ESCOLA POLITÉCNICA

Ludomusicalidade para Imersão em Jogos Virtuais

Orientador: Prof. Dr. Ricardo Nakamura

Engenharia da Computação - Escola Politécnica da USP
Gabriel Yugo Nascimento Kishida **11257647**

São Paulo
Dezembro de 2024

Sumário

Agradecimentos & Dedicatória

Deixo meus agradecimentos ao Prof. Dr. Ricardo Nakamura, o orientador deste projeto. Suas referências e seus auxílios em nossos encontros semanais desde o começo do projeto até o fim foram essenciais para, bem, tudo. Sua paciência e presença ao longo do ano foram a base sólida deste projeto.

Também quero deixar agradecimentos à POLI-USP, por me permitir seguir com este trabalho, que é composto de minha dedicação, ofício e paixão. Sem tudo o que aprendi durante este curso, o projeto não seria possível.

Além disso, agradeço à minha família, aos amigos e colegas que não só me forneceram comentários e *feedbacks* sobre o projeto, mas também me apoiaram com palavras de carinho e de incentivo. É por causa destes que consigo dedicar toda a energia que encontro.

Dedico este trabalho a todos os desenvolvedores de jogos do mundo. Apesar de ser uma indústria crescente, são poucas as pessoas que recebem o reconhecimento que merecem. Falo dos desenvolvedores, artistas e roteiristas, que muitas vezes não podem tomar decisões sobre o próprio produto que criam. Antes de ser uma indústria, é um ofício de Arte, e suas contribuições já definem e vão definir a cultura das próximas décadas. Desejo ver estes Artistas livres. Almejo ser um deles.

> *"To make something well is to give yourself to it, to seek wholeness, to follow spirit. To learn to make something well can take your whole life. It's worth it."*

- Ursula K. Le Guin

1 Introdução

1.1 Motivação

A indústria de jogos tem desempenhado um papel crucial na sociedade contemporânea, tanto como uma forma de entretenimento quanto uma plataforma para a expressão artística e cultural. Com a evolução tecnológica e a crescente demanda por experiências cada vez mais imersivas, os desenvolvedores de jogos têm explorado diversas formas de aprimorar a interação entre o jogador e o ambiente virtual. Um aspecto que se destaca neste meio é a integração da música como elemento essencial na experiência de jogo.

Jogos como *Guitar Hero* (1), *Taiko no Tatsujin* (2) e *Just Dance* (3) incorporam diferentes mecânicas de sincronia com a música – seja por imitar o funcionamento de um instrumento musical ou incentivar a dança.

Por outro lado, também existem jogos que utilizam da música de maneira indireta e criam mecânicas cuja temática principal é fantasiosa, e não uma abstração de uma atividade humana real e relacionada diretamente à música. Incorporam-se mecânicas musicais implícitas, reagindo a situações encontradas pelo usuário, adicionando pequenos trechos de sons para indicar avanços ou sucessos, tanto em combate quanto em desafios – como é o caso da série *The Legend of Zelda* (4), que insere deixas musicais para diferentes eventos do jogo (como abrir uma porta, superar um desafio, etc.).

No meio termo destas aplicações implícitas e explícitas, encontram-se jogos como *Crypt of the Necrodancer* que não abrem mão das influências diretas da música nas mecânicas do jogo, nem do aspecto fantasioso que abstrai a música da temática principal. Trata-se de um jogo cuja mecânica principal gira em torno de seguir um ritmo (fortemente denotado pela trilha sonora) para utilizar comandos como se mover ou atacar inimigos – tudo isso enquanto conta-se uma história de uma aventureira que adentra a caverna de um necromante.

Figura 1: Imagem do gameplay de Crypt of The Necrodancer. A personagem (no centro) deve movimentar-se somente durante certos pontos do ritmo (denotado pelas barras em torno do coração na parte inferior da tela).

Como diz a compositora Winifred Phillips em seu livro *A Composer's Guide to Game Music* (5): "A música é um elemento de jogabilidade como qualquer outro. Ela tem o poder de guiar os jogadores, fornecer feedback e melhorar a imersão ao conectar diretamente as emoções à experiência.". A Ludomusicalidade – palavra que une conceitos lúdicos de mecânicas na jogabilidade com musicalidade – busca dar o próximo passo, colocando a música não só como um elemento que fornece feedback ao jogador, mas como uma entidade ativa que rege as regras do jogo. O intuito deste projeto é estudar este caso de um ponto de vista de engenharia: como a aplicação destas mecânicas pode ser facilitada tanto para o compositor quanto para o desenvolvedor. Desta forma, busca-se aplicar conceitos de *Engenharia de Software* para desenvolver estruturas e arquiteturas úteis no meio de ***Game Development***. Para isso, pretende-se analisar jogos existentes e como são realizadas as mecânicas de sincronia da música e jogabilidade.

1.2 Objetivo

Este Projeto de Formatura consiste no desenvolvimento de uma **Protótipo de um jogo eletrônico** para o computador (futuramente transponível para outras plataformas)

cuja mecânica principal gira em torno da **sincronização dos comandos** do jogador com a **música do jogo**.

O objetivo deste projeto é explorar como **maneiras inovadoras** de **mecânicas de interação sincronizadas com música** podem gerar um valor de entretenimento diferenciado em uma indústria tão recente e inovadora quanto a de jogos eletrônicos. Além disso, almeja-se aplicar conceitos de engenharia e Gestão de Software para garantir que o projeto dê fruto a um produto de qualidade e que gere valor. Por fim, pretende-se adquirir e documentar o conhecimento do *engine* **Unity**, de forma a torná-lo mais acessível ao meio acadêmico.

1.3 Justificativa

O estudo e a aplicação da ludomusicalidade em jogos eletrônicos é interessante devido uma gama de razões:

- **Ampliação da expressividade:** como jogos eletrônicos são importantes formas de expressão e manifestação cultural, explorar e estudar diferentes formas inovadoras pode fornecer novas ferramentas para expressividade – não só para os desenvolvedores, mas também para os compositores musicais – neste meio que está em constante evolução tecnológica e artística;

- **Experiências inclusivas em jogos**: atualmente, jogos eletrônicos são intensamente dependentes do aspecto visual – desta forma, pessoas com deficiências visuais podem se sentir deixadas de lado desta indústria e deste movimento cultural. Aplicar mecâncias que utilizam os sons e música mais do que aspectos visuais pode prover aos jogadores com deficiências visuais uma experiência mais imersiva e inclusiva que outros jogos do mercado;

- **Aplicação de técnicas de engenharia para a produção de jogos**: devido ao tamanho do mercado consumidor e desenvolvedor, a indústria de jogos se destaca por ter muitos guias e tutoriais na mídia, tornando-se fácil aprender o desenvolvimento. No entanto, no que se deve ao nível amador até o nível profissional de estúdios de pequeno e médio porte, o desenvolvimento não aplica técnicas de engenharia, como validações, estruturações e arquitetura – gerando assim, diversos projetos cuja complexidade cresce exponencialmente e cujo retrabalho torna-se dificultado. Busca-se com este projeto aplicar conceitos de engenharia neste meio tão

inovador e tão importante, de forma a definir uma estrutura mais sólida e concreta para o desenvolvimento.

- **Técnicas para ampliação da imersão e entretenimento**: como objetivo final e principal, o desenvolvimento de jogos busca gerar o entretenimento de seu consumidor. O estudo de diferentes técnicas de ludomusicalidade serve, em última instância, para tornar jogos eletrônicos mais divertidos e imersivos.

2 Aspectos Conceituais

No que se deve ao desenvolvimento de um jogo de videogame, é necessário basear-se em conceitos de engenharia já bem estabelecidos entre os desenvolvedores, tais como: Design Patterns (de *software*) para jogos e métodos ágeis para desenvolvimento de projeto. Já que o projeto engloba o desenvolvimento de um protótipo jogável, é também importante estudar conceitos de *Game Design* e *Level Design*, que são úteis para proporcionar experiências de alta qualidade para jogadores.

Portanto, para fins de execução do projeto, foram feitas pesquisas relativas a estes tópicos e, neste capítulo, serão discutidos e destacados aspectos que serão fundamentais para a produção deste trabalho.

2.1 Design Patterns para jogos

Embora o desenvolvimento de código seja um campo comparativamente novo na tecnologia, existem vertentes de pensamento que regem metodologias de programação. A metodologia usada neste projeto é uma das mais famosas: *Clean Code* de *Robert C. Martin* (6). Deste livro, o conceito mais utilizado são os princípios SOLID, que buscam guiar programadores a criar código com alta manutenibilidade, flexível a diferentes propósitos e mais legível.

No que se deve à ideia de aplicar Design Patterns em qualquer projeto de Software é buscar resolver problemas comuns de maneiras padronizadas, de forma a garantir os princípios SOLID. Além disso, os Design Patterns são métodos já estabelecidos por uma comunidade de programadores e que, consequentemente, já foi aprimorado iterativamente por diversas pessoas.

No caso deste projeto, buscam-se resolver dois principais problemas: como conectar diversos (e diferentes) componentes do jogo de forma a não quebrar o princípio de respon-

sabilidade única e como estruturar comportamentos complexos sem ser muito convoluto. Para o primeiro, foi utilizado o padrão *Observer*, já para o segundo, o padrão de *Finite State Machine* ou *FSM*.

Padrão - Observer

O padrão *Observer* é utilizado com o objetivo de resolver a interação de diversos componentes, sem necessitar estabelecer uma relação na classe de cada.

Um exemplo real do projeto é quando, por exemplo, um inimigo é derrotado, uma série de eventos pode ser desencadeada: a câmera pode mudar de ângulo, uma porta pode se abrir, outro inimigo pode mudar seu comportamento, etc. Isto deve ser programado de forma que todas essas aplicações possam ser feitas, de forma a facilitar e flexibilizar o trabalho de Level Design (isto é, a montagem de fases). Isto poderia ser alcançado criando referências a diferentes entidades na classe que descreve o comportamento do inimigo (imagine que, dentro da classe do inimigo, existe uma referência a uma porta). No entanto, este método rapidamente se mostraria ser verboso e repetitivo, pois feriria o princípio de responsabilidade única do SOLID (*Single Responsibility Principle*) – *"Uma classe deve ter uma e apenas uma razão para mudar."* (6). Não cabe à descrição do comportamento do inimigo invocar a abertura de uma porta ou a mudança da câmera.

Ao lidar com este problema, aplicamos o padrão *Observer* que, de acordo com o livro *Level Up Your Code*:

> "It [Observer Pattern] allows your objects to communicate but stay loosely coupled using a "one-to-many"dependency. When one object changes states, all dependent objects get notified automatically."(7)

Em outras palavras, o padrão *Observer* permite que o código funcione tal como uma estação de rádio: um componente age como a estação de rádio que envia ***eventos***, enquanto múltiplos (e possivelmente diferentes) componentes **inscrevem-se** a esta estação, recebendo seus ***eventos*** quando invocados.

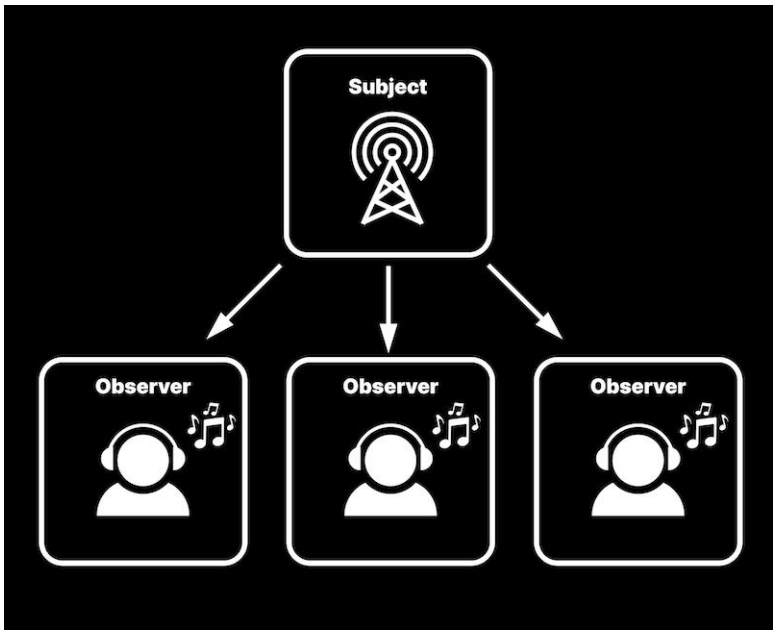

Figura 2: Funcionamento do padrão Observer. Fonte: (7)

Com este Design Pattern, podemos dar a habilidade de certos componentes a **invocar** certos eventos, e outros a **escutar** a eventos. Desta forma, a criação de fases e determinação de regras de causa-efeito no projeto é facilitada. A aplicação em si deste padrão será discutida no capítulo de desenvolvimento.

Padrão - Máquina de Estados Finito

O padrão *Finite State Machine*, simplificado para *FSM* busca simplificar comportamentos complexos (que poderiam ser descritos com uma lista longa de declarações de "se/senão") utilizando uma máquina de estados.

O exemplo principal deste problema no projeto é a descrição do comportamento do protagonista (personagem comandado pelo jogador). Considere que este comportamento depende destas variáveis:

1. *Inputs* do jogador;
2. Posição do personagem;
3. Interferências do ambiente (ataques de inimigos, por exemplo);
4. Momento da música.

Um programador, poderia, descrever o comportamento do protagonista utilizando declarações "se/senão". Porém, só com estas variáveis, estas declarações seriam repetitivas e longas. O padrão de *FSM* busca dividir partes destes comportamentos em **estados**, isto é, uma subcategoria de comportamentos que o protagonista deve assumir caso certas condições sejam cumpridas. Este padrão também ajuda a manter os princípios SOLID, em específico o princípio aberto/fechado (open/closed principle):

> "Aberto para extensão, mas fechado para modificação."(7)

Como cada estado é definido de maneira encapsulada, você pode facilmente adicionar novos estados sem interferir nos já existentes assim como modificar um estado sem afetar os outros. Desta forma, o projeto fica fácil de se estender, e modificações ocorrem em um espaço fechado. A aplicação de fato deste padrão, assim como o *Observer*, será discutida no capítulo de desenvolvimento.

2.2 Métodos ágeis para desenvolvimento

No mundo de desenvolvimento de Software, métodos ágeis de desenvolvimento não só estão bem disseminados mas estão na moda – e por um bom motivo: definem uma série de regras para organizar o trabalho iterativo em software. No que se deve a este projeto, a metodologia selecionada foi o SCRUM, por se tratar do método mais conhecido no mercado e já vivenciado pelo aluno durante experiências de estágio.

SCRUM

Para a definição da metodologia de desenvolvimento e de organização deste projeto, usou-se como guia o livro *Guia do Scrum* (8), que foi o trabalho que padronizou esta metodologia globalmente.

No entanto, como no caso deste projeto se trata do trabalho de uma única pessoa, (que deve exercer o papel de Product Manager, Product Owner e Developer ao mesmo tempo), esta metodologia teve que ser podada, retirando apenas conceitos que faziam sentido nesta aplicação.

Dos conceitos mais importantes, foram selecionados principalmente o *Sprint* e o *Sprint Planning*. Estes conceitos foram selecionados pois funcionam bem no âmbito de planejamento de projeto: auxiliam na determinação das metas a serem cumpridas, os seus pesos em prioridade e no planejamento de quando devem ser cumpridas. De acordo com o próprio *Guia do Scrum*, Sprints:

"São eventos de duração fixa de um mês ou menos para criar consistência. Uma nova Sprint começa imediatamente após a conclusão da Sprint anterior. (...) Sprints permitem previsibilidade, garantindo a inspeção e adaptação do progresso em direção a uma meta do Produto ao menos uma vez por mês."(8)

Desta forma, optou-se por utilizar uma metodologia que determina metas gerais em certos pontos do projeto, e a cada período de Sprint, estas metas são selecionadas para serem cumpridas.

2.3 Game & Level Design

Como o projeto se trata de um protótipo de jogo de Videogame, o aspecto de Game & Level Design não poderia ser deixado de lado. Afinal, mesmo com ótimas implementações de mecânicas e arte, um jogo rapidamente deixa de ser divertido quando o Level – isto é, o arranjo de todas as mecânicas e componentes de jogo – não é bem desenhado. De acordo com o Game Designer *Jesse Schell* em seu livro *The Art of Game Design* (9), Game Design é:

"The act of deciding what a game should be."(9)

Em outras palavras, game designers são as pessoas que decidem a função e posicionamento de cada mecânica e componente no jogo, sempre visando um objetivo central – proporcionar uma experiência ao jogador. No entanto, entre um projeto em branco e este objetivo, existem uma infinitude de opções e metodologias que podem ser tomadas. Schell propõe em seu livro metodologias (focadas na iteratividade e em *playtesting*) para atingir esta meta.

Para este projeto, foram selecionadas algumas ideias para basear o processo criativo e de desenvolvimento com o intuito de trazer as mecânicas de ludomusicalidade para o seu maior potencial possível. Desta forma, será possível avaliar o verdadeiro impacto do que foi desenvolvido.

