.5.1. Um exemplo disso é uma porta deslizante que normalmente deveria atuar como um obstáculo físico imóvel, mas pode ser aberta quando necessário. Ao contrário de um *Static Collider*, um *Rigidbody Kinematic* em movimento aplicará fricção a outros objetos e "acordará"outros *Rigidbody* quando eles fizerem contato. Mesmo quando imóveis, os *Kinematic Rigidbody* têm um comportamento diferente dos *Static Colliders*. Por exemplo, se o *collider* for definido como uma *Trigger* 2.2.1.6, também será necessário adicionar um *Rigidbody* a ele para que este possa receber eventos de *Trigger* no *script*. Se não é desejado que a *Trigger* caia sob a gravidade ou de outra forma seja afetado pela física, é possível definir a propriedade *IsKinematic* em seu *Rigidbody*. Um *Rigidbody Component* pode ser alternado entre o comportamento normal e cinemático a qualquer momento usando a propriedade *IsKinematic*. Um exemplo comum disso é o efeito *"ragdoll"*, em que um personagem normalmente se move sob animação, mas é fisicamente lançado por uma explosão ou colisão pesada. Cada membro do personagem pode receber seu próprio *Rigidbody Component* com *IsKinematic* habilitado por padrão. Os membros irão se mover normalmente por animação até que *IsKinematic* seja desligado para todos eles e eles imediatamente se comportem como objetos físicos. Nesse ponto, uma força de colisão ou explosão fará o personagem voar com seus membros lançados de forma convincente (UNITYUSERMANUAL, 2020e).

2.2.1.6 Triggers

O sistema de *script* pode detectar quando ocorrem colisões e iniciar ações usando a função *OnCollisionEnter*. No entanto, também é possível utilizar-se do mecanismo de física simplesmente para detectar quando um *collider* entra no espaço de outro sem criar uma colisão. Um colisor configurado como *Trigger* (usando a propriedade *IsTrigger*) não se comporta como um objeto sólido e simplesmente permitirá a passagem de outras colisões. Quando um *collider*

entra em seu espaço, uma trigger chamará a função *OnTriggerEnter* nos scripts do objeto *Trigger* (UNITYUSERMANUAL, 2020e).

2.2.1.7 Coroutines

A presente subseção foi retirada do manual da Unity disponibilizado online gratuitamente, na seção sobre *coroutines* (UNITYUSERMANUAL, 2020f). Normalmente quando se chama uma função, ela é concluída antes de retornar. Isso significa que dentro da Unity, qualquer ação que ocorrer em uma função, deve iniciar-se e encerrar-se por completo dentro de uma única atualização de quadro. Uma chamada de função não pode ser usada para conter uma animação procedural ou uma sequência de eventos ao longo do tempo. Como exemplo, consideremos a tarefa de reduzir gradualmente o valor alfa (opacidade) de um objeto até que se torne completamente invisível.

```
void Fade()
{
    for (float ft = 1f; ft >= 0; ft -= 0.1f)
    {
        Color c = renderer.material.color;
        c.a = ft;
        renderer.material.color = c;
    }
}
```

Esta função Fade, escrita da forma tradicional, não produzirá o efeito esperado na tela, porque ela executará todas as mudanças no valor alfa em uma única atualização de quadro. Para que o esmaecimento seja visível, o valor de alfa deve ser reduzido ao longo de uma sequência de quadros para mostrar os valores intermediários sendo renderizados. Os valores intermediários nunca serão vistos e o objeto desaparecerá instantaneamente. É possível lidar com situações como essa adicionando código à função Update que executa o fade quadro a quadro. No entanto, geralmente é mais conveniente usar uma coroutine para esse tipo de tarefa. Uma coroutine é como uma função que tem a capacidade de pausar a sua execução e retornar o controle a Unity, mas então continuar de onde parou no quadro seguinte. Em C#, uma coroutine é declarada assim:

```
IEnumerator Fade()
{
    for (float ft = 1f; ft >= 0; ft -= 0.1f)
    {
        Color c = renderer.material.color;
        c.a = ft;
        renderer.material.color = c;
        yield return null;
```

```
    }
}
```

Esta forma nada mais é que uma função declarada com o tipo de retorno IEnumerator e com a instrução de retorno **yield** incluída em algum lugar do corpo. A linha *"yield return null"* é o ponto em que a execução será pausada e retomada no quadro seguinte. Para definir a execução de uma coroutine, é necessário utilizar a função *StartCoroutine*:

```
void Update()
{
    if (Input.GetKeyDown("f"))
    {
        StartCoroutine("Fade");
    }
}
```

É possível notar que o contador do loop na função Fade mantém seu valor correto durante o tempo de vida da coroutine. Na verdade, qualquer variável ou parâmetro será preservado corretamente entre os yields. Por padrão, uma coroutine é retomada no quadro após ceder, mas também é possível introduzir um atraso de tempo usando *WaitForSeconds* como retorno:

```
IEnumerator Fade()
{
    for (float ft = 1f; ft >= 0; ft -= 0.1f)
    {
        Color c = renderer.material.color;
        c.a = ft;
        renderer.material.color = c;
        yield return new WaitForSeconds(.1f);
    }
}
```

Esta estratégia pode ser usada como uma forma de espalhar um efeito por um período de tempo, mas também é uma forma de otimização. Muitas tarefas em um jogo precisam ser realizadas de maneira periódica, onde nesses casos, a abordagem mais utilizada é incluí-las na função *Update*. No entanto, essa função normalmente será chamada muitas vezes por segundo. Quando uma tarefa não precisa ser repetida com tanta frequência, é possível colocá-la em uma *coroutine* para obter uma atualização regularmente, mas não em todos os quadros. Para parar uma coroutine utiliza-se os métodos *StopCoroutine* e *StopAllCoroutines*. Uma coroutine também encerra quando o *GameObject* ao qual está anexado é desabilitado com *SetActive(false)*. Chamar *Destroy(exemplo)* (onde exemplo é uma instância de *MonoBehaviour*) imediatamente aciona *OnDisable* e a *coroutine* é processada, interrompendo-a efetivamente. As *coroutines* não são interrompidas ao desabilitar um *MonoBehaviour* definindo *enabled* para false em sua instância.

2.3 Camada de Rede na Internet

O papel da camada de rede é simples, mover pacotes de um host de origem para um host de destino. Para fazer isso, duas funções importantes da camada de rede podem ser identificadas:

- Encaminhamento: Quando um pacote chega ao link de entrada de um roteador, o roteador deve mover o pacote para o link de saída apropriado. Por exemplo, um pacote que chega para um roteador R1 com origem de um host H1 e destino para um host H2, deve encaminhar o pacote para o próximo roteador no caminho para o H2.
- Roteamento: A camada de rede deve determinar a rota ou caminho percorrido pelos pacotes como eles fluem de um emissor para um receptor. Os algoritmos que calculam esses caminhos são chamados de algoritmos de roteamento. Um algoritmo de roteamento determinaria, por exemplo, o caminho ao longo do qual os pacotes fluem de H1 para H2.

O encaminhamento se refere à ação local do roteador de transferir um pacote de um link de entrada para a interface de link de saída apropriada. O roteamento refere-se a um processo em toda a rede, que determina os caminhos de ponta a ponta que os pacotes devem percorrer da origem ao destino. Cada roteador possui uma tabela de encaminhamento. Um roteador encaminha um pacote examinando o valor de um campo no cabeçalho do pacote que chega e, em seguida, utiliza este valor para o indexar na tabela de encaminhamento do roteador. O valor armazenado na entrada da tabela para esse cabeçalho indica a interface do link de saída do roteador para qual o pacote deve ser encaminhado. O algoritmo de roteamento determina os valores que são inseridos nas tabelas de encaminhamento dos roteadores. O algoritmo de roteamento pode ser centralizado (por exemplo, com um algoritmo em execução em um site central e baixando informações de roteamento para cada um dos roteadores) ou descentralizado (ou seja, com uma parte do algoritmo de roteamento distribuído executando em cada roteador). Em qualquer caso, um roteador recebe mensagens de protocolo de roteamento, que são usadas para configurar sua tabela de encaminhamento (KUROSE; ROSS, 2012).

2.3.1 Protocolos de roteamento

Dado um conjunto de roteadores, com links conectando os roteadores, um algoritmo de roteamento tem o propósito de encontrar um "bom" caminho do roteador de origem ao roteador de destino, onde normalmente o caminho considerado bom, é aquele com o menor custo (KUROSE; ROSS, 2012).

A Internet é composta por um grande número de redes independentes ou Sistemas Autônomos (AS) que são operados por diferentes organizações, geralmente uma empresa, universidade ou Provedor de Serviços de Internet (ISP) . Quando o roteamento ocorre dentro da própria rede de uma AS, este é chamado de roteamento interno, ou roteamento intra-domínio. Os primeiros protocolos de roteamento intra-domínio usavam um projeto de vetor de distância, baseado no algoritmo distribuído de Bellman-Ford herdado da ARPANET. O *Routing Information*

Protocol (RIP) é o principal exemplo usado até hoje. Apesar de funcionar bem em sistemas pequenos, o mesmo não gera bons resultados à medida que as redes vão se tornando maiores, além de possuir convergência lenta e o problema da contagem ao infinito. Por conta desses problemas, em maio de 1979 a ARPANET mudou para um protocolo de estado de enlace (TANENBAUM; WETHERALL, 2010).