Foco na Mecânica

Partindo da definição de Schell, os "componentes básicos"de um jogo são 4: as mecânicas, a história, a estética e a tecnologia. No que se deve à história e à estética, estes aspectos não serão considerados na monografia, já que extrapolam a proposta de projeto. A tecnologia já foi e será discutida em outros capítulos deste trabalho. Neste capítulo,

será dado um foco à mecânica e como devemos organizá-la de forma a proporcionar a melhor experiência possível ao jogador.

Mecânicas, de acordo com Schell, se referem ao conjunto de regras e sistemas que governam a jogabilidade. Para melhorá-la, é importante ressaltar os aspectos levantados no capítulo 11 do *Art of Game Design*(9) – *"Game Mechanics Must be in Balance"*. O ponto principal é o jogo deve manter o jogador no estado descrito como o estado "flow", considerado como o estado essencial para a diversão e imersão. Para isso, a jogabilidade deve se equilibrar entre o difícil e o fácil, de maneira a não bloquear o jogador mas também não desinteressá-lo com sua falta de desafios. Para isso, a dificuldade de um jogo deve ser incremental, de forma a se adaptar à evolução mecânica do jogador – sempre propondo novos desafios a serem superados.

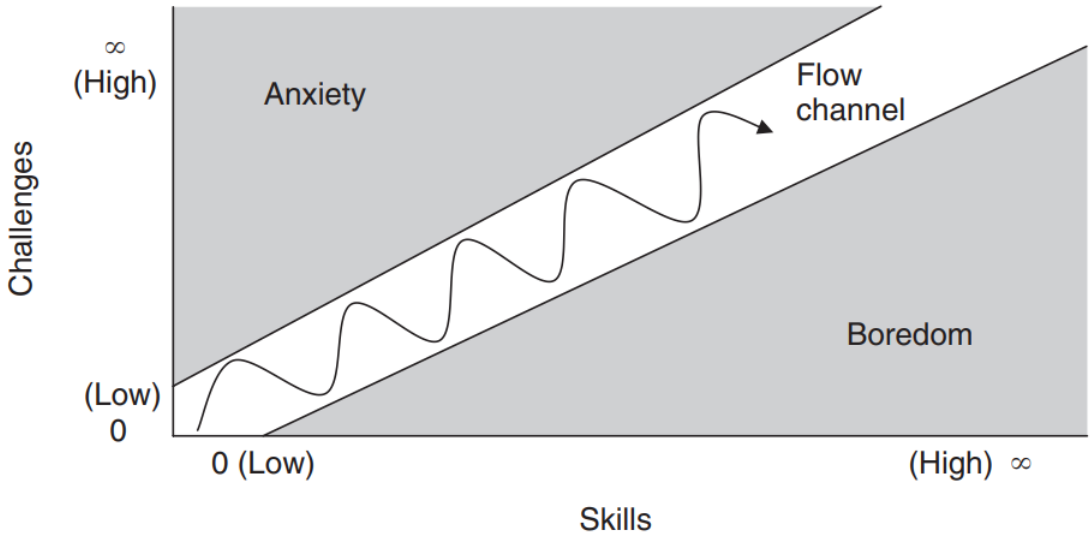

Figura 3: Gráfico representativo do estado de "flow". Retirado do capítulo 11 de *Art of Game Design*(9)

Entretanto, as mecânicas de um jogo não podem ser aplicadas em um vazio: elas devem ser instanciadas em um "espaço digital", ordenado de forma a orientar o jogador a seguir uma série de ações e experimentar a jogabilidade proposta. Este "espaço digital"nada mais é do que o *Level* – isto é, um espaço de componentes cuidadosamente colocado e ordenado por um designer.

Level Design em 2D

Para a construção do *Level*, buscaram-se referências voltadas para produção de jogos em 2D. Isso se deve ao fato de que, embora o jogo se passe em um ambiente tridimensional, a perspectiva *"top-down"* e a ausência de movimentos verticais (como pulos) fazem com que as interações principais do jogo ocorram todas duas dimensões.

Uma das referências selecionadas foi a do jogo *Hyper Light Drifter* (10), cujo estilo de jogabilidade e mecânicas se assemelham muito com as escolhidas para o projeto. Para estudo da metodologia de Level Design deste jogo, analisou-se o vídeo (11), em que a *Level Designer* Lisa Brown discute sobre os métodos e conceitos que foram importantes na concepção das fases do jogo.

Dentre os conceitos importantes, destacam-se 3: **1) Usar métricas estabelecidas como restrições**, **2) O conceito de *Prospect* X *Refuge*** e **3) Isolar combates / negar acesso ao espaço de refúgio**.

No primeiro, destaca-se como é importante basear as proporções e a estrutura de seu *level* com base nas mecânicas já estabelecidas no seu jogo. Isto é, o tamanho do personagem, sua velocidade, o tempo que ele demora para atravessar uma quantidade de espaço, o tamanho da tela, etc. De acordo com Brown:

> "Start laying things out at distances that were already defined by existing metrics in the game and see how they are connected."(8:35) (11)

Com essas restrições em mente, em seu processo criativo, ela desenha pequenas "peças"que vão se repetir ao longo da fase, e que ao serem acopladas, montam o todo. Esta limitação é boa para reduzir o retrabalho e também facilitar no momento de se iniciar o *design* de um novo *level*, principalmente pois ressalta as mecânicas a serem implementadas.

Sobre o conceito de *Prospect* X *Refuge*, Brown comenta sobre como a disposição física dos componentes de uma fase são importantes para passar a mensagem de "perigo"ou "segurança"para o jogador, e como são situações importantes para definir o fluxo do jogo:

> "In a lot of situations, in action games, you'll see these patterns where you'll have someone in a safe space have to go out in danger to get to a secondary refuge space further out."(12:30) (11)

É trabalho do *Level Designer* diferenciar os espaços de refúgio e espaços perigosos, no intuito de direcionar jogadores a se preparar para momentos de perigo ou criar estratégias para contorná-las.

O último conceito é uma evolução dessa dualidade, onde o *Level Designer* priva o jogador de um espaço de refúgio para aumentar a tensão do momento. Prover esta emoção de tensão é útil para demarcar momentos importantes e que exigem a atenção do jogador. Nas palavras de Brown:

> "Denying refuge space in order to ratchet up tension for that particular moment."(18:25) (11)

No entanto, este mecanismo não deve ser usado sempre, e sim para "pontuar"fluxos do jogador. Isto é, para dar uma sensação de finalização ou de desafio completo.

Para validar esses conceitos aplicados em versões já existentes do jogo, é crucial experimentar protótipos (mesmo que incompletos) em processos que chamamos de *playtesting*.

Playtesting

Em *Art of Game Design*(9), Schell define *playtesting* como o processo em que jogadores "reais"(significando, aqui, o público alvo do jogo) testam o jogo com o objetivo de que os game designers observem a experiência, coletem feedback e identifiquem problemas ou áreas de melhoria.

Deve-se orientar o *playtesting* para os seguintes propósitos:

1. Validar assunções: designers muitas vezes especulam como jogadores responderão a certos aspectos do jogo. *Playtesting* permite uma validação em tempo real destas preconcepções, sempre fornecendo *insights* importantes sobre o comportamento dos jogadores;

2. Identificar pontos problemáticos: por meio de *playtesting*, designers podem encontrar elementos do jogo que frequentemente confundem os jogadores, frustram-os ou aparentam ser desbalanceados. Isso permite que, em novas iterações, o produto seja melhorado nesses pontos;

3. Entender reações de jogadores: analisar não só o comportamento dentro do jogo, mas também a reação dos jogadores a certos aspectos: eles se divertem, se frustram, se concentram? Estas reações são os indicativos diretos de quais experiências os jogadores estão tendo;

4. Melhoria Iterativa: *Playtesting* faz parte de um processo iterativo. A partir da coleta de observações das testagens, o jogo deve ser refinado e mais uma sessão de

playtesting deve ser realizada, com o objetivo de sempre melhorar a experiência até que um nível satisfatório seja alcançado.

Esta noção de *playtesting* é extremamente compatível com a metodologia SCRUM, já que ambos se encaixam no sentido de que propõem um desenvolvimento iterativo, sempre em contato com o público alvo do produto.

3 Método de Trabalho

O projeto será desenvolvido utilizando o engine **Unity**, especificamente em um **ambiente 3D**. Será um jogo de batalha com vista superior, e consistirá de 10 a 30 minutos de gameplay em sua totalidade (uma versão mais completa do jogo poderá ser desenvolvida posteriormente). Nesta demo, o jogador irá explorar um *Level* com obstáculos e inimigos. O jogador deverá atacar em sincronia com a música enquanto também desvia dos inimigos. Tanto o jogador quanto os inimigos terão seus comportamentos regrados pela música.

3.1 Organização do Trabalho

Planejamento Geral

O planejamento geral do projeto consiste dos seguintes passos:

- Levantamento de funcionalidades
- Divisão de prioridades
- Criação de metas intermediárias
- Criação de cronograma

Como se trata de um projeto com fortes influências artísticas e criativas, não é possível fazer uma pesquisa da base de consumidores para definir e levantar requisitos. As **funcionalidades** de projeto foram, então, levantadas a partir da comparação com outros jogos já conhecidos e estabelecidos no mercado.

Após levantar as **funcionalidades** do jogo, estas então foram divididas com base em seu nível de prioridade para a formação de um protótipo jogável e testável para jogadores. As prioridades definidas foram: **prioritário**, **secundário** e **extra**.

A partir de um levantamento geral de todas as funcionalidades que o protótipo deve ter (por meio de um *brainstorming*), estas foram divididas entre as três categorias de prioridade determinadas. O critério para esta divisão é explorado em mais detalhes na seção de "Definição de funcionalidades", mas, em resumo – as funcionalidades **prioritárias** são aquelas cujas quais sem o jogo não existiria, nem na forma mais crua. As **secundárias** adicionam valor realçando as mecânicas básicas, e as **extra** agregam valor ao produto final, mas não são necessárias para a existência do protótipo.

Com base nas **funcionalidades** prioritárias e secundárias, montou-se um planejamento baseado em **metas intermediárias**, isto é, pontos no tempo cujo objetivo é desenvolver um grupo de **funcionalidades** do projeto. Dadas as **metas intermediárias**, desenvolveu-se um **cronograma**.

Por fim, foram definidas algumas etapas específicas para a validação do projeto, com o objetivo de certificar o impacto das mecânicas implementadas.

Workflow por funcionalidade

O *workflow*, isto é, o fluxo de desenvolvimento de cada funcionalidade consiste dos seguintes passos:

- Fase de estudo e pesquisa por referências;
- Fase de "playground";
- Desenho de solução;
- Implementação;
- Validação.

A fase de estudo e pesquisa por referências é, basicamente, uma pesquisa por métodos de aplicação de cada funcionalidade já existentes. Como se trata de um meio de desenvolvimento com uma grande comunidade, existem muitos vídeos e guias que fornecem soluções prontas para uma gama de funcionalidades. Após o estudo dessas implementações, inicia-se a fase de "playground", que consiste do teste desses métodos e do entendimento por meio da prática.

Dado um entendimento mais profundo e "*hands-on*"das soluções estudadas, é possível desenhar soluções por meio de **Diagramas UML** e **Design Patterns** conhecidos em Engenharia de Software, tornando a aplicação criada mais fácil de ser compreendida por

outros engenheiros e melhor documentada para o futuro. Por fim, a implementação da solução criada utilizando os princípios de engenharia de software, que deve ser seguida de uma validação, que testa a funcionalidade implementada juntamente com as outras funcionalidades antigas.

Aplicando conceitos de Scrum da seção 2.2, faz sentido, no início de cada *sprint*, analisar as funcionalidades a serem implementadas naquele período de tempo e decompô-las em tarefas atômicas, fáceis de serem determinadas. Cada um destes itens pode ser considerado como uma tarefa.

Para auxiliar no acompanhamento de tarefas e sua execução, a cada início de *sprint* (geralmente uma vez por mês), levantou-se as funcionalidades ainda pendentes e criou-se um "*board*"(uma visualização fácil e interativa) das tarefas a serem realizadas. Este estilo de "*board*"é conhecido em metodologias ágeis como *Kanban*, auxilia a analisar e atualizar o ciclo de vida de tarefas com o uso de 4 principais colunas:

- **Backlog:** contém tarefas que foram repassadas para próximas sprints e não serão executadas agora;
- **To Do:** contém todas as tarefas a serem executadas no sprint atual;
- **Doing:** contém todas as tarefas que já foram iniciadas mas ainda não finalizadas;
- **Done:** contém todas as tarefas já finalizadas.

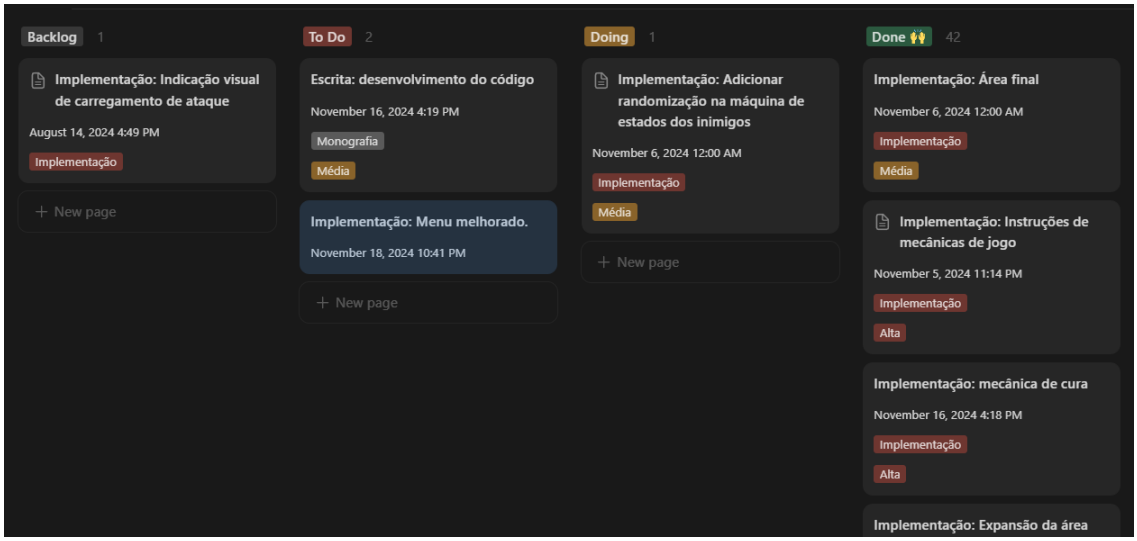

Figura 4: "*Board Kanban*"para acompanhamento do projeto

3.2 Divisão das funcionalidades

Funcionalidades prioritárias

Para as **funcionalidades prioritárias** foram definidas as tarefas que gerariam o jogo na sua versão mais crua, permitindo que fosse testada a premissa básica do projeto. Todas estas tarefas serão realizadas pelo aluno.

- Mecânicas básicas para o jogo (câmera, movimentação, colisão, etc.);
- Mecânica de ataque rítmico para o jogador;
- Mecânica de ataque rítmico para os inimigos;
- Comportamento básico para inimigos (padrões de ataque e perseguição);
- Desenvolvimento de um estágio básico (colocamento de inimigos e obstáculos);
- Tela de início do jogo.

Funcionalidades secundárias

Para as **funcionalidades secundárias**, foram definidas tarefas que realçam fortemente as mecânicas básicas, deixando a experiência geral do jogador mais imersiva. Algumas destas tarefas, por exigirem capacidades externas à engenharia de computação, podem ser realizadas por pessoas externas.

- Modelos 3D para os personagens (jogador e inimigos);
- Fase com mecanismos mais avançados (portas, alavancas, etc.);
- Shaders (para estilização do jogo);
- Sincronização da música com iluminação do cenário.

Funcionalidades extras

Para as **funcionalidades extras**, foram definidas tarefas que agregam muito valor a jogos e tarefas de refinamento, porém não necessariamente precisam ser definidas em uma demo para a validação da mecânica do jogo.

- Fase com inimigo especial (*Boss*);

- Mecânicas de diálogo entre jogador e NPCs (Non-playable characters);
- Definição de roteiro e história para o jogo;
- Aplicação de efeitos visuais;
- Aplicação de efeitos sonoros;
- Melhoria das animações dos modelos de jogo.

3.3 Métodos de validação

Para a validação, foram escolhidos alguns métodos de validação que são frequentemente implementados na indústria:

- ***Playtesting* interno**: Fazer iterações de teste por parte dos próprios desenvolvedores sobre as mecânicas do jogo – validando se estão funcionando como devem;
- ***Playtesting* externo**: convidar pessoas que estão fora do meio de produção para testar o jogo prototipado. Ao final do teste, fazer uma breve entrevista;
- ***Playtesting* em massa**: envio de formulários com um link de fácil acesso para o jogo. As pessoas deverão jogar o jogo conforme quiserem, sem a presença de um entrevistador e, posteriormente, responder ao formulário.

4 Especificação das Funcionalidades

Nesta seção será detalhada cada *funcionalidade* prioritária e secundária inclusa no desenvolvimento do projeto. Cada uma é importante para a entrega de um produto robusto.

As *funcionalidades* extras são importantes, porém não estão diretamente no escopo do projeto e, portanto, não precisam ser definidas na proposta.

4.1 Controles Básicos

Translado e Rotação do Avatar

Esta *funcionalidade* se refere à tradução dos *inputs* do jogador (o teclado do computador e o mouse) em translação do avatar no espaço 3D do *Unity*. Deverá ser considerada

a movimentação em 8 direções (Norte, Nordeste, Leste, Sudeste, Sul, Sudoeste, Oeste, Noroeste) por parte do jogador. Além do translado, o avatar do personagem deve se rotacionar para a direção que está se movendo.

Após testes de usabilidade, confirmou-se que a movimentação deve conter também um fator de aceleração e desaceleração – para propiciar um movimento mais realista e interessante. No entanto, a velocidade deve atingir um máximo e manter-se nele, não permitindo que o avatar atinja velocidades infinitas.

Uma manobra de *dash* também está inclusa para melhorar a dinamicidade do jogo – fornecendo ao jogador a possibilidade para manobrar o campo de batalha e atravessar diferentes obstáculos, como valas ou buracos. Não serão inclusos pulos verticais.

Câmera

Além da movimentação do jogador, a câmera também deve seguir o avatar do personagem com uma movimentação não-linear (se movendo como um sistema amortecido) – de forma a tornar a perspectiva mais natural e menos sensível à mudanças bruscas de posição. Isso deve acontecer para que o avatar não fique fora da visão do jogador. A rotação da câmera em torno do personagem não está contemplada no protótipo.

Interações com objetos

A interação com objetos é uma mecânica utilizada em alguns jogos de RPG para fornecer uma maior variedade de opções de tarefas a serem realizadas em uma fase. Por exemplo, em alguns jogos, existem chaves que o jogador pode coletar (interagindo com baús, por exemplo, para coletá-las), e enfim desbloquear portas que permitem o progresso no jogo. No protótipo, contemplou-se uma simples implementação de duas interações básicas: cura (restaurar a vida do personagem ao seu máximo) e alavancas (que abrem portas e levantam pontes).

4.2 Ataques

Ataques são uma parte crucial do projeto, pois é o que define o avanço do jogo e sua mecânica principal – a sincronização musical. Em uma primeira instância, todos os ataques do jogador serão corpo a corpo (ou *melee*), e o dos inimigos serão à distância.

Além disso, para o funcionamento das funcionalidades de ataque, é também fundamental a existência de um valor de vida para cada personagem (jogador e inimigos),

cujo valor deve ser afetado quando um personagem recebe um ataque. No entanto, a visualização deste valor só será representada em uma interface (a HUD, funcionalidade que é explicada com mais detalhes na seção de interfaces).

Ataques do Jogador

Para os ataques do jogador, serão projetados volumes visíveis (ou caixas) próximas ao jogador. Estas caixas se chamam "*Hurtboxes*", e elas definem o volume em que o ataque tem efeito. Já os corpos dos inimigos possuem volumes invisíveis chamados "*Hitboxes*", que definem onde inimigos podem ser atacados. Se uma *Hurtbox* entra em contato com uma *Hitbox*, considera-se que o ataque atinge o personagem a quem a *Hitbox* pertence.

Desta forma, os ataques do jogador vão projetar estas *hurtboxes*, e os inimigos cujas *hitboxes* forem atingidas deverão tomar dano (isto é, reduzir o valor da vida do inimigo) e serem empurrados na direção contrária do avatar.

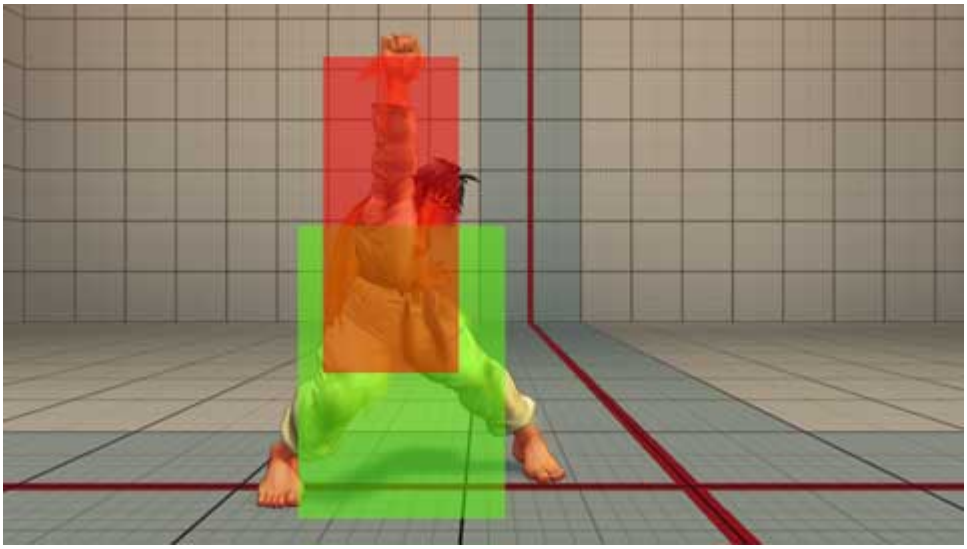

Figura 5: Imagem ilustrativa retirada do jogo *Street Fighter* (12). Na imagem, é possível ver *hitboxes* (em vermelho) – onde o ataque é dado – e *hurtboxes* (em verde) – onde o personagem pode receber ataques.

Como nesse caso não estamos trabalhando com modelos complexos, a *hurtbox* do jogador será uma cápsula (um cilindro acoplado com esferas de mesmo raio nas extremidades superior e inferior), enquanto a *hitbox* será a de um cubo.

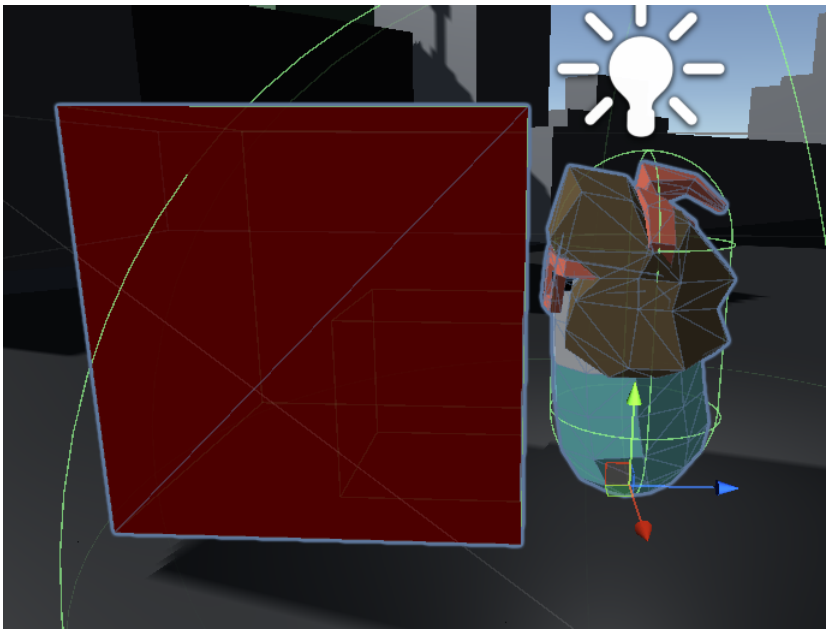

Figura 6: Imagem do personagem no protótipo. Na esquerda, em vermelho, a *hitbox* do jogador, e na direita, encapsulando o personagem, a *hurtbox* do jogador

Ataques dos Inimigos

Para os ataques dos inimigos, serão criados projéteis a partir dos corpos dos inimigos, que possuirão uma velocidade maior que 0. Esses projéteis possuem uma *hurtbox* cada, e também sua própria *hitbox*. Caso o jogador ataque um projétil (acertando a *hitbox* do projétil), este será nulificado em uma ação comumente denominada *parry* em jogos (que significa bloqueio). Caso a *hurtbox* de um projétil atinja a *hitbox* do avatar do jogador, considera-se que o ataque atingiu o jogador. Desta forma, assim como o inimigo, o avatar deve ser empurrado na direção contrária do projétil, e tomar dano.

Assim como o jogador, o *hurtbox* dos inimigos também serão cápsulas, enquanto a *hitbox* dos projéteis serão esferas.

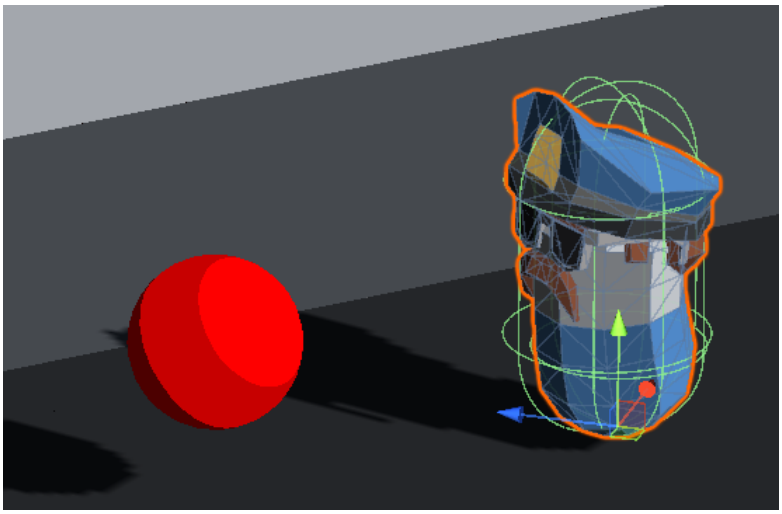

Figura 7: Imagem do inimigo no protótipo. Na esquerda, a esfera em vermelho, a *hitbox* do projétil, e na direita, encapsulando o inimigo, a *hurtbox* do mesmo

4.3 Interfaces

Interfaces são cruciais para o funcionamento de todos os jogos modernos. São objetos de interação com o usuário que não representam elementos no espaço 3D do Unity, mas apresentam informações importantes para o funcionamento do jogo (como por exemplo, a vida do jogador, o mapa, uma descrição de seus objetivos, etc.). Dessa forma, podem apresentar ao usuário diferentes informações, como a quantidade de vidas do jogador, um cronômetro, o menu de seleção de fase, o menu de pause, etc.

Para o protótipo deste jogo eletrônico, serão necessárias dois tipos de interface:

Heads Up Display (HUD)

Heads Up Display ou HUD é a nomenclatura dada à interface presente durante o runtime do jogo, fornecendo informações úteis durante o jogo (como a quantidade de vidas, a quantidade de munição, etc.). No caso do famoso jogo **Legend of Zelda: Ocarina of Time** (4), a interface HUD disponibiliza informações como a vida do personagem (quantidade de corações no canto superior esquerdo), os equipamentos que ele carrega (ícones no canto superior direito), e outras informações úteis ao jogador.

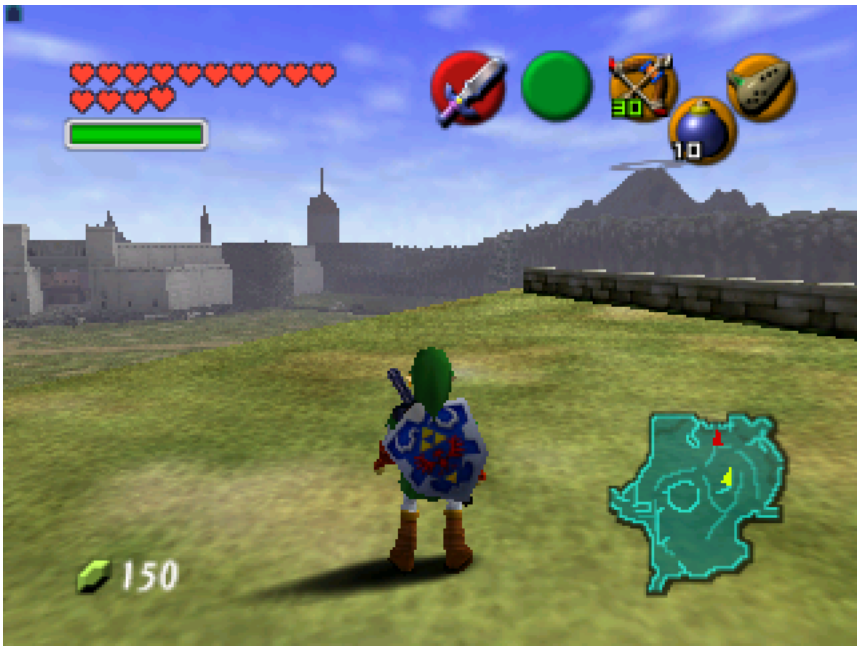

Figura 8: Exemplo de HUD do jogo de sucesso **Legend of Zelda: Ocarina of Time**

Para o protótipo de jogo que está sendo produzido, a HUD deve apresentar somente a vida do jogador, um sinalizador da interação com objetos, e instruções que serão mostradas à tela, ensinando o jogador comandos do jogo.

Figura 9: Captura de HUD do jogo, apresentando a vida (canto superior esquerdo), uma instrução ensinando ao jogador como interagir com alavancas e uma barra circular, apresentando o progresso da interação.

Menu & Tela de Início de Jogo

Menu e a tela de início de jogo são interfaces que param o jogo por completo, pausando a simulação do espaço 3D do Unity. Estas interfaces servem para redirecionar o jogador para diferentes fluxos – podendo permitir ao jogador começar um novo jogo, continuar um jogo já existente, mudar configurações, etc.

Figura 10: Exemplo de tela de início do jogo de sucesso **Minecraft**

Para contemplar os casos necessários para o jogo, foram criadas apenas três interfaces semelhantes: uma para o início do jogo, uma para o caso em que o jogador perde o jogo (sua vida chega a 0) e outra para caso o jogador ganhe o jogo.

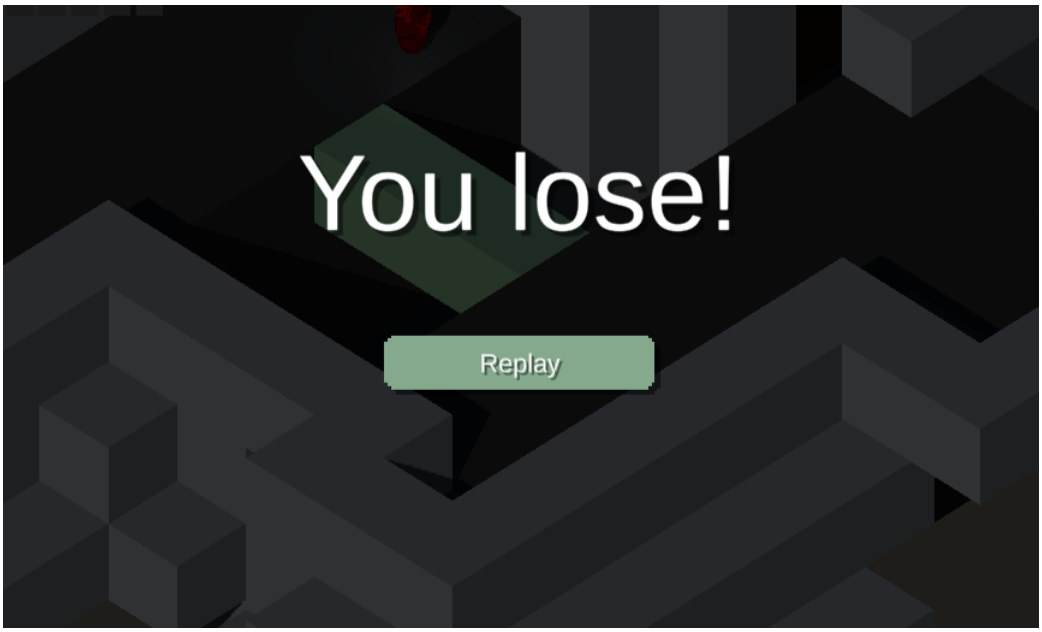

Figura 11: Tela para caso de derrota no protótipo do jogo. O jogador pode pressionar o "*replay*", permitindo uma segunda tentativa.

4.4 Sincronização Musical

A sincronização com a música é a mecânica de maior complexidade e a mais importante deste projeto. Esta mecânica consiste em, nos momentos chamativos da música, realizar eventos em diferentes objetos de jogo de forma simultânea e sincronizada. Em outras palavras – digamos que os eventos de ataques dos inimigos acontecem quando a bateria (instrumento musical) toca o bumbo. A ideia é que, todo momento em que o bumbo é tocado, os inimigos disparem um ataque simultaneamente.

Os eventos que devem ser sincronizados com a música são os seguintes:

- Ataque do jogador;
- Movimento de Dash do jogador;
- Ataque dos inimigos;
- Troca de estado da máquina de estados dos inimigos;

No caso das funcionalidades para o jogador, a ideia é propiciar uma melhoria no ataque e na movimentação caso o jogador insira o *input* sincronizado com o tempo da música.

Para que o efeito desejado seja alcançado, espera-se que o desenvolvedor do jogo seja capaz de determinar momentos na música em que certos eventos devem ocorrer (por exemplo, ataque do jogador ou ataque do inimigo). Desta forma, o workflow de desenvolvimento deve atender não só o programador, que deve facilmente consumir as informações de quando esses eventos irão acontecer, mas também deve atender ao compositor/músico, de forma a ser fácil ter o controle de quando cada evento irá ocorrer.