2.3.1.1 OSPF

Em 1988 a *Internet Engineering Task Force* (IETF) começou a trabalhar em um protocolo de estado de enlace para roteamento intra-domínio. Esse protocolo, denominado *Open Shortest Path First* (OSPF), tornou-se um padrão em 1990 (TANENBAUM; WETHERALL, 2010).

Um protocolo de estado de enlace possui uma ideia relativamente simples, e pode ser definido em cinco passos que todos os roteadores devem realizar:

- Descobrir os seus vizinhos e aprender seus endereços de rede.
- Definir a distância ou métrica de custo para cada um de seus vizinhos.
- Construir um pacote contando tudo o que acabou de aprender.
- Enviar este pacote e receber pacotes de todos os outros roteadores.
- Calcular o caminho mais curto para todos os outros roteadores.

A topologia completa da rede é distribuída para cada roteador. Então, o algoritmo de Dijkstra 2.3.1.2 pode ser executado em cada roteador para encontrar o caminho mais curto para todos os outros roteadores. O protocolo OSPF opera abstraindo a coleção de redes, roteadores e links reais em um grafo direcionado no qual cada aresta recebe um peso (distância, atraso, etc.). Uma conexão ponto a ponto entre dois roteadores é representada por um par de arestas, uma em cada direção e seus pesos podem ser diferentes. O OSPF utiliza de um sistema de troca de informações entre roteadores adjacentes (o que não é o mesmo que entre roteadores vizinhos) chamado de *flooding*. Este sistema baseia-se em trocas de mensagens (pacotes IP), pelas quais cada roteador informa aos demais roteadores em sua área sobre seus links e o custo para outros roteadores e redes, além de permitir a consulta de estado de cada link. Essas informações permitem que cada roteador construa o grafo para sua área e calcule o caminho mais curto de si mesmo para todos os outros nós. Vários caminhos podem ser encontrados igualmente curtos. Nesse caso, o OSPF lembra o conjunto de caminhos mais curtos e, durante o encaminhamento de pacotes, o tráfego é dividido entre eles. Isso ajuda a equilibrar a carga e é denominado *Equal Cost MultiPath* (ECMP) (TANENBAUM; WETHERALL, 2010).

Durante a operação normal, cada roteador envia periodicamente mensagens Link State Update para cada um de seus roteadores adjacentes. Essas mensagens fornecem seu estado e fornecem os custos usados no banco de dados topológico. É realizada a confirmação de recebimento para as mensagens de flooding para torná-las confiáveis. Cada mensagem tem um

Tipo de Mensagem	Descrição
Hello	Usado para descobrir quem são os vizinhos
Link State Update	Fornece os custos do remetente para seus vizinhos
Link State ACK	Reconhece a atualização do estado do link
Database description	Anuncia quais atualizações o remetente tem
Link State Request	Solicita informações do parceiro

Tabela 1 – Cinco tipos de mensagens do OSPF. (TANENBAUM; WETHERALL, 2010)

número de sequência, de modo que um roteador pode ver se um Link State Update de entrada é mais antigo ou mais recente do que o que tem atualmente. Os roteadores também enviam essas mensagens quando um link aumenta ou diminui ou quando seu custo muda. As mensagens de Database Description fornecem os números de sequência de todas as entradas de estado do link atualmente mantidas pelo remetente. Comparando seus próprios valores com os do remetente, o receptor pode determinar quem tem os valores mais recentes. Essas mensagens são usadas quando um link é apresentado. Qualquer um dos parceiros pode solicitar informações de estado do link do outro usando as mensagens Link State Request. O resultado desse algoritmo é que cada par de roteadores adjacentes verifica quem possui os dados mais recentes e novas informações são espalhadas pela área dessa maneira. A Tabela 1 apresenta os cinco tipos de mensagens disponíveis pelo protocolo OSPF (TANENBAUM; WETHERALL, 2010).

2.3.1.2 Algoritmo de Dijkstra

Conforme definido em (CORMEM et al., 2009). O algoritmo de Dijkstra utiliza a técnica de Relaxamento, onde para cada vértice $v \in V$, é mantido um atributo $d.v$, chamado estimativa de caminho mais curto, que é um limite superior sobre o peso de um caminho mais curto desde a origem s até v. A inicialização das estimativas de caminhos mais curtos e os predecessores é realizada ilustrada na Fig.6. De forma que para todo v diferente de s, $d.v$ recebe valor infinito, e $d.s$ recebe valor 0.

Figura 6 – Inicialização das estimativas e predecessores

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Fonte:(CORMEM et al., 2009)

O processo de relaxamento de uma aresta (u,v) consiste em testar se podemos melhorar o caminho mais curto para v encontrando até agora pela passagem através de u, e nesse caso, atualizar $d.v$ $p.v$ 7.

Figura 7 – Algoritmo de relaxamento

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```

Fonte:(CORMEM et al., 2009)

O algoritmo de Dijkstra resolve o problema de caminhos mais curtos de fonte única em um gráfico direcionado ponderado $G = (V,E)$ para o caso em que todos os pesos das arestas são não negativos. O algoritmo de Dijkstra mantém um conjunto S de vértices cujos pesos finais de caminho mais curto partindo da origem s já foram determinados. O algoritmo seleciona repetidamente o vértice $v \in V - S$ com a menor estimativa de caminho mais curto, adiciona v ao conjunto S, e relaxa todas as arestas que saem de v. A Fig.8 ilustra uma implementação, utilizando uma fila de prioridade mínima Q de vértices, codificados por seus valores d (CORMEM et al., 2009).