Além disso, foram definidas as seguintes metas de qualidade:

- A sincronização deve ser bem feita, a ponto de ser imperceptível qualquer *delay*;
- A sincronização deve ter um bom *feel* – isto é, se o jogador sentir que os eventos na música devem ocorrer, eles deveriam ocorrer;
- A sincronização por parte do jogador deve ter uma janela – isto é, um *threshold* em volta do momento do evento no qual o *input* é considerado sincronizado.

4.5 Arte do Jogo

A arte de um jogo tem um impacto muito grande no seu valor e no efeito de imersão de seus jogadores. Por exemplo, o xadrez, jogo com mais de um milênio de idade, que possui uma identidade visual clássica definida pelo tabuleiro e o formato de suas peças.

No entanto, como não se trata de um aspecto crucial para o *Proof-of-Concept* estabelecido, a arte do jogo será simplificada e reduzida aos seguintes itens:

- Modelos 3D para o jogador e para os inimigos;
- Determinação de cores para o cenário;
- Shaders para estilização do jogo (programas que especificam como serão renderizados os dados gráficos do jogo).

4.6 Comportamento dos Inimigos

O comportamento dos inimigos é o que descreve a tomada de decisão destes personagens que não são controlados pelo jogador. Dentre os aspectos que ele determina, estão:

- A direção da movimentação;

- A velocidade de movimentação;
- O momento em que atacam;
- Em que direção atacam.

O comportamento dos inimigos no jogo deverá ser programado para adicionar dinamicidade e estratégia ao jogo. Os inimigos devem ser móveis, de forma a não gerar uma jogabilidade estática e previsível. Os inimigos também poderão ter algumas informações para melhor adaptar sua tomada de decisão, como a posição do jogador, ou se há linha de visão com o jogador para poder disparar um projétil.

TODO: Imagem do inimigo com linha de visão versus um sem linha de visão

Para ser engajante, os inimigos deverão ser minimamente inteligentes: devem evitar manter proximidade com o jogador, recuando quando este está próximo, devem se movimentar para conseguir linha de visão com o jogador para lançar projéteis, etc. A ideia é que o jogador consiga observar comportamentos genéricos, de forma a conseguir montar estratégias para derrotar os inimigos – mas o comportamento também não pode ser completamente determinístico, para evitar que seja muito previsível e – portanto, monótono.

Diferentes tipos de comportamento podem ser desenvolvidos (por exemplo, um inimigo com ataques corpo a corpo teria comportamentos diferentes), mas para este projeto, só o comportamento à distância será considerado.

5 Desenvolvimento Técnico

Neste capítulo serão descritos aspectos técnicos relativos ao projeto, especificamente no que se deve à sincronização musical, ao sistema de eventos e às máquinas de estado do jogo. O objetivo é aprofundar em termos mais técnicos de forma a detalhar os problemas que as soluções desenvolvidas buscam resolver, junto com uma discussão sobre sua eficiência.

5.1 Sincronização Musical

Conforme foi abordado na definição da funcionalidade de sincronização musical 4.4, trata-se de uma das mecânicas principais – portanto, deve ser bem executada. A principal preocupação do ponto de vista de código é a de manter a sincronização consistente ao longo da execução do jogo.

Problema: Trabalhando com tempos discretos

O *Unity* opera fazendo atualizações a cada chamada da função *Update*. Desta forma, as operações de código sempre ocorrem em tempos discretos (e não contínuos), o que acarreta numa inerente desincronia entre o evento na música e o evento em código que ele dispara.

A tarefa não se trata de reduzir a desincronia a 0, o que seria impossível considerando que nenhum computador consegue operar em tempo contínuo. A ideia é reduzir a desincronia para valores que são imperceptíveis aos jogadores. Isto pode ser alcançado disparando o evento na próxima chamada da função *Update* do *Unity*. Como a taxa base de chamadas de *Update* é de $60hz$ (60 vezes por segundo), a maior desincronia possível seria de aproximadamente 16 milissegundos, o que é considerado tolerável já que o tempo de reação médio humano é de 200 milissegundos. (Vale comentar que o *Unity* também tem uma forma de atualizar numa frequência maior que $60hz$. No entanto, esta taxa é variável, dependendo da velocidade de cada máquina que executa o jogo. Desta forma, o uso desta forma de *Update* é mais inconsistente e não foi utilizado.)

Com isso, o problema passa a ser garantir que a chamada da função acontecerá na chamada do *Update* posterior ao tempo do evento na música. No entanto, como podemos garantir que o valor analisado do tempo do evento na música é correto?

Problema: Threads Separadas

A sincronização com a música é a mecânica de maior complexidade e a mais importante deste projeto. Como a música em Unity é processada em uma *thread* separada, não é possível comparar o tempo de execução de quando um efeito musical acontece usando o tempo da thread do *script* do jogo – pelo menos não diretamente. Além disso, como o jogo tem variações no seu tempo de processamento e FPS (*frames* por segundo), o contador de tempo do jogo não é o mesmo do contador de tempo da música – sendo assim, impossível sincronizá-los simplesmente por tempo. Não só isso, mas devido ao fato de se tratarem de duas threads diferentes (no cálculo do tempo da música e o tempo do jogo) torna a comparação de tempo irrelevante, e erros nessa comparação são acumulativos.

Solução: Obter o tempo da música

O Unity possui uma funcionalidade que permite analisar quantos "samples de áudio"já foram tocados de um som, chamada **AudioSource.timeSamples**. Com ela e sabendo

quanto tempo cada *sample* consome, é possível determinar em que exato tempo da música o áudio está tocando a partir da função:

$$musicTime = \frac{musicSource.timeSamples}{musicSource.clip.frequency} \tag{1}$$

Dado o tempo da música, é possível determinar quando certas batidas musicais importantes ou chamativas acontecem – e assim, realizar o efeito de sincronização desejado. No entanto, as músicas são variáveis, e determinar cada momento de sincronização manualmente seria ineficiente, trabalhoso e também impreciso.

Uma maneira para lidar com a determinação desses tempos seria utilizando arquivos **MIDI** (um tipo de arquivo representativo de notas musicais e o tempo que são tocadas). Este tipo de arquivo é bem disseminado pela comunidade produtora de música, e músicos/compositores conseguem exportar este tipo de arquivo com facilidade em softwares de produção musical (como *Ableton Live*, por exemplo). Desta forma, os desenvolvedores do áudio podem produzir um MIDI com todos os tempos em que algum evento deve ocorrer em jogo. Com este arquivo MIDI, é possível desenvolver um programa que obtém um vetor de valores de tempo em que um determinado evento deve ser executado (por exemplo, ataque dos inimigos). Este vetor pode então ser passado para o Unity, que assim conseguirá chamar os eventos dentro do jogo.

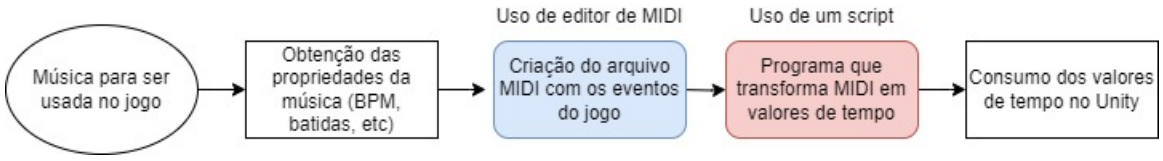

Figura 12: Esquemático do *workflow* para a sincronização musical

Dado os vetores com os valores dos tempos para diferentes eventos, é possível então criar efeitos diretamente no Unity. Para os inimigos. isto é simples: mandar os inimigos atacarem quando um de seus eventos for chamado. Para o jogador, um pouco mais de tratamento deve ser necessário, pois é improvável que o jogador acerte exatamente o tempo em que o evento for chamado.

Para mitigar este efeito, o jogo deverá aceitar uma "margem de erro-- ou *threshold* – para o input do jogador. Assim, o jogo considera que o ataque foi realizado com sucesso se o input estiver dentro de um intervalo estabelecido pelo momento do evento somado ou subtraído com o *threshold* estabelecido. Em outras palavras, se a diferença entre o tempo em que o input for identificado e o tempo do evento for menor do que a "margem

de erro", então o *input* está sincronizado. Caso contrário, não está.

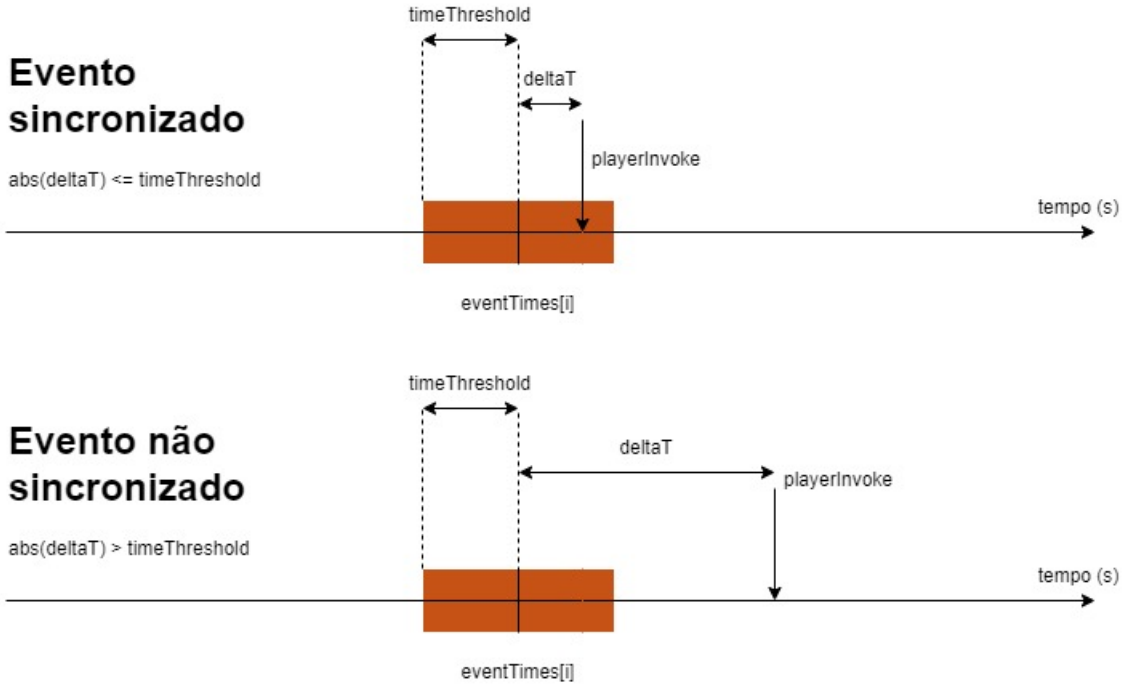

Figura 13: Gráfico para representação da "margem de erro"para o *input* do jogador.

O valor da margem de erro foi obtido através de um processo de *playtesting* interno. O valor obtido foi de $0.2s$, que pode ser considerada uma margem grande. No entanto, como o protótipo serve também para introduzir jogadores à mecânica de sincronia musical, seu propósito é incentivar jogadores a compreender a mecânica e a ter uma boa experiência com a mesma. Assim, é melhor que seja mais fácil acertar o ritmo do que errá-lo.

Pontos a melhorar: Delay no Áudio

Embora o código faça com que os momentos de chamada de eventos musicais seja correto diante dos critérios estabelecidos, existem alguns pontos de melhoria caso este projeto fosse se tornar um produto amplamente usado. O principal deles é o ajuste do valor no **delay do áudio**. Considere a seguinte situação: um jogador está utilizando um fone de ouvido *bluetooth* para jogar. Neste caso, seria necessário considerar a latência do áudio introduzida pela tecnologia bluetooth. Em realidade, todo dispositivo de áudio introduz um *delay* ao sistema, alguns maiores do que os outros. Desta forma, caso o produto fosse distribuído para diferentes máquinas com diferentes sistemas de som, seria necessário implementar uma solução que ajustasse esse *delay* conforme necessário.

Uma proposta de solução para esta questão seria implementar uma configuração que ajustasse este valor conforme necessário. Uma forma de implementar uma forma de se ajustar este valor é por meio de uma etapa de calibração: antes de começar o jogo, requereria-se ao jogador que ele insira comandos de acordo com o que ele ouve na música – desta forma calibrando de acordo com o sistema atual do jogador. Esta opção de calibração estaria sempre disponível nas configurações do jogo, permitindo que o usuário ajuste caso ele mude o seu sistema de áudio.

5.2 Sistema de Eventos - Observer

Problema: Múltiplas referências ao mesmo evento

Diante da criação de uma mecânica que chama uma função para um devido momento musical, surge então outra questão: como fazer esta função chamar diversos componentes simultaneamente? Como já foi previamente abordado na seção de Aspectos Conceituais 2.1, podemos utilizar o *Design Pattern* ***Observer*** para resolver esta questão – no entanto, com algumas modificações para se adaptar à **arquitetura de componentes** do *Unity*.

Esta **arquitetura de componentes** designa que para cada objeto possa ser agregado um ou mais componentes – uma classe chamada **MonoBehaviour**. Estes "comportamentos"funcionam como componentes, já que eles adicionam *scripts* diferentes a um mesmo objeto, podendo referenciar outros componentes ou objetos. Diante disto, é necessário repensar como seria uma implementação do padrão **Observer**: existiriam componentes do tipo **Subject** – isto é, que chamam os eventos – e componentes do tipo **Observers**, que se inscreveriam aos *Subjects*. No entanto, esta estrutura faz com que seja necessário que os *Observers* tenham uma referência direta aos *Subjects*, criando uma forte dependência e quebrando todo o propósito dessa aplicação.

Solução: Padrão Observer com ScriptableObjects

Para resolver essa questão, cria-se uma camada: um objeto da classe **ScriptableObject** do *Unity* cuja responsabilidade é fazer este intermédio entre os *Observers* e *Subjects*. A vantagem de se utilizar a classe de *ScriptableObjects* é que esta classe, diferentemente dos *MonoBehaviours*, não precisa estar vinculada a um objeto em jogo: ela é uma instância na pasta de *Assets* do projeto. Desta forma, o desenvolvedor pode criar eventos sem ter que vinculá-los a objetos no projeto.

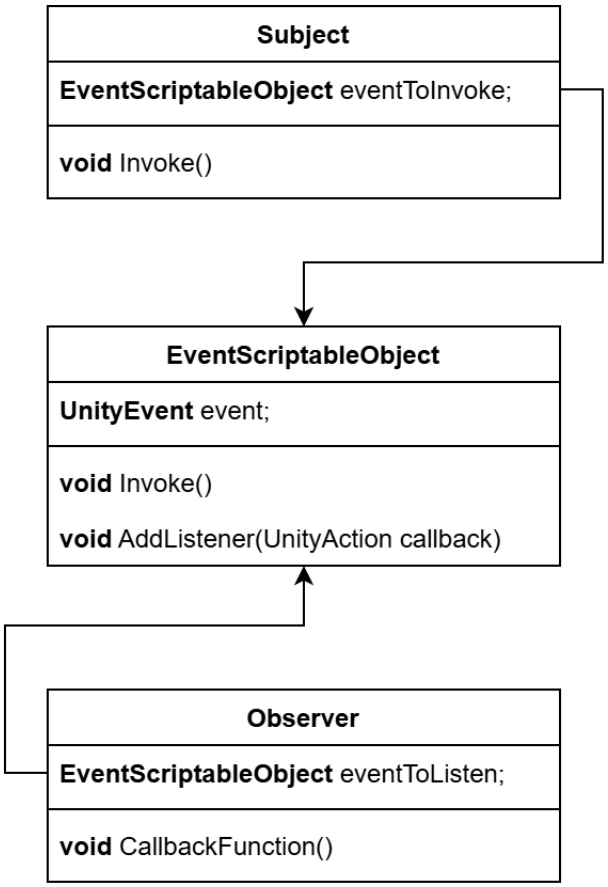

Figura 14: Diagrama representativo do **sistema de eventos**

Além disso, a criação dessa camada intermediária permite que haja uma relação ***N para N*** entre todas as camadas – se assim o desenvolvedor desejar. Por exemplo, se na classe de *Subject* ou *Observer* for interessante ter múltiplos eventos que devem ser chamados ou observados, basta instanciar os múltiplos **EventScriptableObjects**, ou até mesmo uma lista deste tipo. O mesmo vale para os *EventScriptableObjects*: como são uma classe que é apenas referenciada e não contém referência nem aos *Subjects* ou *Observers*, um mesmo evento pode ser referenciado por múltiplos ou nenhum de cada (isto é, múltiplos *Subjects* e *Observers* podem chamar e observar, respectivamente, o mesmo evento).

Aplicação: Eventos Musicais

A aplicação dos eventos musicais requer uma implementação de algumas funções a mais no padrão. Desta forma, criou-se uma classe filha (por herança) da *EventScriptableObject*: a **MusicEventScriptableObject**. Esta classe contém funções específicas

no contexto musical, como a **UpdateMusicTime**, que recebe o tempo da música e, baseado no vetor de valores de tempo musicais (cuja obtenção é descrita na seção 5.1), determina-se se os *Observers* devem ser chamados ou não. Além disso, disponibiliza-se a função *CheckEventNearTriggerTime*, que devolve um *booleano* para caso o momento atual da música esteja dentro da janela de tempo descrita na mesma seção mencionada.

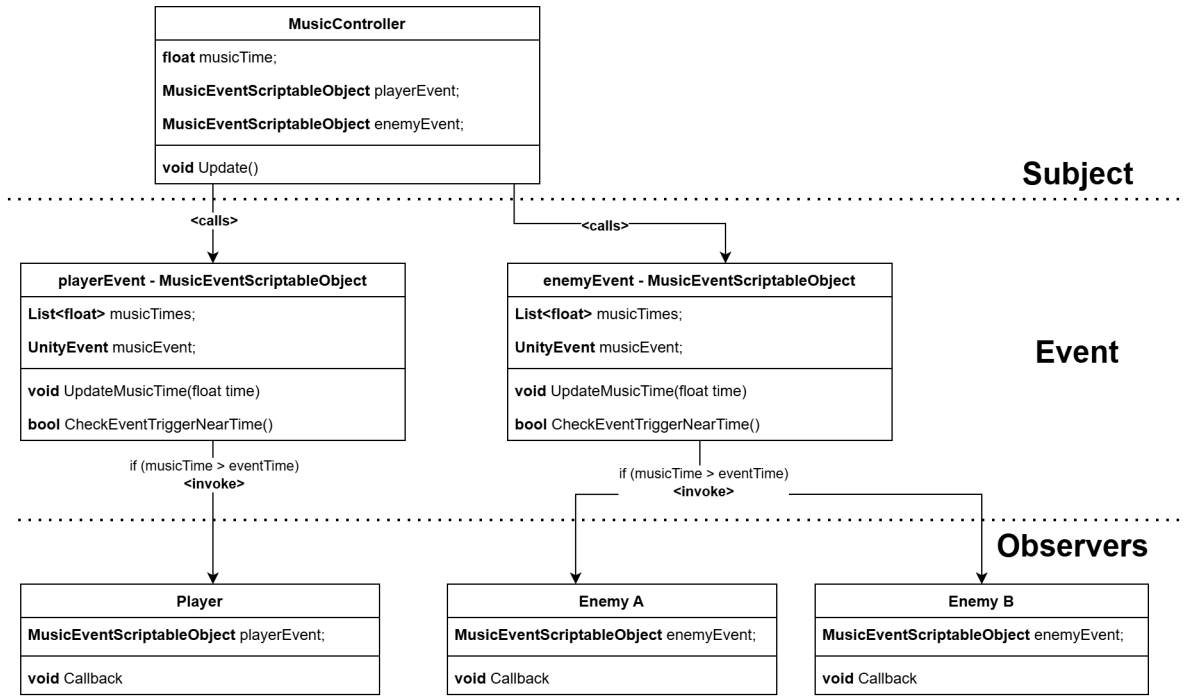

Figura 15: Representação da arquitetura de sistema de eventos musicais.

Neste caso, é possível visualizar a aplicação deste sistema no contexto do jogo, onde somente dois tipos de eventos musicais foram implementados. Os benefícios dessa estrutura são:

- **Escalabilidade de eventos:** caso seja interessante adicionar um novo tipo de evento, basta adicionar uma referência no *MusicController* e criar outro *MusicEventScriptableObject* – colocando no último o valor dos tempos obtidos a partir do arquivo MIDI conforme desejado;

- **Escalabilidade de observers:** para adicionar uma nova instância de um inimigo, basta só copiar a referência ao evento *enemyEvent*. Caso se queira criar uma nova classe que observa o *playerEvent*, basta criar uma classe com uma referência a um *MusicEventScriptableObject* e obter a referência ao evento desejado;

- **Manutebilidade:** para modificar os tempos utilizados para os eventos, ou para modificar alguma lógica na chamada de eventos, basta modificar na camada do *MusicController* ou *MusicEventScriptableObject* e as mudanças vão refletir nas chamadas de todos os *Observers*. Além disso, ao modificar o código de cada um dos *Observers*, nada muda no código das outras duas camadas e também no código de *Observers* de classes diferentes – cada componente tem sua responsabilidade única e bem definida.

Pontos a melhorar: Implementação de UI no Desenvolvimento

Dois grandes problemas foram percebidos durante o desenvolvimento após a implementação desta estrutura:

- **Encontrar quais componentes referenciavam certos eventos:** dado um evento, digamos o *playerEvent*, é difícil na atual estrutura do *Unity* encontrar todos os componentes que chamam esse evento;

- **Aspecto visual na implementação no Unity:** ao adicionar um evento no sistema, é necessário criar um *EventScriptableObject* na pasta *Assets*, e em seguida referenciá-lo em cada componente desejado. Este processo, embora simples, é pouco visual.

Estes problemas não foram de grande tamanho durante o desenvolvimento deste protótipo, principalmente devido ao tamanho da equipe. Quando se tratam de projetos maiores, com equipes mais diversas, que (por exemplo) não são tão versadas em programação, essas questões seriam mais problemáticas. Isso se deve ao fato de que, embora faça sentido lógico e seja simples, não se trata de uma implementação trivial para pessoas de outras áreas.

Assim sendo, uma boa solução para esta questão seria a criação de uma **interface com grafos**, onde desenvolvedores poderiam criar nós de *EventScriptableObjects* e conectar com as funções dos componentes *Subjects* e *Observers*, facilitando assim tanto o processo de encontrar quais eventos são referenciados por quais componentes, assim como ajudando no aspecto visual da implementação.

5.3 Máquinas de Estados - FSM

Problema: Descrição de comportamentos complexos

Como foi descrito na seção de Aspectos Conceituais 2.1, existe o problema de como se devem programar comportamentos mais complexos, como o controle do jogador e de inimigos no jogo. Para resolver esta questão, desenvolveu-se um sistema de máquina de estados finita (ou *Finite State Machine*) adaptado ao *Unity*, de forma a ser mais fácil de aplicá-lo e modificá-lo conforme necessário.

Solução: Padrão Finite State Machine

Para aplicar o padrão de Finite State Machine, é importante definir as classes que servirão como base do *Design Pattern*: a classe **State** e a classe **StateMachine**. A primeira será responsável por descrever cada estado e como suas ações se darão, e a segunda descreve a máquina em si, e o que ela executa a cada ciclo. No caso da classe *State*, essa classe será, na verdade, uma **classe abstrata**. Isto é, não terá nenhuma instância própria, mas terá classes filhas que herdarão suas variáveis e suas funções para serem implementadas individualmente. Em contrapartida, a classe *StateMachine* não precisará ser diferente em suas aplicações, já que somente executa os estados e suas transições.

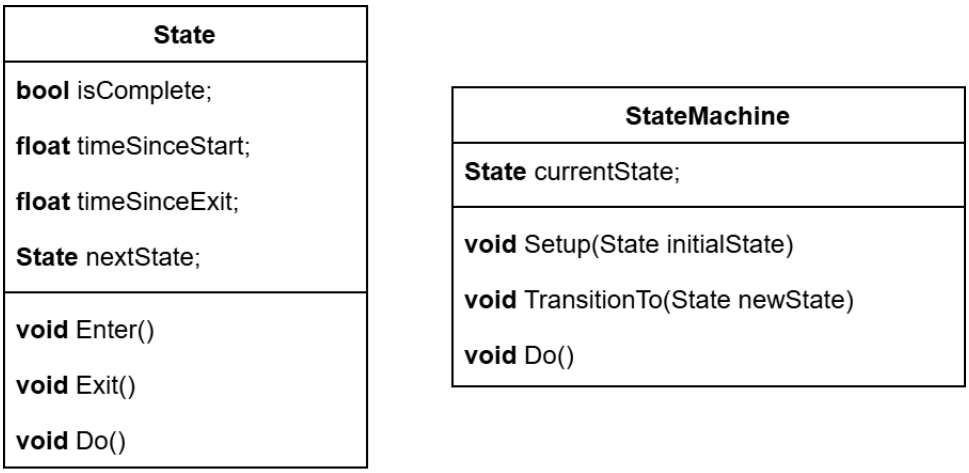

Figura 16: Representação das classes **State** e **StateMachine**.

No caso da classe **StateMachine**, seu funcionamento é simples: ela possui um estado atual que é configurado na função **Setup**. Quando deseja trocar o estado atual por outro, chama-se a função **TransitionTo**, que faz os tratamentos necessários. Por fim, em toda

atualização (isto é, função *Update*), chama-se a função **Do**, que somente chama a função **Do** do estado atual (já que o funcionamento da máquina de estados é definido pelo seu estado atual).

As funções mais importantes da classe **State** são as funções **Enter**, **Exit** e **Do**. A primeira é executada quando o estado se torna o estado atual na máquina de estados, a segunda quando o estado atual é trocado por outro (isto é, na saída do estado da máquina), e a terceira é executada em toda função **Do** da máquina de estados – isto é, a cada atualização.

No caso da classe *State* também vale ressaltar que a variável *isComplete* é uma booleana que permite sinalizar quando o estado considera que foi finalizado – isto é, se internamente o estado reconhece que foi completado e deve ser substituído por outro. Quando a variável *isComplete* for verdadeira, a máquina de estados deve transicionar para o **nextState**, que é determinado pelo estado atual. Já as variáveis *timeSinceStart* e *timeSinceExit* são indicadores temporais de quanto tempo o estado em questão se tornou o estado atual e deixou de sê-lo, respectivamente.

Para a instanciação deste modelo, deve-se utilizar uma outra classe, geralmente denominada **EntityController**, que referencia tanto *StateMachine* quanto os *States* entre os quais ela deve transitar. Este *EntityController* geralmente é específico para cada funcionalidade, e portanto, não pode ser descrita de maneira genérica (por exemplo, tanto o jogador quanto o inimigo possuem uma classe **PlayerController** e **EnemyController**, respectivamente). No entanto, a responsabilidade para com a máquina de estados é a mesma: devem não só instanciar a *StateMachine* e os *States*, mas também, executar suas funções de **transição** e **atualização**.

Aplicação: Comportamento do Jogador - *PlayerController*

No caso do jogador, a máquina de estados serve para organizar a reação do jogo diante de diversos aspectos, mas o principal fator problemático são os *inputs*, que podem ser de diferentes tipos:

- **Input de movimentação**: recebido como um vetor 2D normalizado, indica a direção para a qual o jogador tentou se mover;
- **Input de ataque**: recebido como uma booleana, indica se o jogador tentou atacar ou não;

- **Input de *dash***: recebido, assim como o ataque, como uma booleana, indicando se o jogador tentou realizar um *dash* ou não;
- **Input de ção**: recebido, assim como o ataque, como uma booleana, indicando se o jogador tentou interagir ou não.

Diante desses possíveis valores de *inputs*, deve-se pensar qual é o comportamento final a ser atingido. Diante das especificações apresentadas na seção 4.1 e 4.2, entende-se que o desejado é desenvolver um sistema que permita o translado e a rotação do jogador, ataques e a interação do mesmo com objetos. Para tanto, foram determinados os seguintes estados que o jogador pode ter:

- **MoveState**: estado em que o jogador se move e pode mudar sua direção pelos *inputs*;
- **IdleState**: estado em que o jogador não coloca nenhum *input* – fica parado;
- **InteractState**: estado em que o jogador está interagindo com algum objeto: curas ou alavancas;
- **HoldAttackState**: estado em que o jogador segura o ataque. Pode se mover mas mais lentamente;
- **AttackState**: estado em que o jogador está de fato atacando. Não pode se mover;
- **DashState**: estado em que o jogador realiza um *dash*. Se move em uma única direção que não pode ser modificada.

Tendo descrito todos os estados, deve-se descrever a transição e a relação entre eles:

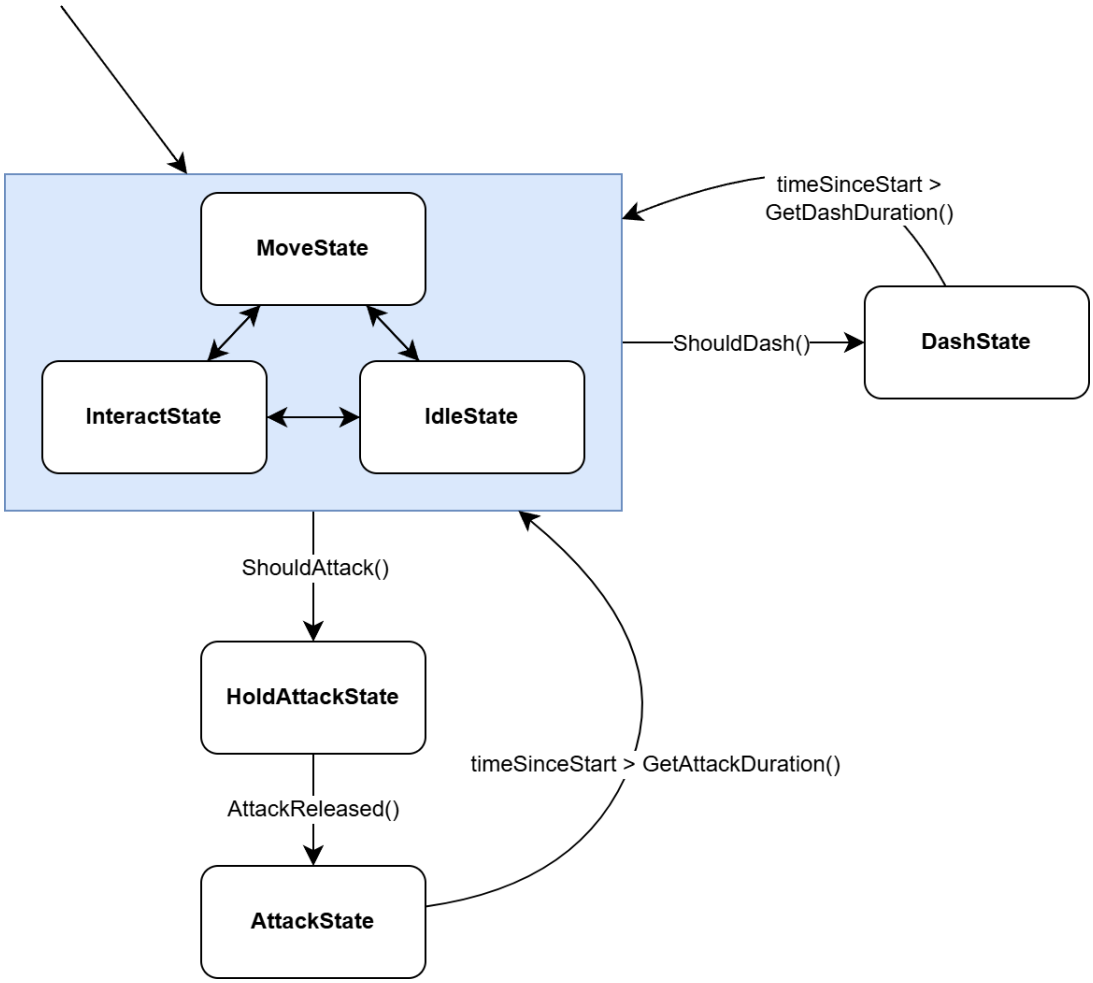

Figura 17: Representação da máquina de estados do *Player*

Neste diagrama, nota-se que existem estados representados dentro de uma caixa azul: **MoveState**, **InteractState** e **IdleState**. Denominaram-se estes estados como **estados transitáveis**, pois a troca entre os três depende **somente** dos *inputs* do jogador. Por exemplo, se o jogador está parado, ele imediatamente troca para o *MoveState* quando pressiona uma tecla de movimento, e imediatamente volta para o *IdleState* quando para de se mover Além disso, os estados **DashState** e **HoldAttackState** podem ser alcançados por qualquer um dos três caso as condições **ShouldDash()** ou **ShouldAttack()** se tornem verdadeiras. Os outros estados não envoltos pela caixa azul são considerados estados comuns, já que só podem ser atingidos caso as condições forem cumpridas.

Para analisar melhor o funcionamento desta máquina de estados, vamos realizar um simples fluxo de ataque. O passo a passo é o seguinte:

1. Jogador está em qualquer um dos *estados transitórios*: se movendo, interagindo ou parado;

2. Jogador pressiona o *input* de ataque – condição *ShouldAttack()* se torna verdadeira e a máquina de estados troca para o estado *HoldAttackState*;

3. Jogador segura o *input* de ataque por um período qualquer, se mantendo no estado *HoldAttackState*;

4. Jogador solta o *input* de ataque, condição *AttackReleased()* se torna verdadeira e a máquina de estados troca para o estado *AttackState*;

5. Tempo o suficiente passa para que a condição *timeSinceStart ¿ GetAttackDuration()* se torne verdadeira, e a máquina de estados troca para qualquer um dos estados transitórios (depende do input do jogador no momento);

Aplicação: Comportamento do Inimigo - *EnemyController*

Diferentemente da máquina de estados desenvolvida para o *Player*, o *Enemy* se diferencia pelo fato de que este não recebe nenhum *input*: todos os estados e suas transições devem ser decididos pelo código, e independem da ação do jogador. No entanto, para desenvolver um comportamento que lembre algum tipo de inteligência, o inimigo deve tomar decisões com base em informações do cenário e do jogador. Formalizando, a mudança dos estados depende das seguintes informações:

- A posição do jogador (e sua distância);

- Se o inimigo tem linha de visão (*Line of Sight*) com o jogador.