Figura 8 – Algoritmo de Dijkstra

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5        u = EXTRACT-MIN(Q)
6        S = S ∪ {u}
7        for each vertex v ∈ G.Adj[u]
8              RELAX(u, v, w)
```

Fonte:(CORMEM et al., 2009)

3 ASPECTOS METODOLÓGICOS

3.1 Ambiente de Desenvolvimento

A plataforma de desenvolvimento Unity, na versão 2020.3 foi utilizada para o desenvolvimento do projeto, juntamente com a linguagem de programação C#.

3.2 Desenvolvimento de jogos com a Unity

No intuito de adquirir familiaridade com a plataforma e aprender os conceitos relacionados ao desenvolvimento de jogos digitais, foram desenvolvidos juntamente com o estudo do material teórico, uma série de pequenos projetos experimentais na plataforma aplicando os conceitos estudados. Dentre eles, listo abaixo os que acredito serem de maior importância para o desenvolvimento do jogo final, os quais estão relacionados com a movimentação de personagens.

3.2.1 Movimentação 2D

Com a finalidade de aplicar os conceitos de movimentação 2d na Unity, foi desenvolvido um jogo de duas fases. Utilizando-se de sprites disponíveis gratuitamente na Asset Store e a série de vídeos "Criando um jogo 2d de plataforma na Unity" (CRIANDO. . . , 2020) como base para este primeiro contato com a plataforma. Nesta oportunidade, foram trabalhados diversos conceitos como:

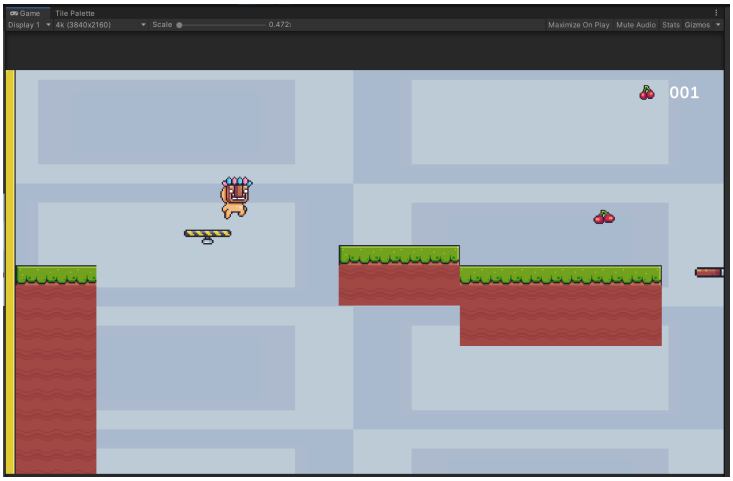
- *GameObject*
- *Hierarchy*
- *Rigidbody*
- *Colliders*
- *Triggers*

Para este jogo de plataforma 9, o objetivo do jogador era de coletar frutas pelo caminho, tantas quanto fosse possível para ganhar pontos e chegar até o fim da fase. A Fig.10 exemplifica uma das fases do jogo.

3.2.2 Movimentação 3D

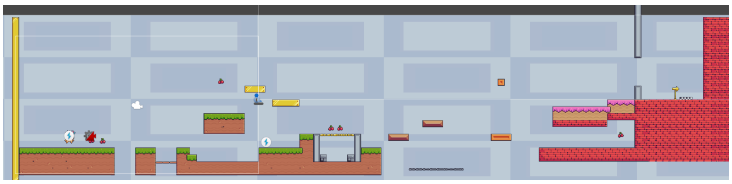
Para entender os conceitos de movimentação e criação de jogos 3d, foram desenvolvidos 2 pequenos experimentos, um seguindo o tutorial da seção learn do site da Unity, chamado "Roll a ball", e o segundo feito por autoria própria apenas para validação dos conceitos de movimentação de personagens em um ambiente 3d.

Figura 9 – Jogo de plataforma desenvolvido na Unity.



Fonte: Autoria própria.

Figura 10 – Mapa completo da fase 1 do jogo de plataforma.



Fonte: Autoria própria.

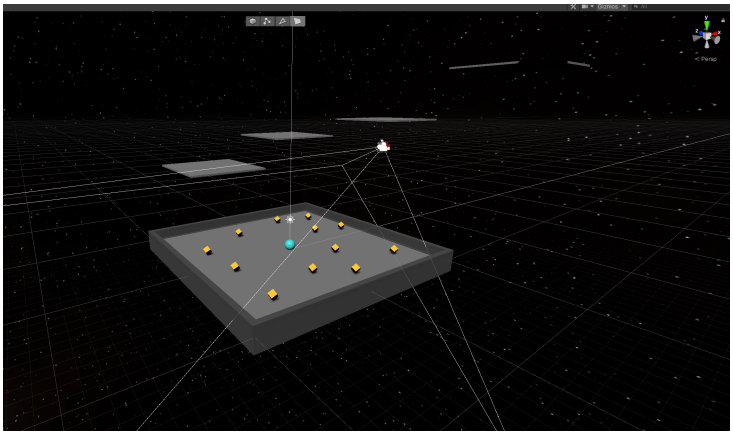
3.2.2.1 Roll A Ball

Para o desenvolvimento do jogo roll a ball, foi realizado um estudo sobre movimentação 3D simples, baseado apenas nas teclas direcionais, esquerda, direita, cima e baixo, onde a câmera observa o personagem no jogo e o segue conforme o mesmo se movimenta. Devido ao personagem não possuir orientação definida, como frente e costas, o esquema de movimentação simples foi adequado, pois não era necessário que o personagem realizasse rotações para adequar sua orientação conforme se movia. A Figura 11 ilustra uma visão panorâmica do jogo desenvolvido.

3.2.2.2 Android Movement

Para o jogo Android Movement, uma abordagem diferente para movimentação 3D foi utilizada. Foram aplicados conceitos de movimentação baseados em uma visão de terceira pessoa. Quando alguma das teclas de direção horizontal, esquerda ou direita, é pressionada, o personagem aplica uma rotação em direção ao sentido desejado, proporcional ao tempo que a

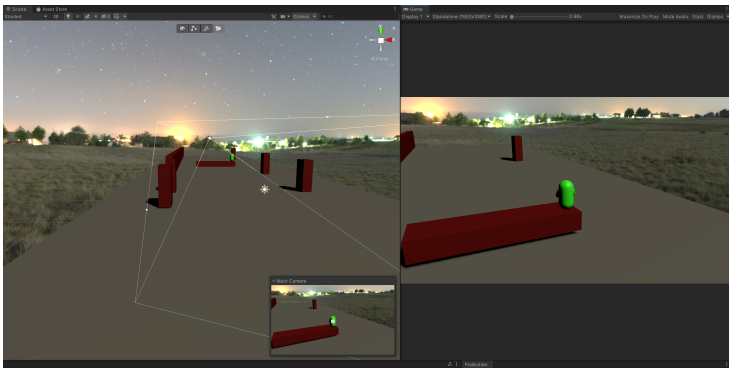
Figura 11 – Visão de cena do jogo roll a ball.



Fonte: Autoria própria.

tecla é mantida pressionada. Além disso, também foi utilizada uma técnica de movimentação, onde ao pressionar o botão para andar, o android se movimente na direção para qual a câmera está direcionada. O resultado proporciona uma movimentação mais fluida, mantendo a câmera sempre no mesmo sentido da visão do personagem. A Fig.12 ilustra a visão de cena do jogo.

Figura 12 – Visão de cena do jogo Android movement.



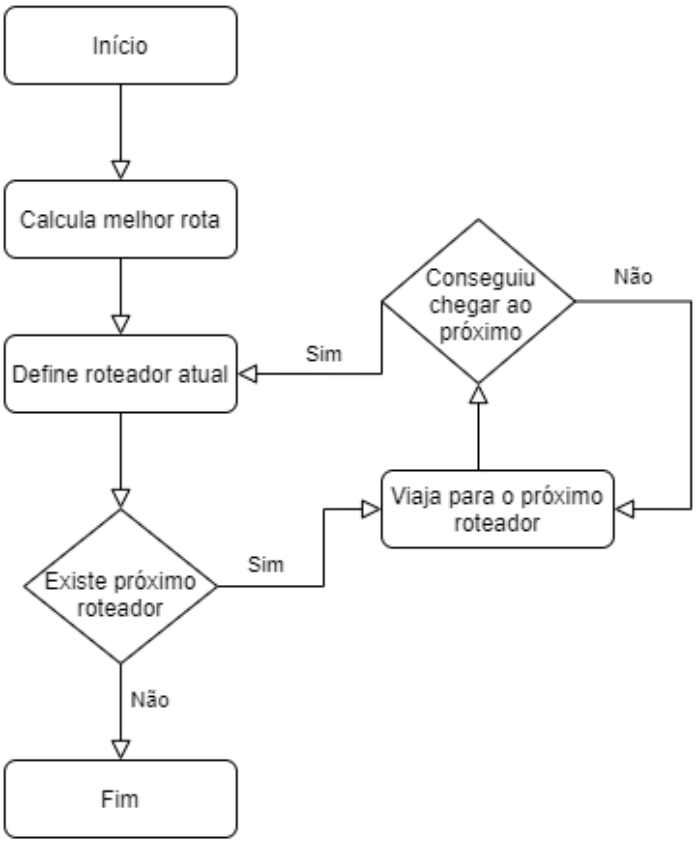
Fonte: Autoria própria.

3.3 Definição do Jogo

A principal ideia do jogo Packet Journey consiste em tentar propiciar uma abstração da viagem realizada por um pacote de dados através da rede de internet, navegando a nível da camada de rede 2.3, o jogador irá realizar sua jornada de salto em salto até o seu destino final, de forma interativa, divertida e educacional. Cada fase do jogo é constituída de duas etapas, onde na primeira etapa, o jogador no contexto do protocolo de roteamento OSPF 2.3.1.1, é

responsável por descobrir a melhor rota entre o roteador de origem e o destino. Após definida a melhor rota, o jogador agora no papel de packet, irá viajar pelo trajeto definido na primeira etapa, passando de roteador em roteador e evitando os obstáculos no caminho, representados na forma de interferências, que podem fazer com que o pacote se perca e seja necessária a retransmissão a partir do último roteador visitado. A Fig.13 ilustra o fluxo pelo qual o jogador deve percorrer para terminar uma fase.

Figura 13 – Fluxograma da resolução de uma fase.



Fonte: Autoria própria.

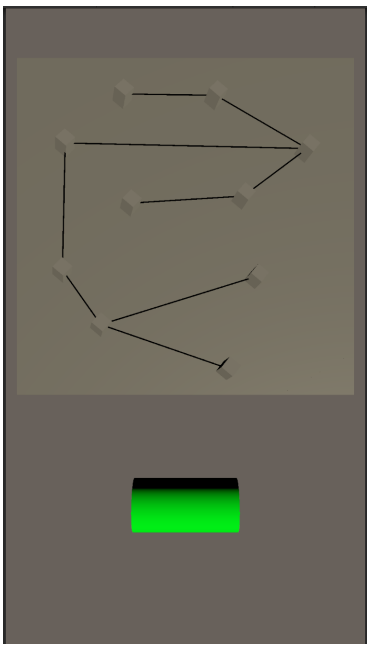
Como o objetivo do jogo é ser um jogo educacional e divertido, não existem penalidades que forcem com que o jogador tenha de recomeçar totalmente do início do jogo, podendo tentar avançar para o próximo roteador/fase quantas vezes quiser.

3.4 Prototipação

Com o intuito de aumentar a facilidade na validação de ideias e mecânicas, o jogo foi desenvolvido em forma de protótipo, para que houvesse uma maior facilidade para aplicar mudanças nos conceitos que estavam sendo aplicados, utilizando-se das formas geométricas básicas para representação visual, a ideia de prototipação também foi utilizada a nível de

código, sendo desenvolvido de forma não muito robusta a falhas, porém funcional para validar a jogabilidade, A Fig.14 exemplifica uma tela de prototipação.

Figura 14 – Exemplo de uma tela de prototipação.



Fonte: Autoria própria.

3.5 Cálculo das Melhores Rotas

Como dito anteriormente, a primeira etapa de uma fase do jogo consiste na definição das rotas pela qual o jogador, no papel de packet, posteriormente irá viajar. Cada fase inicia-se com um conjunto de roteadores dispostos na tela, onde o jogador, utilizando-se das informações contidas na tela, irá definir a melhor rota entre o roteador de origem e o roteador de destino. Por motivos de simplificação e também para que a jogabilidade do jogo não fosse prejudicada, devido ao grande número de tarefas para se realizar, a etapa de *flooding* foi suprimida, iniciando-se o jogo já com todas as distâncias entre os roteadores já dispostas na tela. Ficando a cargo do jogador a responsabilidade de determinar os melhores caminhos.

Nesta etapa o jogador possui um timer e um número máximo de erros para poder terminar esta tarefa, sendo considerado como erro, a escolha de um roteador a ser visitado de custo não mínimo ou não ter aplicado o processo de relaxamento para todos os roteadores vizinhos antes de tentar prosseguir para a próxima iteração 2.3.1.2.

3.6 Algoritmo de Dijkstra Interativo

Era desejado que o jogador pudesse atuar na resolução das melhores rotas, por isso o algoritmo de dijkstra foi adaptado para que o jogador tivesse tomadas de decisão na resolução do algoritmo. Para que fosse possível fazer com que o jogo esperasse pela tomada de decisão do jogador, avaliasse as entradas como corretas ou incorretas, e só em caso de acerto, prosseguisse com o algoritmo, foi codificada uma máquina de estados finita ref, possuindo 3 estados:

- *choosingRouterState*
- *relaxingState*
- *doNothingState*

Todos os estados possuem os métodos *EnterState, ExiteState, VerifyRelax* e *OnChoosing*. O método *EnterState*, foi utilizado para exibir uma mensagem de instrução do que se deve fazer em cada estado. O método *ExiteState*, foi utilizado para limpar as variáveis de controle do mapa a cada iteração e destruir elementos de tela pertencentes a iteração atual. Os demais métodos implementam a lógica das tarefas do usuário, de forma que o estado atual da máquina decide o que o usuário pode ou não fazer, ignorando os inputs do usuário ou os processando.

Aliada a isso, foram utilizadas as *Coroutines* 2.2.1.7, como uma forma de controlar o fluxo de execução do código. Foram adicionados ao algoritmo de Dijkstra 2.3.1.2, loops condicionados ao estado atual da máquina, de forma que só ocorresse a execução do algoritmo entre a transição dos estados, a Fig.15, ilustra o código com os pontos de interrupções baseados no estado atual da máquina.

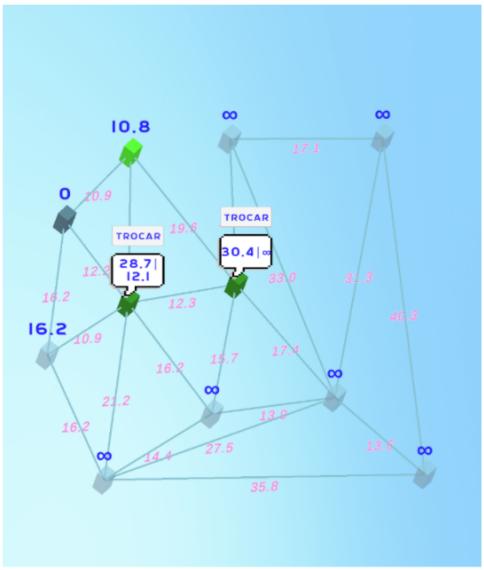
Figura 15 – Código ilustrando a utilização dos estados no algoritmo de Dijkstra.