Além disso, o *EnemyController* também tem informações sobre a topografia da fase, sendo assim possível montar trajetos de movimentação para o personagem. Para realizar esta lógica de movimentação e de trajeto, foi utilizada a biblioteca do *Unity* **NavMesh** – no entanto, como este aspecto não é relativo à máquina de estados, não será aprofundado.

Tendo determinado as informações que o inimigo terá acesso, deve-se determinar os estados pelos quais a máquina de estados deve transitar:

- **AimState**: estado em que o personagem não se movimenta mas se rotaciona, para poder disparar projéteis na direção do jogador;

- **PivotState**: estado em que o personagem se movimenta realizando rotações em torno do jogador, mantendo à distância;
- **ChaseState**: estado em que o personagem persegue o jogador, encurtando a distância;
- **RetreatState**: estado em que o personagem se movimenta para longe do jogador, aumentando à distância;
- **DeathState**: estado atingido quando o personagem fica com 0 ou menos vida, e deve morrer.

De todos estes estados, o inimigo pode assumi-los por "vontade própria", com exceção do *DeathState*, que só se torna o estado atual quando a vida do inimigo fica em 0 ou menos. Quando este estado é atingido, uma pequena animação de morte é executada, e o inimigo desaparece do mapa.

Inicialmente, programou-se uma rotina que realizava a transição de estados de maneira **determinística**. No entanto, este método fazia com que a jogabilidade ficasse extremamente previsível e desinteressante: os inimigos sempre realizavam o mesmo comportamento se estivessem nas mesmas condições. Além disso, notou-se que se os inimigos tomassem "decisões perfeitas"(isto é, sempre recuar quando o jogador estiver muito perto, persegui-lo quando estiver muito longe ou sem linha de visão, etc.) o jogo se tornava muito difícil – os jogadores têm um tempo de reação muito maior do que a máquina. Ademais, nota-se que em combates com mais de um inimigo, se o sistema é determinístico, os dois inimigos sempre tomam as mesmas decisões, e acabam se aglomerando e agindo de forma idêntica, o que deixa o combate muito mais desinteressante.

Desta forma, uma solução para esta questão foi criar um sistema de transição de estados que envolvesse **aleatoriedade**, fazendo com que as trocas de estados não sejam tão eficientes sempre, mas criando uma maior variedade na jogabilidade e leve imprevisibilidade. O sistema desenvolvido faz com que, a cada troca de estados, gere-se um número aleatório de 0 a 1 (*float*, e portanto, decimal). Com este valor aleatório, avaliam-se as condições do inimigo (se ele possui linha de visão e a sua distância com o jogador) e determina-se o próximo estado a partir de uma tabela de probabilidades.

Além disso, para comportar trocas de estado aleatórias, foi necessário determinar uma taxa de frequência com a qual os estados seriam trocados. Isto é importante, pois se os estados transitassem aleatoriamente a cada atualização do jogo (60 vezes por segundo),

o comportamento do inimigo seria muito errático e não teria nenhuma constância ao longo de curtos períodos de tempo. Para remediar essa questão, determinou-se que as trocas de estado ocorreriam sempre em sincronia musical, para entrar na temática de ludomusicalidade e também fornecer um espaço de tempo longo o suficiente para que os comportamentos determinados pelos estados fossem significativos.

No entanto, não é desejado que o comportamento seja completamente aleatório: é mais interessante que o inimigo tome preferência por certos tipos de atitudes dependendo da situação em que se encontra. Por exemplo, se o personagem não tiver linha de visão com o jogador, é ideal que ele entre em estados que o façam conseguir ver o jogador (como o *ChaseState* ou o *PivotState*). Para desenvolver um sistema aleatório mas que ainda trabalhe com essas condições, criou-se uma tabela de probabilidades de transição de estado para diferentes condições em que o personagem inimigo pode se encontrar. Determinaram-se 6 casos, cujas condições são:

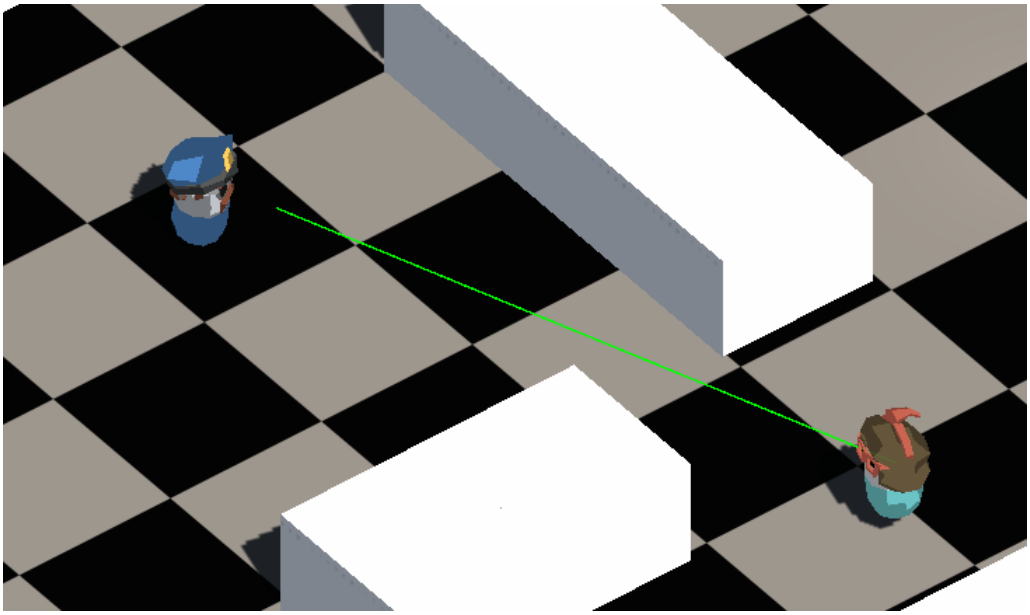

Figura 18: Representação de uma situação em que o inimigo tem linha de visão

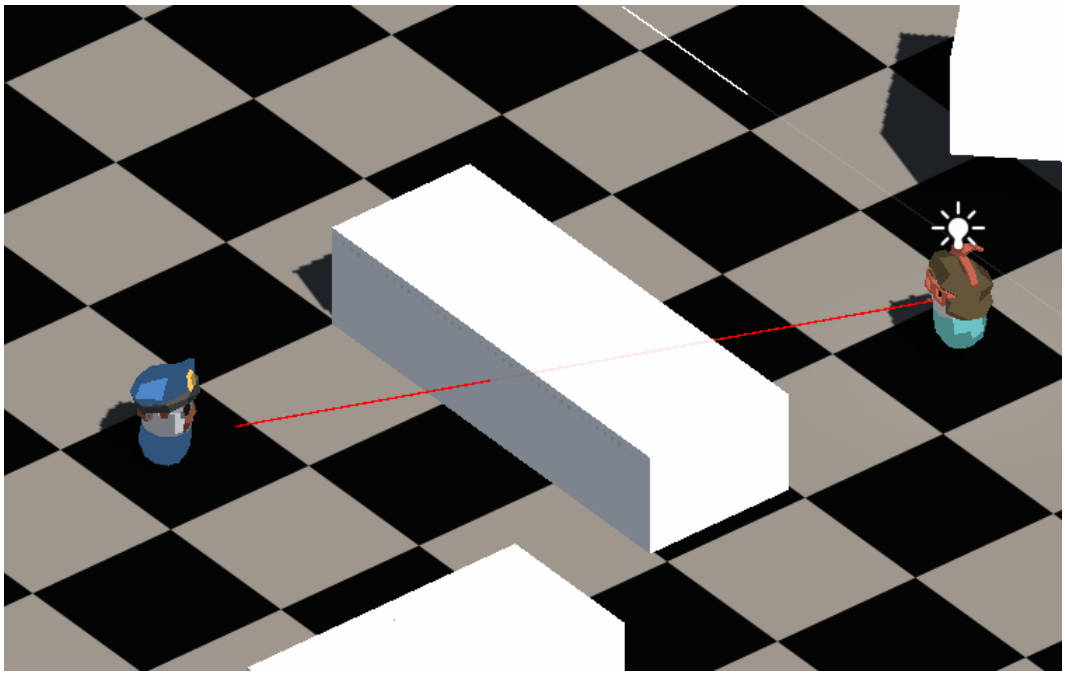

Figura 19: Representação de uma situação em que o inimigo **não** tem linha de visão

- **Ter ou não ter linha de visão**: duas possibilidades;
- **3 categorias de distância - *close*, *shoot* e *far*** : três possibilidades.

As categorias de distância definem um valor de distância próximo demais (*close*), adequado para o inimigo atirar projéteis (*shoot*) e longe demais (*far*). Para cada caso, foi montada uma linha na tabela que determina a probabilidade do próximo estado ser selecionado:

Linha de Visão	Distância	AimState	PivotState	ChaseState	RetreatState
Tem	*Close*	0%	30%	0%	70%
	Shoot	30%	50%	20%	0%
	Far	20%	30%	50%	0%
Não tem	*Close*	0%	80%	0%	20%
	Shoot	0%	60%	30%	10%
	Far	10%	10%	80%	0%

Figura 20: Tabela de probabilidades da seleção do próximo estado da FSM do inimigo.

Então, por exemplo, no caso em que o personagem inimigo tem linha de visão do jogador e está na categoria de distância *Shoot*, existe uma chance de 30% de que o

próximo estado seja o *AimState*, uma chance de 50% de que o próximo estado seja *PivotState*, e 20% de chance de ser *ChaseState*.

Nota-se que esse mapa de probabilidades é apenas uma versão das que foram desenvolvidas e que a jogabilidade foi interessante. Estes valores podem ser modificados a qualquer momento no código – ou até em tempo de jogo em iterações futuras. Por exemplo, se um inimigo toma dano mas sobrevive, é possível modificar o mapa de probabilidade para tomar uma atitude de "covarde", aumentando o peso do *RetreatState*. Muito pode ser experimentado com este sistema.

Pontos a melhorar: Escalabilidade

O sistema de máquina de estados finita, tanto para o jogador quanto para o inimigo, mostrou-se eficiente para programar comportamentos complexos. No entanto, em projetos de maior escala ou com equipes maiores, sua implementação poderia se tornar excessivamente complicada – e isso principalmente devido ao fato de que a classe *State* herda do *MonoBehaviour* e precisa ser acoplada a um objeto de jogo. Por exemplo, para adicionar um estado à máquina de estados é necessário realizar um passo a passo bem repetitivo:

1. Criar o código do novo estado herdando da classe base *State*;
2. Adicionar instância do estado no *EntityController* (*PlayerController* ou *EnemyController*);
3. Adicionar instância do componente no objeto em jogo no *Unity*;
4. Associar componente no objeto com o código do *EntityController*;
5. Adicionar transições dos estados para o novo estado.

Estas modificações, embora sejam simples de serem realizadas, não são triviais para pessoas que nunca olharam o código desenvolvido ou não tem formação em programação. Desta forma, o ideal seria desenvolver algum tipo de ferramenta automatize parte do processo:

- **Realizar os passos 2, 3 e 4 automaticamente**, já que são só instanciações e associações decorrentes do passo 1;

- **Criar interface gráfica de grafos que apresentasse todos os estados**, representando todos os estados como nós e as transições de estados como conexões entre os nós;

- **Permitir a representação da transição entre estados** na própria interface gráfica, associando cada transição com uma função booleana do próprio estado que é executada quando *isComplete* é verdadeiro.

Essas mudanças tornariam o processo mais ágil e simples de se ensinar para outras pessoas que eventualmente trabalhariam no projeto.

6 Validação e Playtesting

Tendo finalizado as mecânicas principais do projeto, é de extrema importância realizar processos de validação para melhorar funcionalidades com impacto positivo e remover as de impacto negativo. Segundo o que foi descrito por Schell na seção de Aspectos Conceituais 2.3, e também vindo da metodologia ágil SCRUM, é um ponto importantíssimo para a natureza iterativa do desenvolvimento.

O playtesting do projeto se deu em duas etapas: um primeiro *playtesting* com as mecânicas mais básicas implementadas no final do primeiro semestre de 2024, e outro *playtesting* com implementações finais e aplicações de *Level Design* no final do segundo semestre de 2024.

6.1 Metodologia

Para a coleta de resultados, a metodologia selecionada foi um misto de dois tipos de *playtesting*:

- ***Playtesting* em massa:** consiste em enviar o *link* do *site* do jogo para diversas pessoas (utilizando redes sociais) e pedindo para que estas respondam o formulário online logo em seguida. Este método permite que muitas pessoas interajam com o jogo, embora o *feedback* seja mais pontual;

- ***Playtesting* individual:** consiste em um processo semelhante a uma entrevista, em que se observa uma pessoa jogar o jogo pelo período de aproximadamente 20 minutos, vendo como a pessoa interage com todos os aspectos do jogo, além das reações emocionais apresentadas pelas pessoa.

Com estes dois métodos de coletas de resultado, pode-se ter uma visão mais geral com um espaço amostral maior, e uma visão mais individual com espaço amostral menor.

6.2 Primeiro Playtesting

O primeiro *playtesting* usou uma versão do jogo em que grande parte das mecânicas já estava presente:

- Movimentação e translado da câmera;
- Inimigos imóveis e sem mudança de estado;
- Ataques sincronizados;
- Dash do jogador (sem dash sincronizado).

Além disso, esta versão do *playtesting* também não contou com uma fase que foi cuidadosamente desenvolvida usando aspectos de *Level Design*. A fase desenvolvida foi criada mais rapidamente para poder realizar um teste das mecânicas, e seu esboço não usou nenhum parâmetro para sua concepção (em contrapartida com a fase da segunda etapa, que foi desenvolvida usando as proporções do *viewport*, ou tela, do jogador);

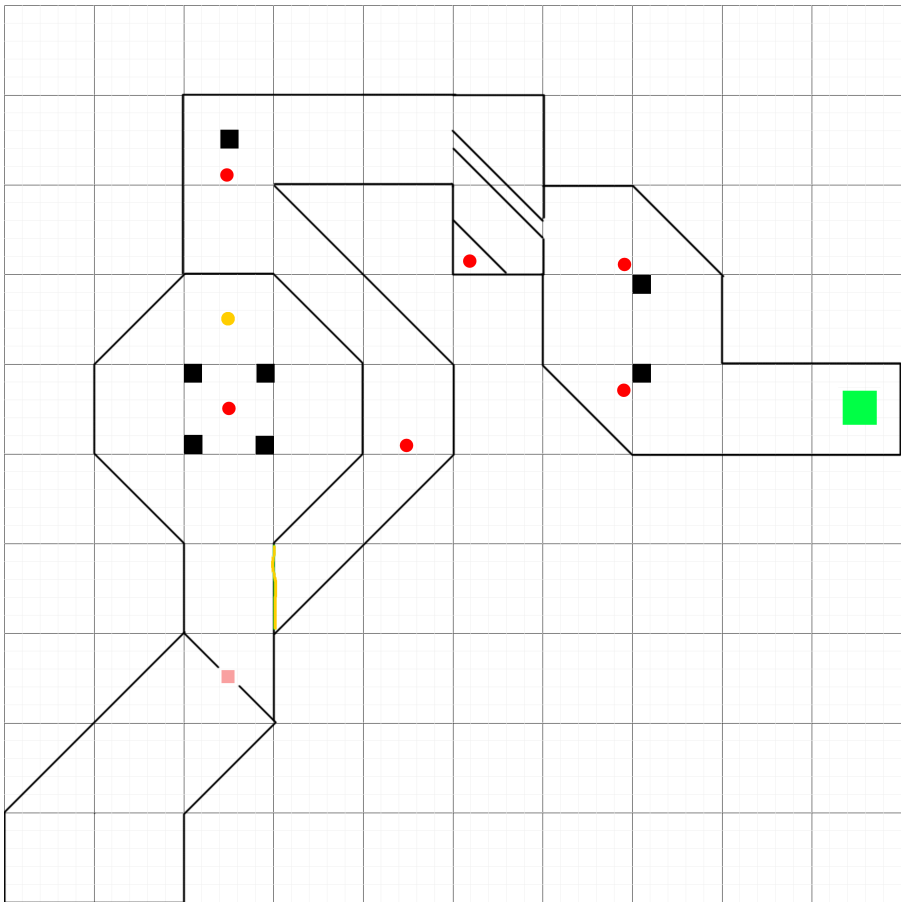

Figura 21: Imagem do esboço da primeira fase desenvolvida para o *playtesting*.

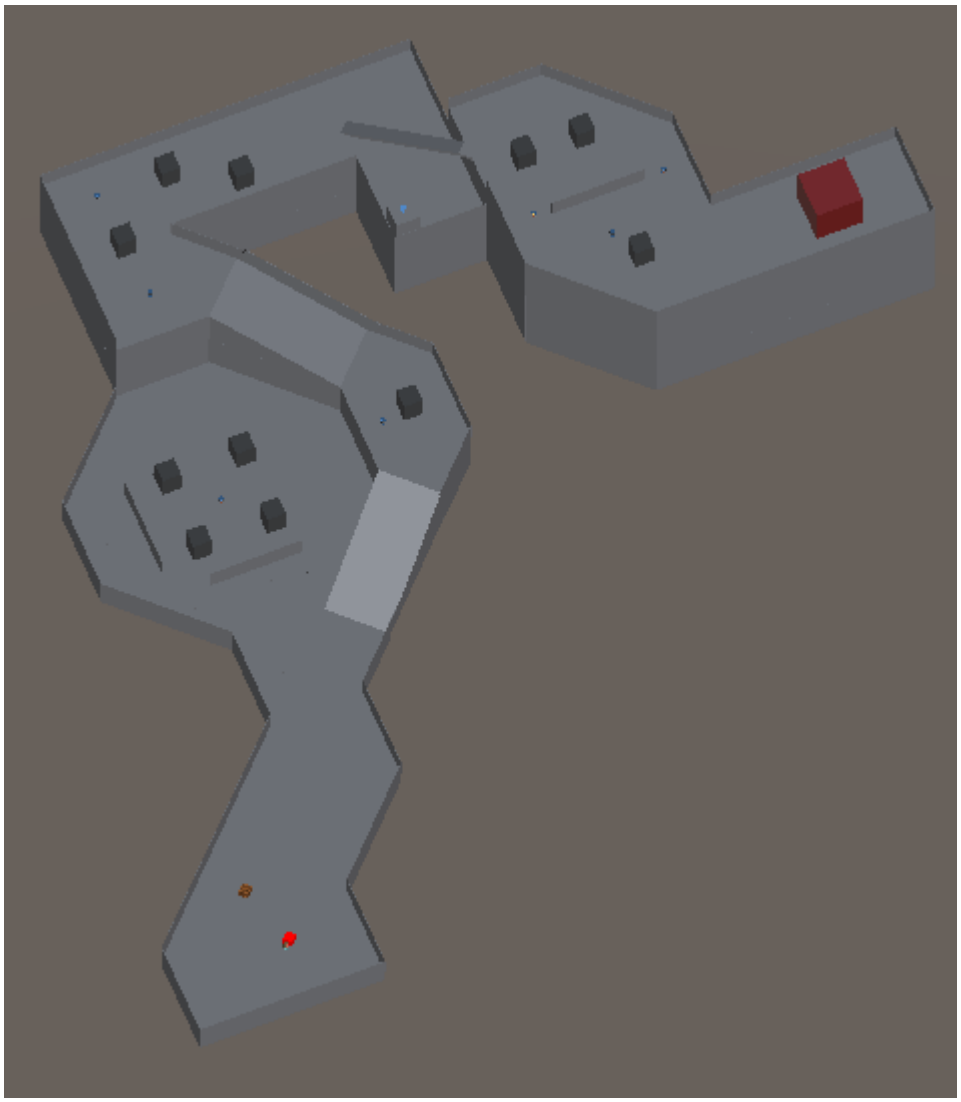

Figura 22: Imagem superior da primeira fase desenvolvida para o *playtesting*.

Como é possível ver na imagem, não existem portas nem alavancas nesta etapa. Os inimigos não precisam ser derrotados para o avanço na fase, e ao atingir no bloco vermelho final, o jogador finaliza o processo.

Resultado

Os gráficos do formulário na primeira fase foram os seguintes:

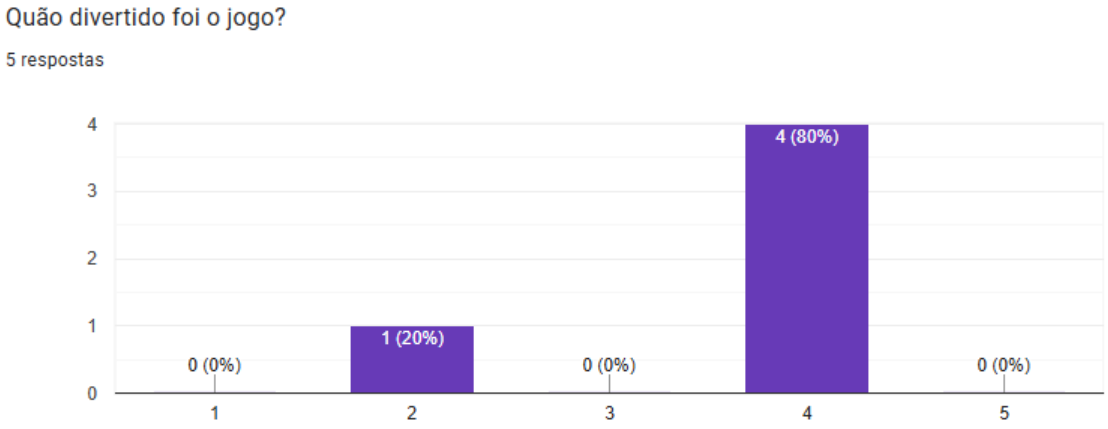

Figura 23: Gráfico sobre a **diversão** na primeira etapa de playtesting

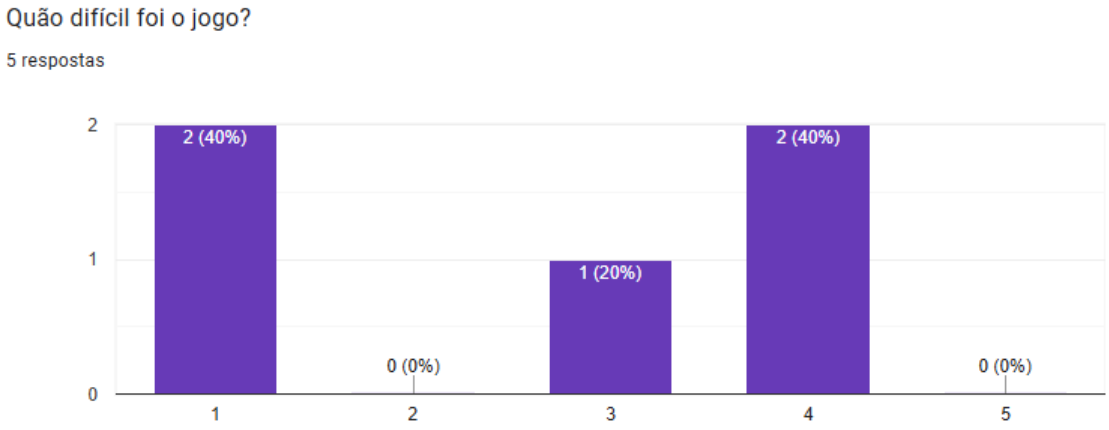

Figura 24: Gráfico sobre a **dificuldade** na primeira etapa de playtesting

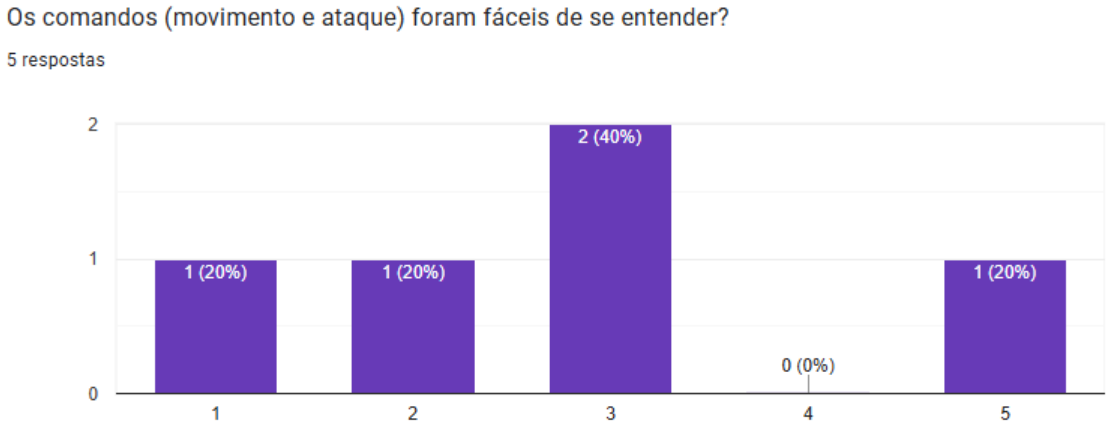

Figura 25: Gráfico sobre a **compreensão dos comandos gerais** na primeira etapa de playtesting

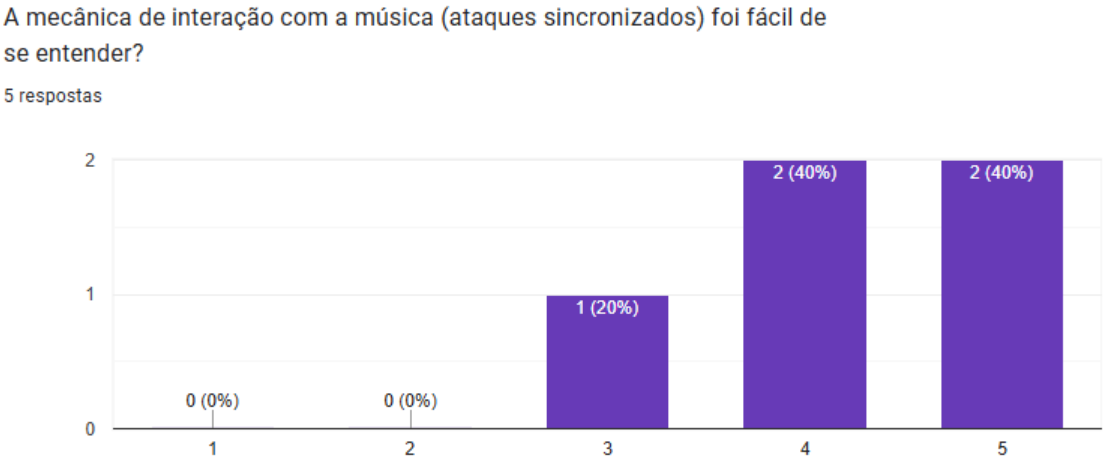

Figura 26: Gráfico sobre a **compreensão da mecânica de sincronização** na primeira etapa de playtesting

Os dados apontam que o fator de diversão está presente, já que a média da nota da diversão foi de 3,6. No entanto, também notam-se duas questões preocupantes: a dificuldade do jogo estava muito esparsa (difícil para alguns, fácil para outros), o que indica que houve uma confusão no propósito a ser cumprido no jogo, e os comandos de movimento e ataque foram difíceis de se entender. Estas duas questões foram importantes para o desenvolvimento das mudanças no segundo semestre do projeto.

Além disso, no que se devem aos comentários escritos e obtidos nos *playtestings* individuais desta primeira etapa, notou-se que as mecânicas foram bem implementadas e

tinham um potencial, dado que a maioria dos comentários positivos sobre o jogo foram relativos à sincronização dos ataques do jogador. No entanto, a falta de se ter uma fase que implementa conceitos de *Level Design* fez com que o foco dos comentários negativos se voltasse para o propósito da fase e um senso de direção. Para citar um dos comentários:

> "Qual a minha recompensa em limpar uma sala de inimigos ao invés de correr e chegar no fim de uma fase?"

Este comentário evidenciou que, embora as mecânicas fossem interessantes, o jogo necessita engajar o jogador de forma a desafiá-lo e exigir dele que aprenda as mecânicas, utilizando-as para superar os desafios. No geral, os comentários negativos foram mais voltados para essa confusão de propósito e direção, que foram identificados pelo fato de que o mapa não direcionava o jogador de forma alguma, e não limitava/recompensava o jogador por realizar certos tipos de ação. A segunda fase desenvolvida para o playtesting buscou remediar essas questões.

6.3 Segundo Playtesting

Tendo em mente os pontos levantados pela primeira etapa de *playtesting*, o foco no desenvolvimento para a segunda etapa foi reduzir a confusão das pessoas com o propósito da fase, implementando conceitos de *Level Design*, e reduzir a confusão das pessoas em relação aos comandos de movimento e ataque. Para alcançar esse objetivo foram selecionados os seguintes focos de desenvolvimento:

- **Implementação de tarefas obrigatórias na fase:** de forma a forçar os jogadores a cumprirem certas tarefas e aumentar o nível de dificuldade para encaixar no *canal de flow* descrito em 2.3. Esta mecânica foi implementada com alavancas, que abrem portas e levantam pontes, assim como combates obrigatórios;
- **Implementação de interface de instruções:** a funcionalidade apresentada em 4.3, de forma a demonstrar em escrito durante o jogo quais os comandos devem ser executados para ensinar o jogador a realizá-los;
- **Implementação de mecânicas de câmera mais complexas:** que mudam o foco e o ângulo da câmera, de modo a redirecionar o foco do jogador ou chamar a atenção dele para uma diversidade de situações (uma alavanca que deve ser aberta, uma zona com mais inimigos, etc);

- **Mudança na mecânica de ataque:** anteriormente, a mecânica de ataque não apresentava elemento direcional, e as pessoas não entendiam com facilidade que o ataque era projetado na direção do *mouse*. Para remediar isso, adicionou-se uma mecânica que direciona o jogador para o ponteiro *mouse* enquanto ele segura o ataque;

- **Aplicação de conceitos de *prospect & refuge*:** fazendo com que o *viewport* (isto é, a tela de visualização) do jogador englobe tudo que lhe é importante entender: saídas, locais de segurança e perigo – sinalizando ao jogador por meio do ambiente qual a situação em que ele se encontra;

- **Implementação do comportamento dos inimigos:** de forma a variabilizar a jogabilidade, tornar o combate mais interessante e levemente mais difícil.

A implementação dessas mecânicas foi o que guiou o desenvolvimento durante o segundo semestre, tudo em um processo iterativo com o professor orientador. Dessa forma, foi possível identificar, durante o desenvolvimento, pontos de fraqueza e de força do projeto.

A fase, especificamente, foi desenvolvida primeiramente em 2D usando a proporção da tela do jogador como medida principal. Em seguida, essa imagem 2D foi transformada em um ambiente 3D em *Unity*, que foi populado com as novas mecânicas desenvolvidas.

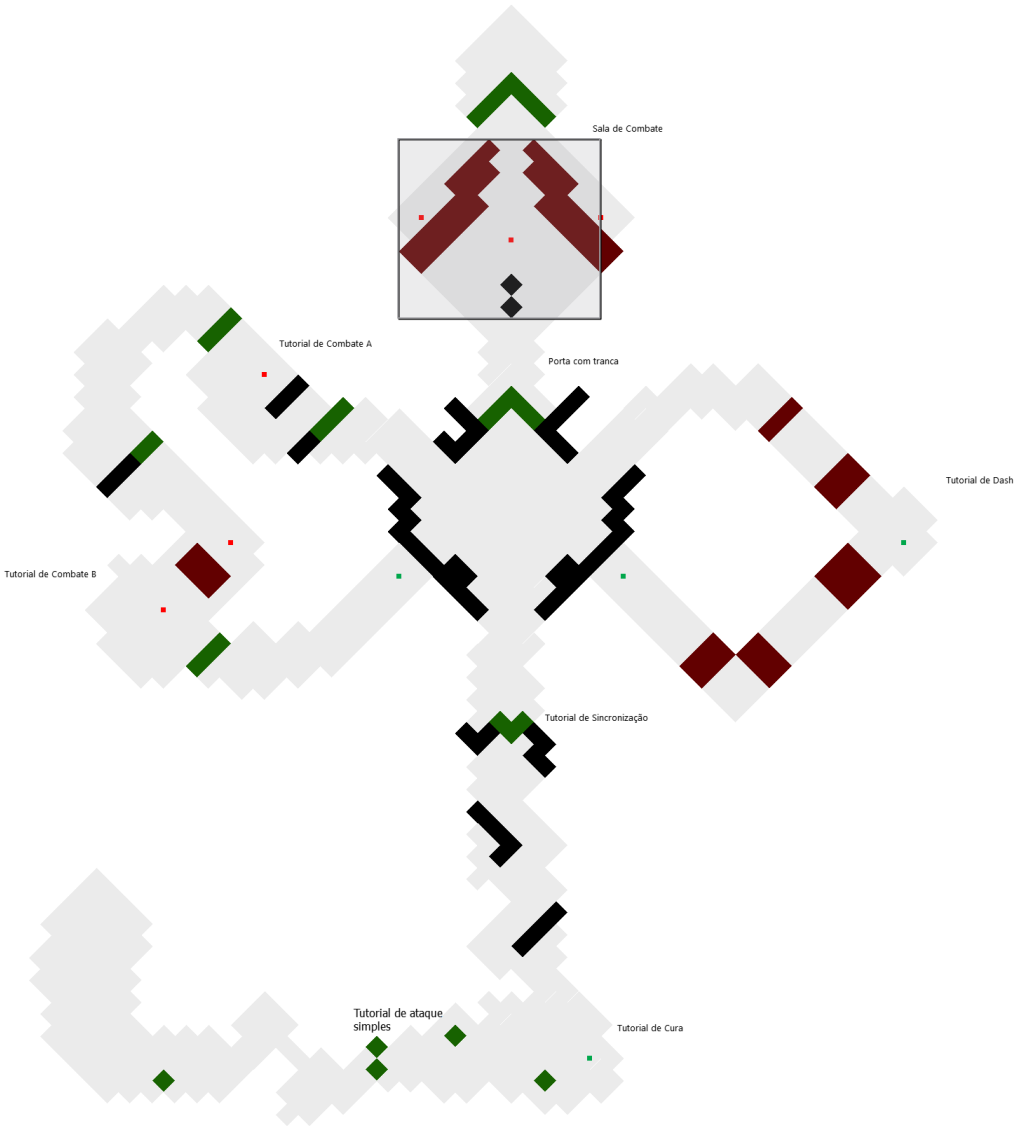

Figura 27: Imagem do esboço da segunda fase desenvolvida para o *playtesting*.