```
public IEnumerator FindMinimunPath(Stack<Node> routerQueue) {
    OnGameStart();

    while (exploredNodes.Count != nodes.Length) {
        if (currentState == doNothingState) {
            yield break;
        }
        DrawDistances();

        unexploredNodes.Sort();
        CurrentVisiting = unexploredNodes[0];
        unexploredNodes.RemoveAt(0);
        exploredNodes.Add(CurrentVisiting);

        while (currentState == choosingRouterState) {
            yield return null;
        }
        if (currentState == doNothingState) {
            yield break;
        }

        unexploredNeighbours = unexploredNodes.FindAll(nodeW => CurrentVisiting.node.neighbourList.ContainsKey(nodeW.node));
        ColorRouters();
        colorNeihgboursRouters(unexploredNeighbours);

        foreach (var nodeW in unexploredNeighbours) {
            var distanceN = CurrentVisiting.node.neighbourList[nodeW.node];
            nodeW.newCost = CurrentVisiting.weight + distanceN;
        }

        DrawCompare(unexploredNeighbours);

        while (currentState == relaxingState) {
            yield return null;
        }
    }
```

Fonte: Autoria própria.

No caso de o usuário estourar o limite de tempo ou de erros durante a resolução, o estado “Sem processamento” é ativado para que todos os inputs do usuário sejam desativados e o programa realize as tarefas necessárias para possivelmente recomeçar esta etapa ou encerrar conforme o usuário decidir. Dessa forma a resolução dos melhores caminhos é realizada de forma semi automática, ficando a cargo do sistema realizar os cálculos de distância, e atualização dos valores para feedback, e o jogador por sua vez, fica responsável por decidir a cada rodada, qual o roteador que será visitado e para cada vizinho do roteador visitado, quais devem atualizar seus valores de custo, podendo assim, participar ativamente no algoritmo e entender seus conceitos, sem ter de se preocupar com a parte burocrática de cálculos e atualização de valores. A Fig.16 ilustra a tela do jogo na etapa de resolução dos melhores caminhos.

Figura 16 – Mapa de roteadores onde é realizada a resolução dos melhores caminhos.



Fonte: Autoria própria.

3.7 Geração Procedural de *Mesh*

Como ilustrado na Fig.13, após o trajeto ter sido definido, o jogador terá de percorrê-lo, agora no papel de packet. Para poder dar ao jogador a sensação da jornada através da rede, foi decidido que seria utilizada geração procedural de objetos para formar o caminho pelo qual o jogador percorreria. Dado que o meio de transmissão a ser representado seria via cabos, a forma geométrica Torus foi utilizada para composição do modelo 3D. Esta seção foi desenvolvida com base no tutorial *"Swirly Pipe Prototyping an Endless Racer"* (FLICK, 2020), onde é desenvolvido um jogo do tipo *endless running* (corrida sem fim).

Cada Torus é gerado proceduralmente, utilizando-se da fórmula : (3.1)

$$\begin{aligned} x(\alpha,\beta) &= (R + rcos\beta)cos\alpha \\ y(\alpha,\beta) &= (R + rcos\beta)sin\alpha \\ z(\alpha,\beta) &= rsin\beta \end{aligned} \tag{3.1}$$

Onde 'R' corresponde ao raio maior, ou raio da curva, 'r' corresponde ao raio menor, ou raio do torus, β corresponde ao ângulo no torus, e α ao ângulo na curva.