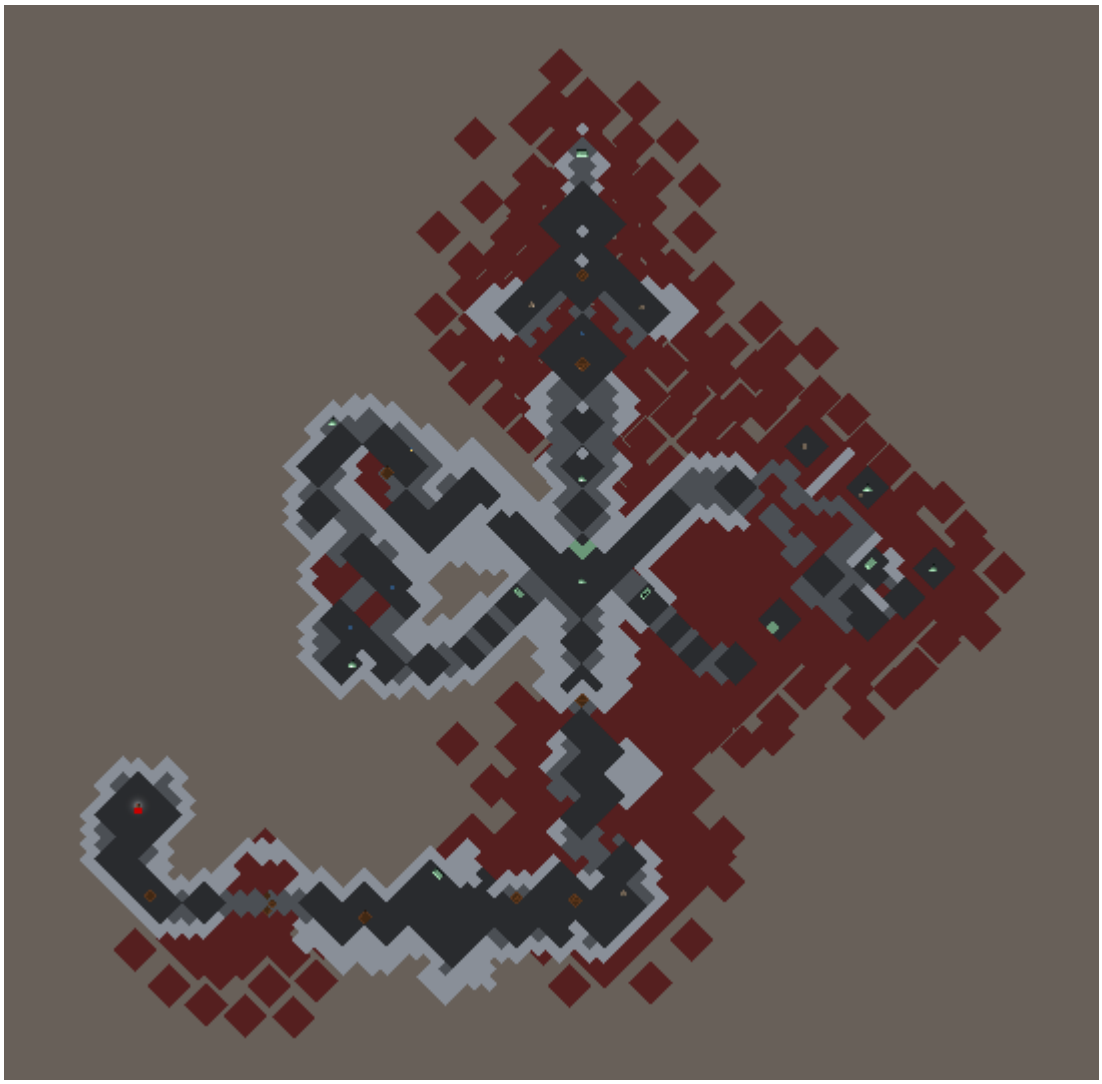

Figura 28: Imagem de captura de tela da segunda fase desenvolvida para o *playtesting*.

Resultado

Os gráficos do formulário na segunda fase foram os seguintes:

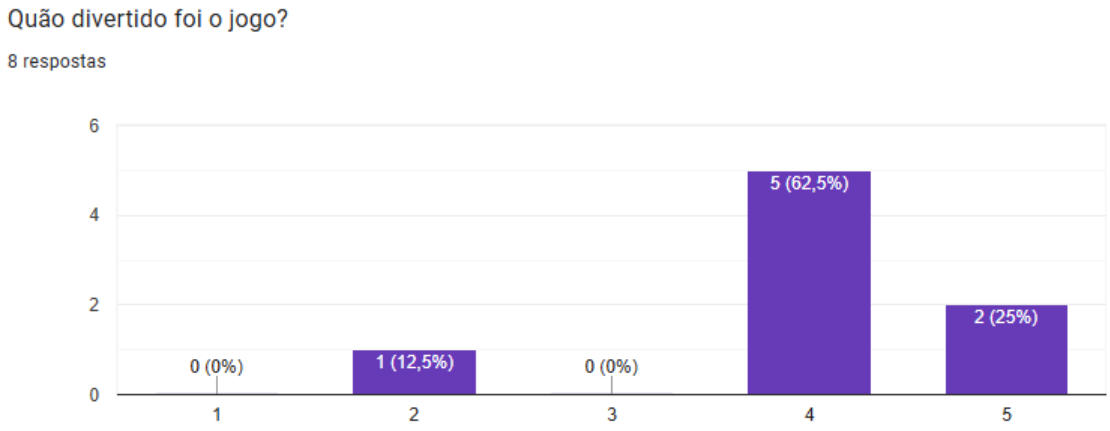

Figura 29: Gráfico sobre a **diversão** na segunda etapa de playtesting

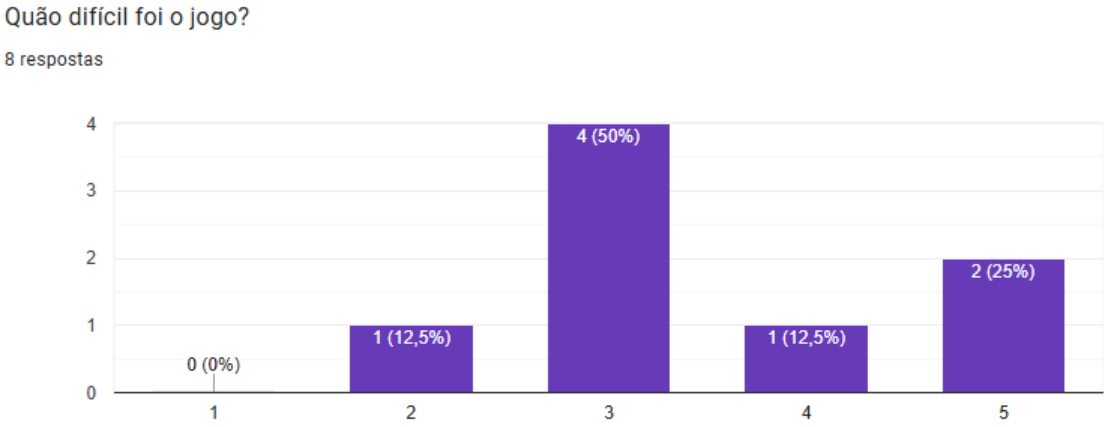

Figura 30: Gráfico sobre a **dificuldade** na segunda etapa de playtesting

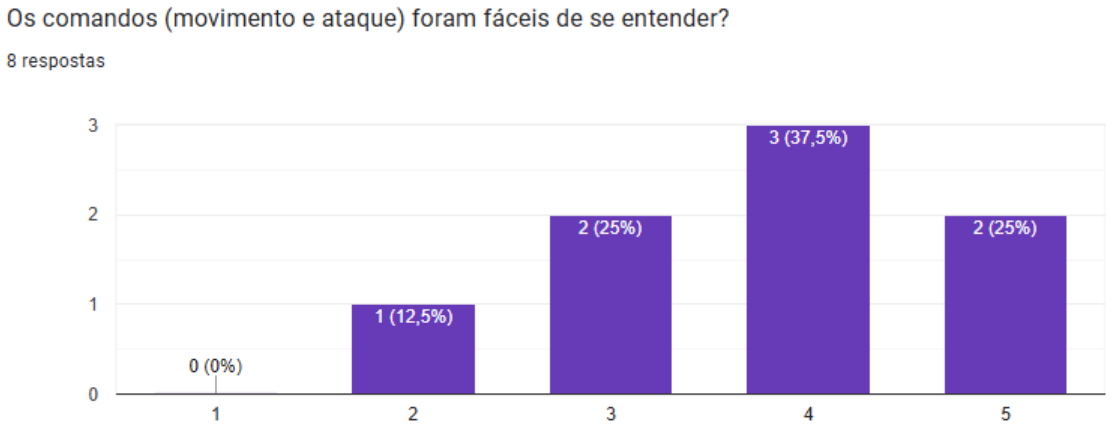

Figura 31: Gráfico sobre a **compreensão dos comandos gerais** na segunda etapa de playtesting

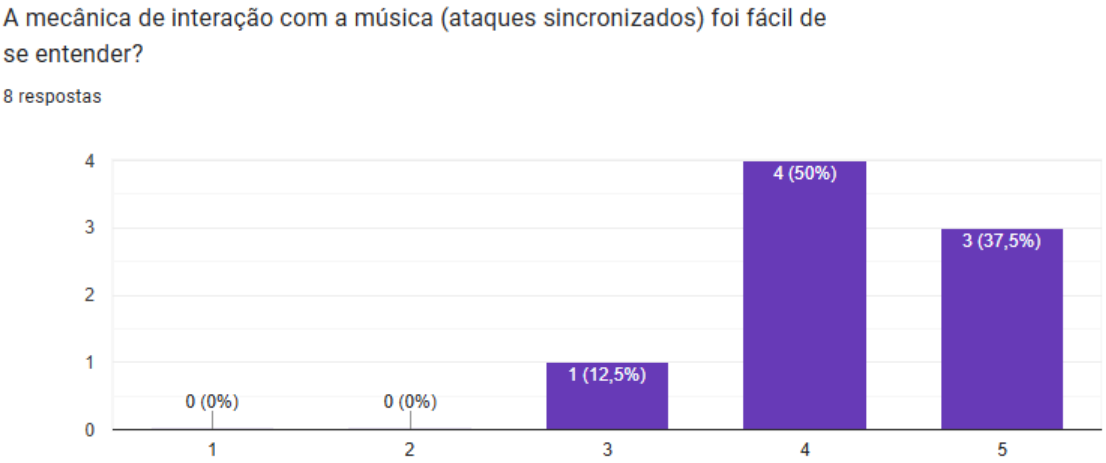

Figura 32: Gráfico sobre a **compreensão da mecânica de sincronização** na segunda etapa de playtesting

Os dados apontam que o fator de diversão na segunda fase está significativamente melhor, já que a média do valor foi de 3,6 para 4. Além disso, a dificuldade do jogo ficou menos diversa, e sua média ficou no valor de 3,5, que é considerado um pouco alto mas adequado para atingir o canal *flow*. Ademais, a compreensão dos comandos gerais melhorou significativamente, indo de 2,8 para o valor de 3,75 na média. Desta forma, entende-se que os conceitos de *Level Design* e as instruções foram efetivas em ensinar o jogador a aprender a jogar. A dificuldade de compreensão da mecânica musical não foi

muito afetada nesse período, mas seu valor já estava alto (de 4,2 para 4,25 na média), então é bom que este valor se manteve.

Já no que se deve aos comentários escritos e os obtidos nos *playtestings* individuais, notou-se que os comentários positivos se mantiveram no foco da mecânica musical, e os negativos no geral foram menos sobre a falta de propósito e confusão relativa a fase, o que é considerado ótimo (já que este foi o foco da segunda etapa de desenvolvimento). Ainda assim, os comentários negativos apontaram pontos de melhoria:

- **Falta de responsividade do movimento:** pois o jogador "escorrega"um pouco em sua movimentação, isto é, não desacelera de imediato;
- **Problema de visibilidade da fase:** em alguns momentos é difícil de se perceber "buracos", já que a visão isométrica confunde planos distantes e próximos;
- **Dificuldade do jogo**: já que é muito fácil morrer, os inimigos atacam frequentemente, e a punição é muito grande (não há *checkpoint*, então o jogador deve começar tudo de novo).

No entanto, já que a questão da confusão foi superada, entende-se que estas iterações foram bem-sucedidas. Algumas outras iterações foram executadas após essa coleta inicial: criou-se um *checkpoint* na fase (isto é, um ponto a partir do qual o jogador retornaria caso morresse), uma redução na vida dos inimigos e uma melhoria na movimentação do jogador. Mesmo assim, para validar que estas questões são resolvidas seria necessária mais uma etapa de testes – eis a natureza iterativa do projeto.

Para próximos passos ainda não realizados no projeto, seria interessante implementar um efeito visual que apresentasse o jogador com mais clareza uma diferenciação de planos mais próximos e mais distantes, para resolver a questão da visibilidade. Além disso, para embelezar o projeto, seria interessante melhorar o cenário e adicionar animações e indicadores de movimento tanto para o jogador quanto para os inimigos (outros pontos que foram comentados em menor escala no *playtesting*).

7 Conclusão

Este protótipo de jogo virtual que aplica o conceito de Ludomusicalidade avalia que o fator de diversão e imersão dos jogadores foi melhorada pela mecânica implementada, já que grande parte dos comentários positivos obtidos nas etapas de validação eram

relacionados a esta mecânica. No que se deve ao andamento do projeto, foi um longo trabalho iterativo, que contou com muitas atividades diversas – incluindo o levantamento de funcionalidades, especificações, planejamento, desenvolvimento, validação e iteração de melhorias. Estas atividades constituem a cerne do trabalho de Engenheiros de Software, e em conceito, de todas engenharias.

Ao fim do projeto, o aluno reconhece várias etapas em que o trabalho poderia ser melhorado. Em específico, na quantidade de iterações de *playtesting*. Ao iniciar a validação no meio do ano, reconheceu-se uma grande gama de problemas que poderia ser atacada para melhorar a jogabilidade e o produto como um todo. Contudo, o fato de apenas ter-se realizado duas etapas e não etapas iterativas ao longo do ano foi prejudicial para o projeto, gerando problemas como o alto nível de dificuldade do jogo na segunda etapa. Uma maior frequência iterativa de testagem forneceria ao projeto uma maior conexão com seu público-alvo, melhorando assim o resultado final.

Além disso, entende-se que o conhecimento técnico do aluno sobre *Design Patterns* no geral, mas especificamente no contexto de *Games* e arquiteturas de componentes foi muito melhorado. O sistema de eventos desenvolvido, assim como o de máquinas de estados, (embora possam ser melhorados) auxiliaram intensamente no desenvolvimento do projeto. Foram sistemas desenvolvidos no primeiro semestre do projeto mas continuaram sendo utilizados até o seu fim, o que mostra que foram construídos bem o suficiente para se manterem relevantes até o último momento.

Referências

1 WIKIPEDIA: Guitar Hero. 2005. Acessado no dia 17/03/2024. Disponível em: ⟨https://en.wikipedia.org/wiki/Guitar_Hero⟩.

2 WIKIPEDIA: Taiko no Tatsujin. 2001. Acessado no dia 17/03/2024. Disponível em: ⟨https://en.wikipedia.org/wiki/Taiko_no_Tatsujin⟩.

3 WIKIPEDIA: Just Dance. 2009. Acessado no dia 17/03/2024. Disponível em: ⟨https://en.wikipedia.org/wiki/Just_Dance_(video_game_series)⟩.

4 WIKIPEDIA: The Legend of Zelda. 1986. Acessado no dia 17/03/2024. Disponível em: ⟨https://en.wikipedia.org/wiki/The_Legend_of_Zelda⟩.

5 PHILLIPS, W. *A Composer's Guide to Game Music*. 2014 lts. ed. [S.l.]: The MIT Press, 2014.

6 MARTIN, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. [S.l.]: Prentice Hall, 2008.

7 DUNSTAN, J. *Level up your code with Game Programming Patterns*. 2021 lts. ed. [S.l.]: Autopublicado, 2021.

8 SUTHERLAND, K. S. e J. *O Guia do Scrum*. Novembro 2020. [S.l.]: Scrum.org and Scrum Inc., 2010.

9 SCHELL, J. *The Art of Game Design*. [S.l.]: CRC Press, 2008.

10 WIKIPEDIA: Hyper Light Drifter. 2016. Acessado no dia 24/11/2024. Disponível em: ⟨https://en.wikipedia.org/wiki/Hyper_Light_Drifter⟩.

11 APPLYING 3D Level Design Skills to the 2D World of Hyper Light Drifter. 2017. Acessado no dia 14/11/2024. Disponível em: ⟨https://www.youtube.com/watch?v=LFsMenc5Q8I⟩.

12 WIKIPEDIA: Street Fighter. 1987. Acessado no dia 17/03/2024. Disponível em: ⟨https://en.wikipedia.org/wiki/Street_Fighter⟩.