Para geração do mesh do modelo utilizando a fórmula dada, o torus é desenhado como uma composição de anéis, desenhados a uma mesma distância entre si, sobre a curva do Torus. Algumas variáveis de controles foram definidas para que fosse possível alterar a forma com a qual o torus é desenhado, que são:

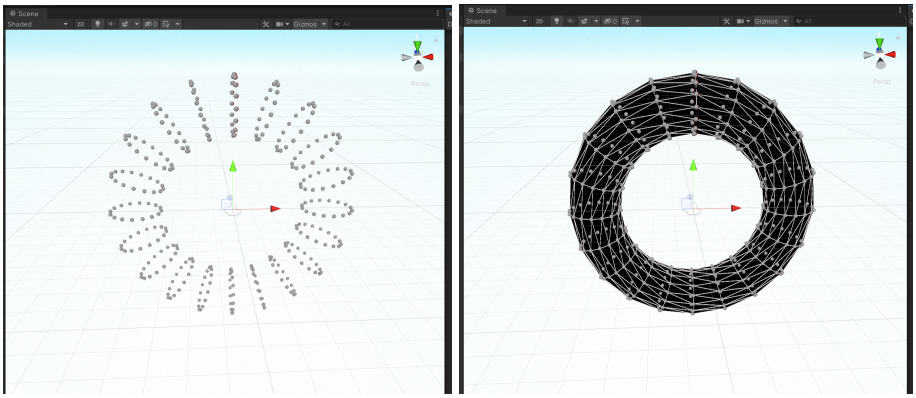
- Raio da Curva
- Raio do Torus
- Número de pontos de cada anel
- Número de aneis

Desta forma é possível alterar o formato do torus para que este tenha um formato mais suave ou com mais arestas. Tendo definido os valores das variáveis de controle, os anéis vão sendo desenhados em sentido horário, gerando um torus completo. A Fig.17 ilustra um *mesh* gerado.

Dado que foi criado *mesh* de um torus completo, é possível diminuir a distância entre os anéis do torus, para criar um segmento de torus. Os segmentos de torus então são alinhados uns após os outros, a partir do último ângulo do segmento anterior, ilustrado em vermelho na Fig.18, para que possa se formar um tipo de tubo curvado, que neste caso, representaria o cabo de transmissão pelo qual o jogador irá correr pelo seu interior.

Variando-se o raio maior e menor, e a direção de cada segmento de torus a ser gerado, é possível criar um caminho com inclinações diversificadas, de comprimento tão longo quanto seja necessário. Para a disposição dos segmentos, as variações nos raios e direção foram colocadas

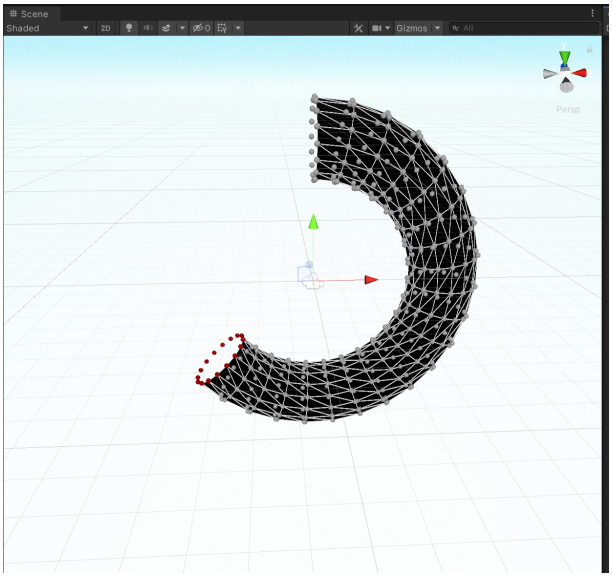
Figura 17 – Mesh gerado com raio de curva 5, raio de torus 1, 20 anéis e 16 pontos por ane.



(a) Modelo renderizando gizmos. (b) Modelo renderizando o *Mesh*.

Fonte: Autoria própria.

Figura 18 – Segmento de um Torus, com o último ângulo gerado destacado em vermelho.

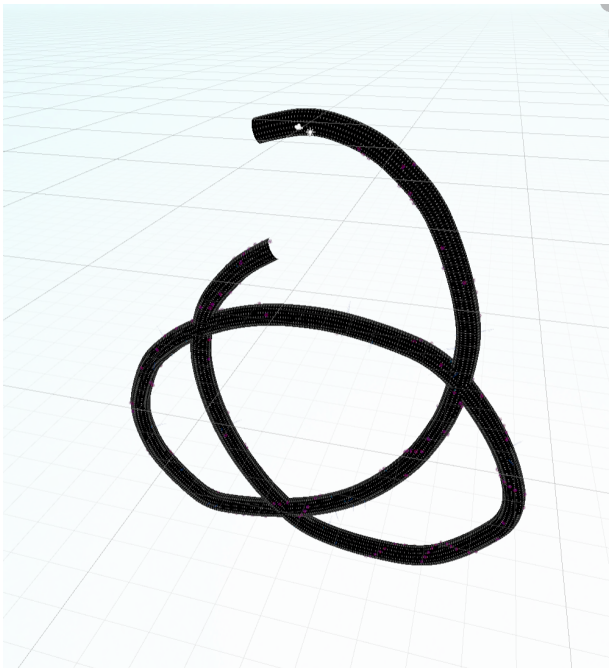


Fonte: Autoria própria.

de forma aleatória, variando dentro de um range, que poderia ser escolhido através da Unity, podendo assim, serem ajustados conforme fosse desejado para melhorar a sensação de viagem dentro dos segmentos. Mesmo assim, como é ilustrado na Fig.19, ao se escolher um número relativamente grande de segmentos para compor o caminho, começou-se a obter um problema visual causado pela possível intersecção dos segmentos, gerando a impressão de uma parede dentro do caminho do jogador.

Visando a economia de recursos e também para resolver o problema da intersecção entre os segmentos, foi utilizado um número limitado de segmentos para composição do caminho. Para que fosse mantida a sensação de um caminho completo, o sistema foi codificado de tal

Figura 19 – Caminho gerado por 70 segmentos de Torus alinhados um após o outro, onde ocorreu uma intersecção.



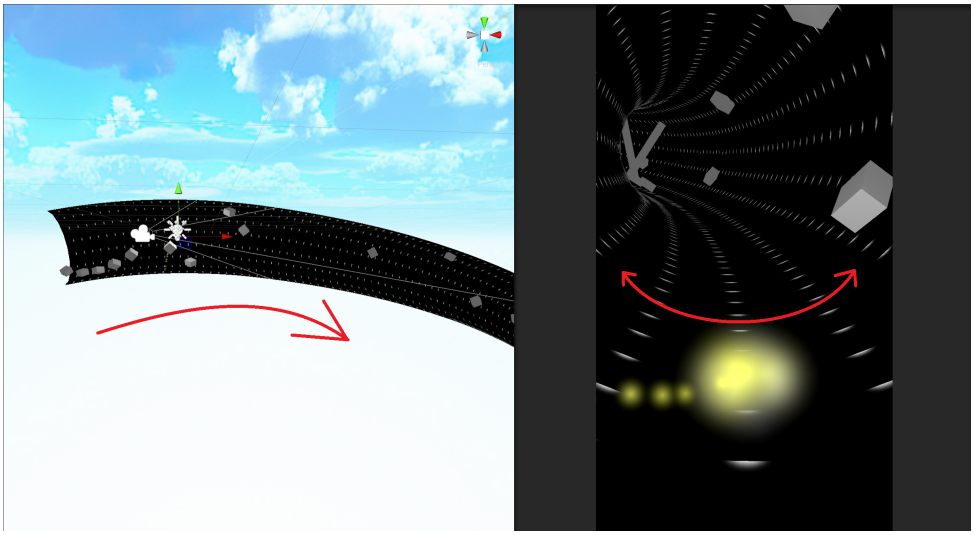
Fonte: Autoria própria.

forma que conforme o jogador passasse completamente por toda extensão de um dado segmento, o mesmo fosse reposicionado para o final do caminho, gerando assim um caminho infinito.

3.8 Movimentação

Assim como no tutorial base, foi decidido utilizar a ideia de movimentação do mundo ao invés do personagem. Este tipo de movimentação, semelhante ao efeito parallax no mundo 2D, baseia-se na ideia de manter o personagem em repouso enquanto o mundo e os objetos se movimentam em direção ao personagem do jogador. No caso deste trabalho, o personagem foi posicionado em direção ao eixo x, e foi aplicada uma rotação em torno do eixo z no sistema de segmentos de torus, gerando assim a ilusão de movimento por dentro do modelo 3D do cabo de transmissão. Para a movimentação do jogador, como o mesmo já se movimenta para frente automaticamente, e a movimentação para trás não é permitida, somente foi realizada a movimentação no sentido horizontal, de forma que quando o jogador pressione um botão para se movimentar para esquerda ou direita, é aplicada uma rotação local, restrita ao raio menor do segmento de torus no qual o jogador está passando, evitando que o mesmo saia de dentro do torus e permitindo assim que o jogador desvie dos obstáculos no caminho. A Fig.20 ilustra os sentidos de movimentação.

Figura 20 – Sentido da movimentação do personagem.



Fonte: Autoria própria.

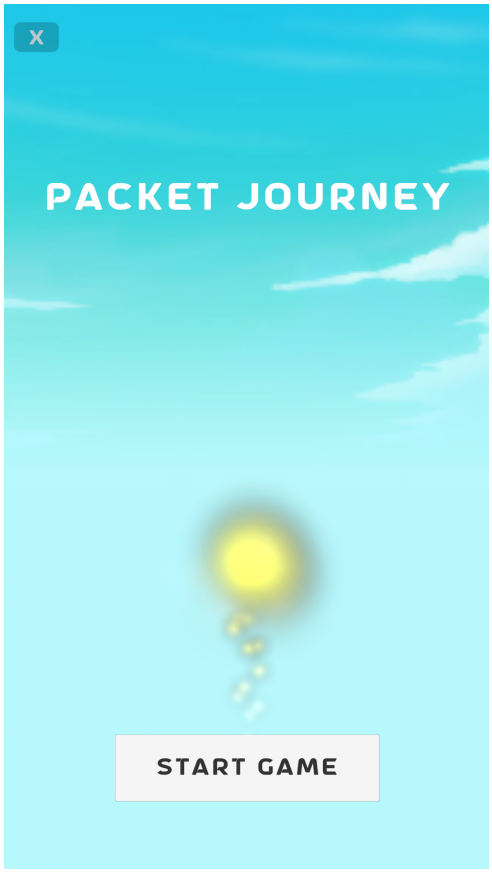
Como o jogador poderia percorrer pelo caminho indefinidamente, foi utilizada a distância euclidiana entre os roteadores do segmento atual no mapa, multiplicada por uma constante, como a definição da distância que o jogador terá que percorrer para chegar ao próximo roteador.

4 RESULTADOS

4.1 Elementos de Tela

A tela inicial do jogo é composta por um fundo de tela, com uma pequena animação do player e um botão para iniciar o jogo. A Fig.21 ilustra o cenário descrito.

Figura 21 – Tela inicial do Packet Journey.



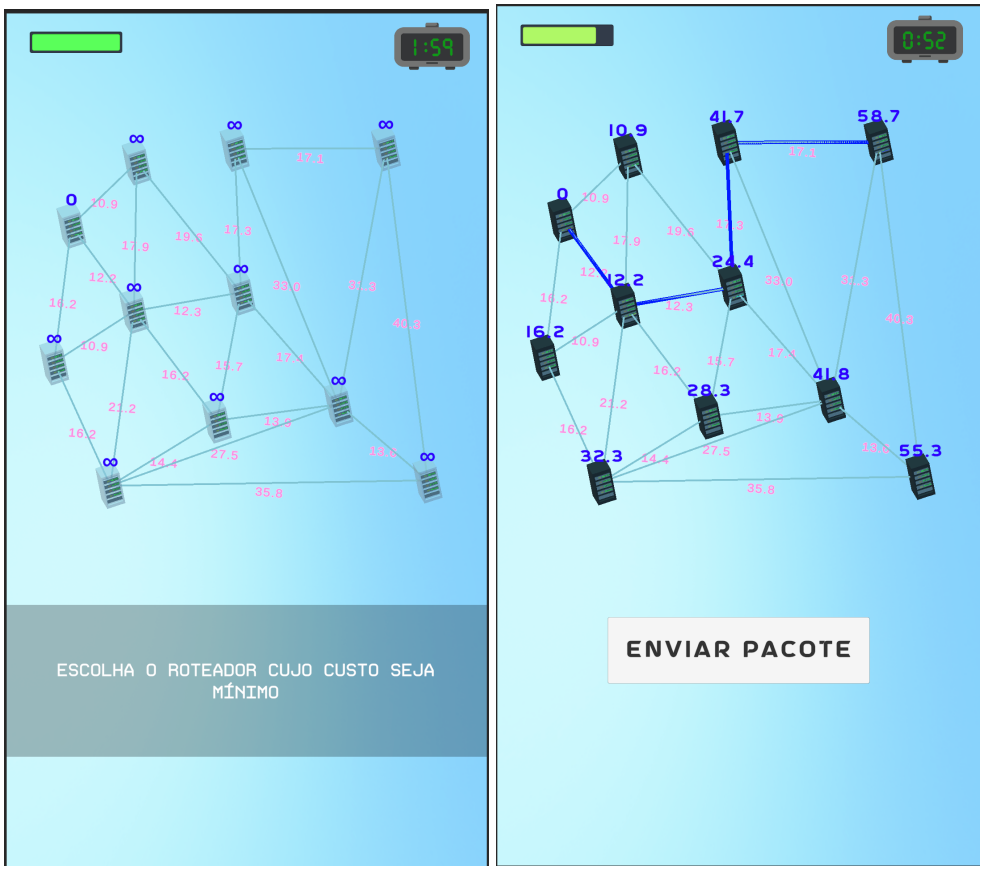
Fonte: Autoria própria.

Ao iniciar-se o jogo, são apresentadas várias telas contendo informações introdutórias sobre o contexto do jogo e o objetivo,estas podem ser consultadas no apêndice **Telas de Instrução**.

4.1.1 Mapa de Roteadores

Após as instruções, o jogador é levado para a tela do mapa de roteadores, onde o mesmo irá realizar o calculo das melhores rotas entre os roteadores, e assim determinar o trajeto pelo qual irá viajar posteriormente, Esta tela possui o grafo dos roteadores, um campo de mensagens de instrução, um timer e uma barra de vida para representar os erros restantes do usuário. A Fig.22 ilustra esta tela.

Figura 22 – Tela de resolução dos melhores caminhos.



(a) Mapa de roteadores antes da definição do trajeto.

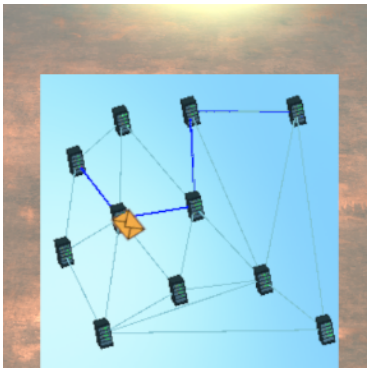
(b) Mapa de roteadores depois da definição do trajeto.

Fonte: Autoria própria.

4.1.2 Mini-mapa

Com o intuito de dar ao jogador informações sobre o trajeto pelo qual ele está percorrendo, como por exemplo qual o trajeto atual e distância até o próximo roteador, foi adicionada na tela de jogo da etapa de viagem pela rede, um mini-mapa que apresenta o mapa de roteadores, e um ícone representando o jogador, que se move em direção ao próximo roteador conforme o usuário se move pelo cabo de transmissão. A Fig.23 ilustra a tela mencionada.

Figura 23 – Mini-mapa indicando a posição do jogador no trajeto

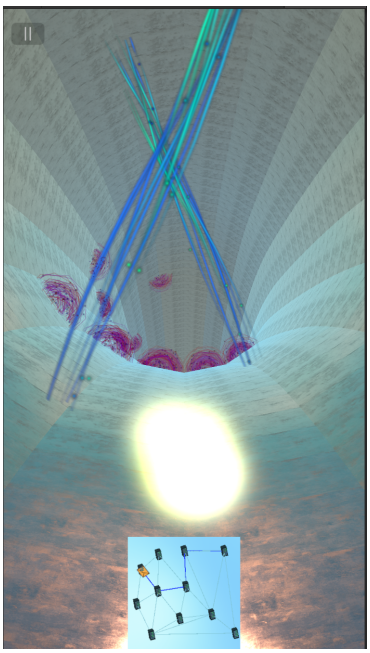


Fonte: Autoria própria.

4.1.3 Trajeto entre os roteadores

Dentro do cabo de transmissão foram adicionados obstáculos para que o jogador pudesse desviar-se. Os obstáculos são de dois tipos, o pequeno e o transversal, representados por sistemas de partículas para ilustrar interferências eletromagnéticas dentro do cabo. Ao tocar em algum dos obstáculos, o jogador terá de reiniciar o trajeto atual a partir do último roteador alcançado. A Fig.24 ilustra os dois tipos de obstáculos encontrados no jogo.

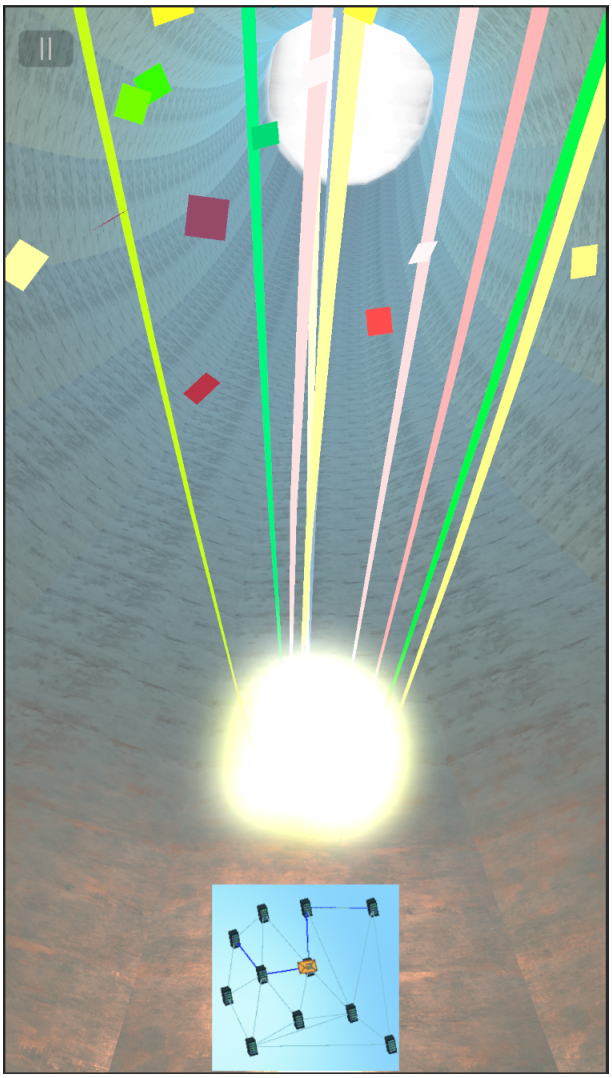
Figura 24 – Obstáculos no trajeto



Fonte: Autoria própria.

Quando o jogador percorre uma distância equivalente a distância entres os roteadores do trajeto atual, os obstáculos são retirados, a velocidade do jogador é reduzida e um sistema de partículas simulando confetes é disparado como forma de feedback ao usuário, logo em seguida o personagem atinge um objeto invisível que possui uma trigger que dispara o processo de encerramento desta etapa do jogo e retorna para a tela do mapa de roteadores, onde o jogador poderá iniciar a viagem para o próximo roteador. A Fig.25 ilustra o momento de chegada a um roteador.

Figura 25 – Liberação de confetes ao percorrer a distância necessária até o próximo roteador



Fonte: Autoria própria.

5 CONSIDERAÇÕES FINAIS

Uma preocupação constante durante o desenvolvimento deste trabalho foi a de conseguir fazer com o que jogo conseguisse manter o interesse do usuário. Visto que realizar as atividades de um protocolo de qualquer camada da internet, acaba tornando-se um processo maçante e cansativo. Por este motivo, as decisões de quando e onde o jogador deveria participar não foram triviais.

A utilização de elementos como um contador de erros e timer, elementos utilizados em vários jogos desde os primórdios do jogos, foi uma decisão tomada já em fase final de desenvolvimento, com o intuito de trazer um certo sentimento de urgência e responsabilidade com cada ação que o usuário toma, fazendo assim com que ele tenha que pensar sobre a atividade que está realizando antes de realizá-la.

A união das mecânicas utilizadas na resolução das melhores rotas em conjunto com o a mecânica de jogos do tipo *endless runner*, mostrou-se uma boa abordagem para manter o engajamento do jogador, pois mesmo ambos os cenários possuindo dificuldades para serem superadas, a alternância entre essas dificuldades traz mais dinamismo de forma geral.

Ainda existe muita abertura para aprimoramento e expansão deste jogo. Até o fim deste trabalho, o mesmo possui apenas dois mapas, mas como um trabalho futuro, poderia-se trabalhar no desenvolvimento de um módulo de geração de mapas de forma procedural. Dado que o jogo é do estilo casual, o mesmo não teria necessariamente que ter um fim.

Também para trabalhos futuros, o desenvolvimento de novas mecânicas para abordar a troca de mensagens do OSPF, ou até mesmo a introdução de mecânicas para protocolos de outras camadas também poderiam ser desenvolvidas, trazendo assim, um conhecimento ainda mais rico sobre redes de computadores.

Com isso, apesar de não representar a implementação final do jogo, esse trabalho mostra que é possível experimentar e reutilizar de mecânicas de jogos já conhecidas do mercado em contextos de ensino, mesmo que mais complexos como redes de computadores, de forma que ainda seja agradável e produtivo para o jogador.

REFERÊNCIAS

CORMEM, T. H. et al. **Introduction to Algorithms**. 3rd. ed. London, England: MIT Press, 2009. 648 - 658 p. Citado 2 vezes nas páginas 24 e 25.

CRIANDO UM JOGO 2D DE PLATAFORMA NA UNITY. **Crie Seus Jogos**, 2020. Disponível em: <https://www.youtube.com/watch?v=Vt7VtkWb3R4>. Acesso em: 20 ago. 2020. Citado na página 26.

FLICK, J. Prototyping an endless racer. **Catlike Coding**, 2020. Disponível em: <https://catlikecoding.com/unity/tutorials/swirly-pipe/>. Acesso em: 15 set. 2020. Citado na página 33.

GOLDSTONE, W. **Unity game development essentials**. Birmingham, England: Packt Publishing, 2010. 1 - 2 p. Citado na página 14.

GREGORY, J. Game engine architecture. **CRC Press**, Boca Raton, FL, p. 8 – 9, 2009. Citado na página 13.

GROS, B. The impact of digital games in education. **First Monday**, v. 8, n. 7, p. 8 – 9, Jul 2003. Disponível em: <https://www.mackenty.org/images/uploads/impact_of_games_in_education.pdf>. Acesso em: 17 out. 2020. Citado na página 11.

HAAS, J. K. A history of the unity game engine. **Thesis, Worcester Polytechnic Institute**, 2014. Disponível em: <https://web.wpi.edu/Pubs/Eproject/Available/E-project-030614-143124/unrestricted/Haas_IQP_Final.pdf>. Acesso em: 15 ago. 2020. Citado na página 13.

HSIAO, H. A brief review of digital games and learning. **IEEE International Workshop on Digital Games and Intelligent Toys-based Education**, Jhongli City, p. 124 – 129, 2007. Citado na página 11.

KUROSE, J. F.; ROSS, K. W. **Computer networking: A top-down approach**. 6th. ed. Upper Saddle River, NJ: Pearson, 2012. 305 - 389 p. Citado na página 22.

SILVA, G. M. O uso do computador na educação, aliada a softwares educativos no auxílio ao ensino e aprendizagem. **Educação Pública**, 2008. Disponível em: <https://educacaopublica.cecierj.edu.br/artigos/8/9/o-uso-do-computador-na-educaccedilatildeo-aliada-a-softwares-educativos-no-auxiacutelio-ao-ensino-e-apr Acesso em: 17 ago. 2020. Citado na página 11.

TANENBAUM, A. S.; WETHERALL, D. J. **Computer Networks**. 5th. ed. Upper Saddle River, NJ: Pearson, 2010. 362 - 479 p. Citado 2 vezes nas páginas 23 e 24.

TAROUCO, L. M. R. et al. Jogos educacionais. **Revista Novas Tecnologias na Educação**, Porto Alegre, v. 2, n. 1, MAR 2004. Citado na página 11.

UNITYUSERMANUAL. Gameobjects. 2020a. Disponível em: <https://docs.unity3d.com/Manual/GameObjects.html>. Acesso em: 12 out. 2020. Citado 2 vezes nas páginas 15 e 16.

UNITYUSERMANUAL. The hierarchy window. 2020b. Disponível em: <https://docs.unity3d.com/Manual/Hierarchy.htm>. Acesso em: 12 out. 2020. Citado na página 16.

UNITYUSERMANUAL. Meshes. 2020c. Disponível em: <https://docs.unity3d.com/Manual/class-Mesh.html>. Acesso em: 12 out. 2020. Citado na página 18.

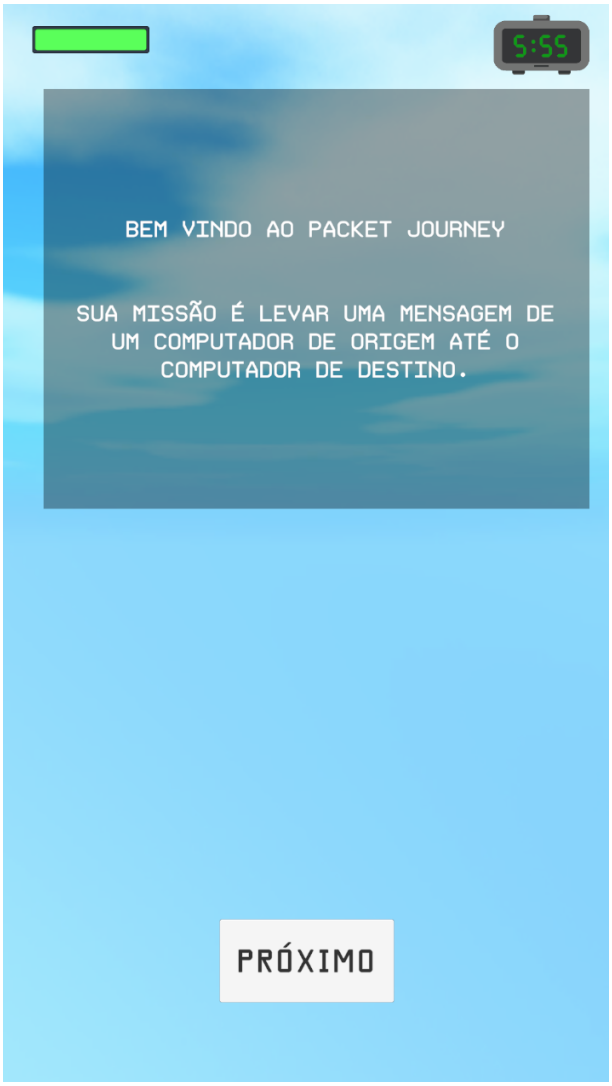
UNITYUSERMANUAL. Rigidbody. 2020d. Disponível em: <https://docs.unity3d.com/Manual/class-Rigidbody.html>. Acesso em: 12 out. 2020. Citado na página 18.

UNITYUSERMANUAL. Colliders. 2020e. Disponível em: <https://docs.unity3d.com/Manual/CollidersOverview.html>. Acesso em: 12 out. 2020. Citado 3 vezes nas páginas 18, 19 e 20.

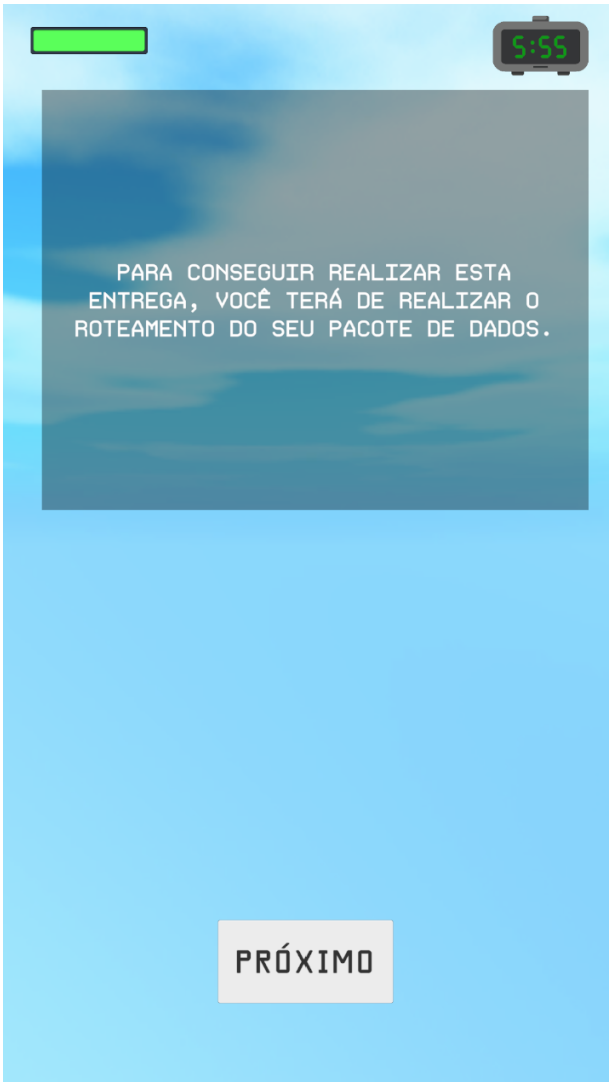
UNITYUSERMANUAL. Coroutines. 2020f. Disponível em: <https://docs.unity3d.com/Manual/Coroutines.html>. Acesso em: 12 out. 2020. Citado na página 20.

WIKIPEDIA. Unity (game engine). Wikipedia, The Free Encyclopedia, 2020. Disponível em: <https://en.wikipedia.org/w/index.php?title=Unity_(game_engine)&oldid=988625322>. Acesso em: 12 out. 2020. Citado na página 14.

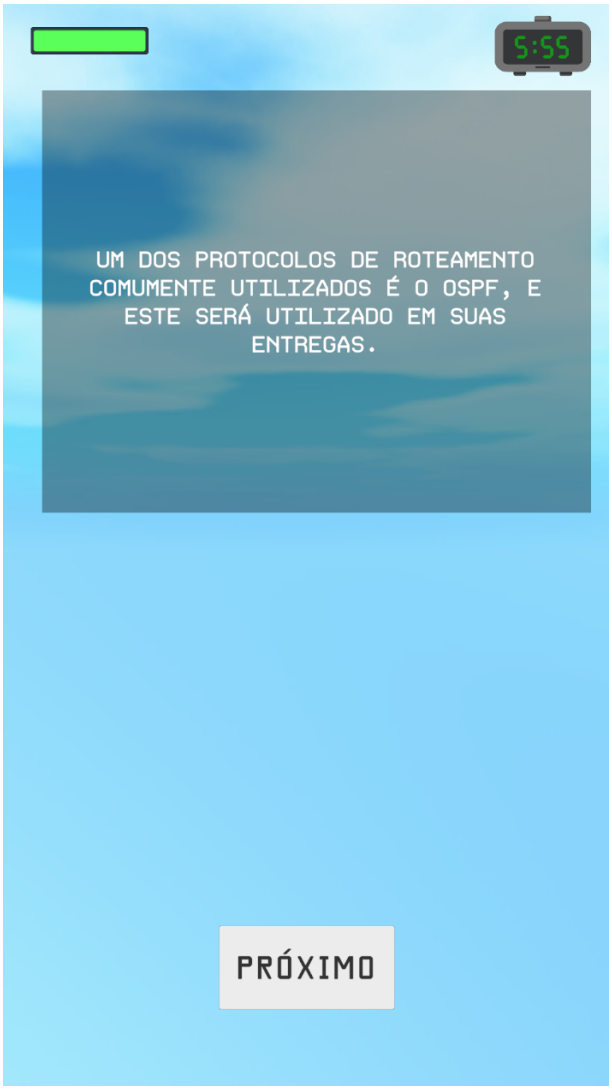
A TELAS DE INSTRUÇÃO



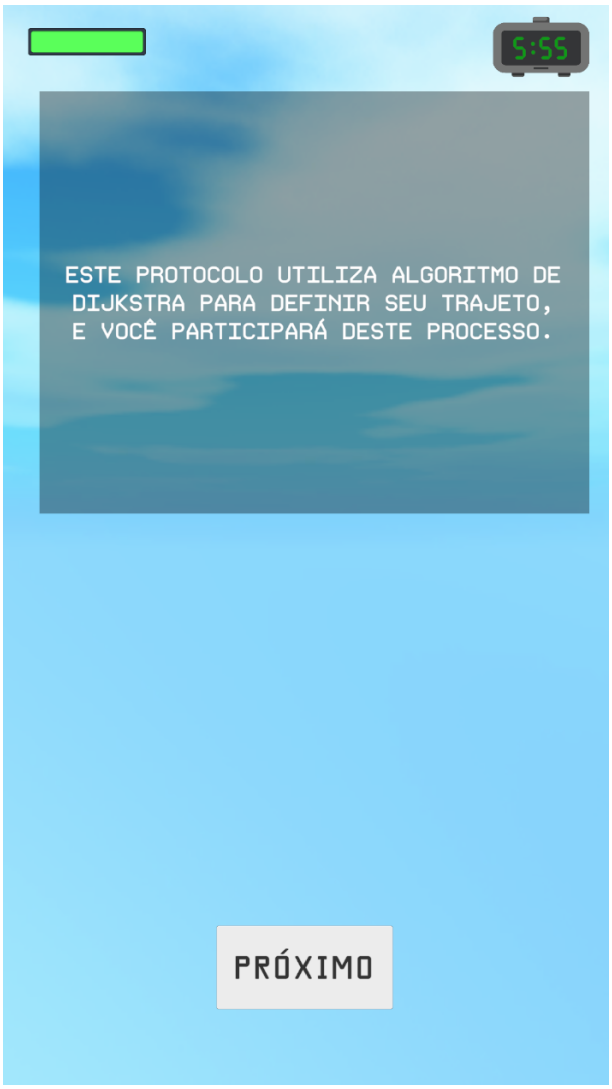
Fonte: Autoria própria.



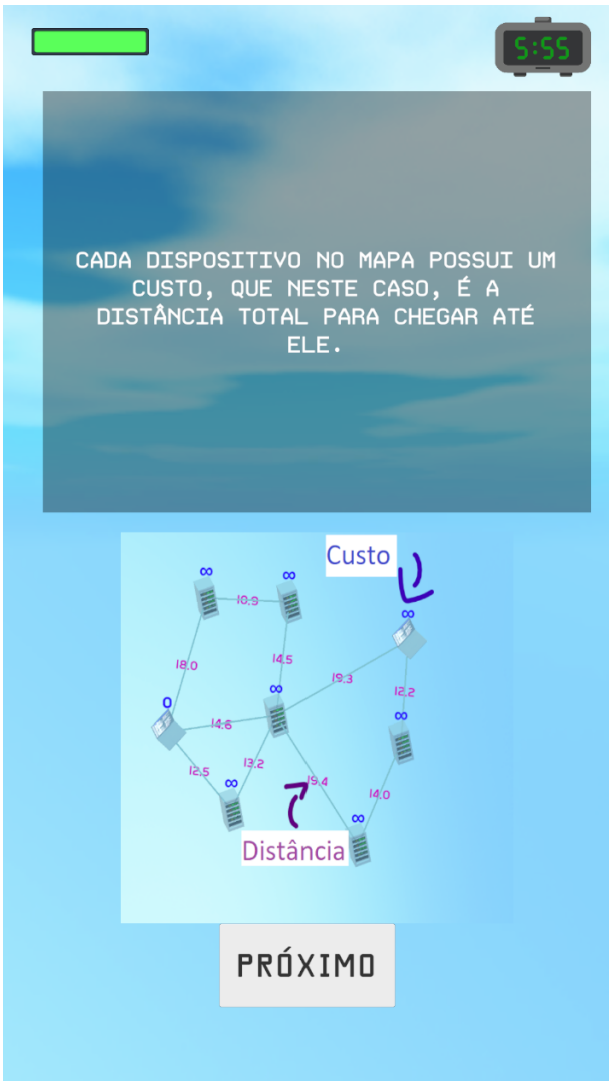
Fonte: Autoria própria.



Fonte: Autoria própria.



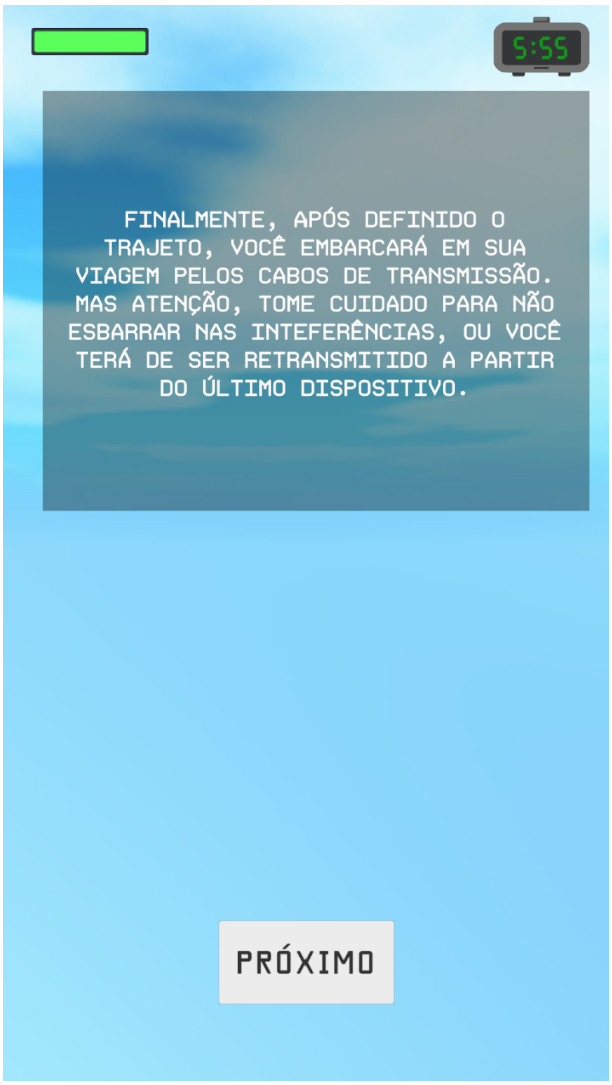
Fonte: Autoria própria.



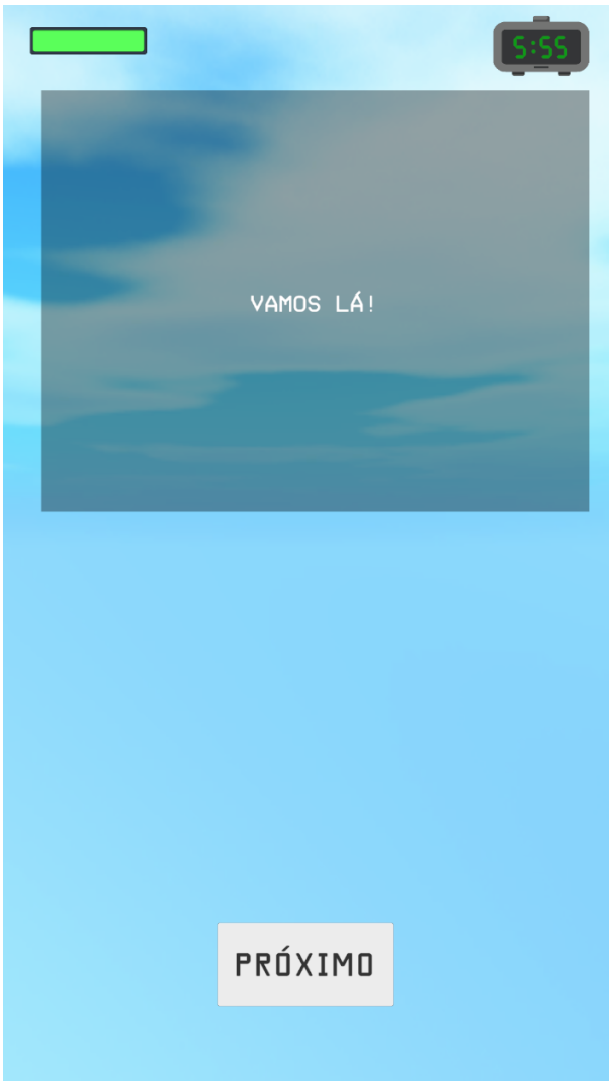
Fonte: Autoria própria.

Fonte: Autoria própria.

Fonte: Autoria própria.



Fonte: Autoria própria.



Fonte: Autoria própria.